I2C Bus Control Using Simplified IIC (Arduino API)

Introduction

In this application note, a program written in a language such as Arduino is used to control the
temperature/humidity sensor HDC1080 connected to the I2C bus of the Pmod connector on the RL78/G14
Fast Prototyping Board (FPB).

Target Device

RL78/G14

When applying the sample program covered in this application note to another microcomputer, modify the
program according to the specifications for the target microcomputer and conduct an extensive evaluation of
the modified program.
1. Specifications

This application note describes how the temperature/humidity sensor HDC1080 is controlled in Fast mode (at 380 kbps) by using a program coded in a language such as Arduino via the I2C bus of Pmod connector 1 on the FPB. This application note also describes how the data obtained from the temperature/humidity sensor is displayed on the LCD indicator (16 chars x 2 lines) in Standard mode (at 85 kbps) via the I2C bus of the Arduino connector.

Data is obtained from the temperature/humidity sensor and displayed on the LCD indicator on a regular basis at one-minute intervals or by pressing a switch.

Pmod connectors 1 and 2 on the RL78/G14 FPB use different pin positions for SCL and SDA signals on the I2C bus from those pin positions on the standard Pmod connector. To connect to an I2C module that supports Pmod, as shown in Figure 1.1, prepare a separate conversion board that exchanges SCL and SDA signals.

The RL78/G14 FPB is equipped with two Pmod connectors. The temperature/humidity sensor HDC1080 is controlled by using Wire1 allocated to Pmod connector 1.

To use Wire2 allocated to Pmod connector 2, processing to enable Wire2 must be performed. In r_cg_userdefine.h, comment out line 50, and then uncomment line 51 and change the name of the API function used in AR_SKETCH.c from Wire1 to Wire2 (see the lines surrounded in a red frame in Figure 1.2).

<table>
<thead>
<tr>
<th>Peripheral Function</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital input</td>
<td>Reads the status of the switch (SW_USR).</td>
</tr>
<tr>
<td>I2CA0</td>
<td>Controls the LCD indicator via the I2C bus.</td>
</tr>
<tr>
<td>I2C00</td>
<td>Controls the sensor via the I2C bus of the Pmod1 connector.</td>
</tr>
<tr>
<td>I2C20</td>
<td>Controls the sensor via the I2C bus of the Pmod2 connector.</td>
</tr>
<tr>
<td>Timer array unit</td>
<td>Measures the elapsed time.</td>
</tr>
</tbody>
</table>
1.1 Program Execution Environment

In this application note, a program in an Arduino language is executed in a development environment specific to the RL78 family. A conceptual diagram of the program execution environment is shown in Figure 1.3.

![Program execution environment diagram]

Table 1.2 Library functions (1/3)

<table>
<thead>
<tr>
<th>Item</th>
<th>Library Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital I/O</td>
<td>pinMode(pin, mode)</td>
<td>Specifies the operation mode (input mode/output mode/input mode with internal pull-up resistor enabled) for the pin specified by &quot;pin&quot;.</td>
</tr>
<tr>
<td></td>
<td>digitalWrite (pin, value)</td>
<td>Sets the pin specified by &quot;pin&quot; to the state specified by &quot;value&quot; (high level/low level).</td>
</tr>
<tr>
<td></td>
<td>digitalRead(pin)</td>
<td>Reads out the state of the pin specified by &quot;pin&quot;.</td>
</tr>
<tr>
<td>Time control</td>
<td>millis()</td>
<td>Returns, in millisecond units, the time from the start of program execution to the present time.</td>
</tr>
<tr>
<td></td>
<td>micros()</td>
<td>Returns, in microsecond units, the time from the start of program execution to the present time.</td>
</tr>
<tr>
<td></td>
<td>delay (ms)</td>
<td>Stops the program for the specified time in millisecond units.</td>
</tr>
<tr>
<td></td>
<td>delayMicroseconds (us)</td>
<td>Stops the program for the specified time in microsecond units.</td>
</tr>
</tbody>
</table>
### Table 1.3 Library functions (2/3)

<table>
<thead>
<tr>
<th>Item</th>
<th>Library Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C control (Wire)</td>
<td>Wire.begin()</td>
<td>Initializes IIC0 and connects to the I^2^C bus as the master.</td>
</tr>
<tr>
<td></td>
<td>Wire.requestFrom(saddr7, bytes, stop)</td>
<td>Receives data with the size specified by &quot;bytes&quot; from the specified slave. The Wire.available() function is used to obtain the number of bytes and the Wire.read() function is used to read data.</td>
</tr>
<tr>
<td></td>
<td>Wire.requestFrom(saddr7, bytes)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wire.beginTransmission(saddr7)</td>
<td>Prepares for sending data to the specified slave. Then, the Wire.write() function is used to enqueue data and the Wire.endTransmission() function is used to send the data.</td>
</tr>
<tr>
<td></td>
<td>Wire.endTransmission(stop)</td>
<td>Sends data from the queue to the slave, and then ends processing.</td>
</tr>
<tr>
<td></td>
<td>Wire.write(data)</td>
<td>Enqueues data that is to be sent to the slave.</td>
</tr>
<tr>
<td></td>
<td>Wire.available()</td>
<td>Uses the Wire.read() function to check the number of bytes that can be read.</td>
</tr>
<tr>
<td></td>
<td>Wire.read()</td>
<td>Reads receive data from the slave.</td>
</tr>
<tr>
<td>Simplified IIC control (Wire1)</td>
<td>Wire1.begin()</td>
<td>Initializes IIC0 (Wire1) connected to Pmod connector 1 and connects to the I^2^C bus as the master.</td>
</tr>
<tr>
<td></td>
<td>Wire1.requestFrom(saddr7, bytes, stop)</td>
<td>Receives data with the size specified by &quot;bytes&quot; from the specified slave. The Wire1.available() function is used to obtain the number of bytes and the Wire1.read() function is used to read data.</td>
</tr>
<tr>
<td></td>
<td>Wire1.requestFrom(saddr7, bytes)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wire1.beginTransmission(saddr7)</td>
<td>Prepares for sending data to the specified slave. Then, the Wire1.write() function is used to enqueue data and the Wire1.endTransmission() function is used to send the data.</td>
</tr>
<tr>
<td></td>
<td>Wire1.endTransmission(stop)</td>
<td>As Wire1, sends data from the queue to the slave, and then ends processing.</td>
</tr>
<tr>
<td></td>
<td>Wire1.write(data)</td>
<td>Enqueues data that is to be sent to the slave of Wire1.</td>
</tr>
<tr>
<td></td>
<td>Wire1.available()</td>
<td>Uses the Wire1.read() function to check the number of bytes that can be read.</td>
</tr>
<tr>
<td></td>
<td>Wire1.read()</td>
<td>Dequeues data that was received from the slave of Wire1.</td>
</tr>
</tbody>
</table>

**Note:** The slave function of the I^2^C bus is not supported. For some functions, a limit is placed on the arguments that can be specified or the number of arguments that can be specified.
### Table 1.4 Library functions (3/3)

<table>
<thead>
<tr>
<th>Simplified IIC control (Wire2)</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wire2.begin()</td>
<td></td>
<td>Initializes IIC00 (Wire2) connected to Pmod connector 1 and connects to the I²C bus as the master.</td>
</tr>
<tr>
<td>Wire2.requestFrom(saddr7, bytes, stop)</td>
<td></td>
<td>Receives data with the size specified by &quot;bytes&quot; from the specified slave. The Wire2.available() function is used to obtain the number of bytes and the Wire2.read() function is used to read data.</td>
</tr>
<tr>
<td>Wire2.requestFrom(saddr7, bytes)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wire2.beginTransmission(saddr7)</td>
<td></td>
<td>Prepares for sending data to the specified slave. Then, the Wire2.write() function is used to enqueue data and the Wire2.endTransmission() function is used to send the data.</td>
</tr>
<tr>
<td>Wire2.endTransmission(stop)</td>
<td></td>
<td>As Wire2, sends data from the queue to the slave, and then ends processing.</td>
</tr>
<tr>
<td>Wire2.write(data)</td>
<td></td>
<td>Enqueues data that is to be sent to the slave of Wire2.</td>
</tr>
<tr>
<td>Wire2.available()</td>
<td></td>
<td>Uses the Wire2.read() function to check the number of bytes that can be read.</td>
</tr>
<tr>
<td>Wire2.read()</td>
<td></td>
<td>Dequeues data that was received from the slave of Wire2.</td>
</tr>
</tbody>
</table>

#### Note
The slave function of the I²C bus is not supported. For some functions, a limit is placed on the arguments that can be specified or the number of arguments that can be specified.

### 1.2 Program (Sketch) Configuration

Subfolders are prepared for each integrated development environment below the folder (workspace) in which the project is stored. In the folders for each of the integrated development environments the files are stored that are used in the RL78 family development environment.

In each sketch subfolder, AR_SKETCH.c is stored which is the Arduino language program (sketch). When viewing or modifying sketch, the "AR_SKETCH.c" file in the sketch subfolder is used.

### 1.3 Preparations for Project Startup

Preparations for project startup are different depending on the integrated development environment used. For details, refer to the following application note.

RL78 Family Arduino API Introduction Guide (R01AN5413)
1.4 Definitions in the Program (sketch)
Definitions in the program (sketch) are indicated in Figure 1.4.

```
1) int swPin = 18;  // assign D18 pin to swPin for SW_USER.

2) #define SLADDR_HDC1080  (0x40)  // I2C bus slave address of HDC1080
#define MINUTE_   (60000/16)  // 1 minute divided by 16 milli sec
unsigned int old_time = 0x0000;  // previous time(milli sec.)
unsigned char hdc1080_buff[4] =
| 0x00, 
| 0x00, 
| 0x00, 
| 0x00 |

3) unsigned char humid;
   int temp;
   // Humidity data(unit %)
   // Temperature data (0.1 degree unit).
```

Figure 1.4 Program definition details

1) "18" is set for the swPin pin that controls the on-board SW_USR switch so that the pin is assigned to D18.

2) Then, the following items are defined: the "old_time" 16-bit variable to check the elapsed time (in milliseconds), the "hdc1080_buff" 4-byte array for communication use to control the HDC1080 sensor, the "humid" variable to store the humidity data obtained, and the "temp" variable to store temperature data in units of 0.1 degrees.

3) In the display data area for the LCD indicator, the following 40-byte arrays are defined: the "disp_line1" variable to store the data for line 1, the "disp_line2" variable to store the data for line 2. In addition, the "count16ms" counter to obtain 1 minute by counting 16-ms intervals and the "sw_work" variable to check the switch are defined.

The "Wire1" API_Wire-type structure is used to reference any objects that are defined by AR_LIB_WIRE1.c that provides Wire-related API functions.
1.5 Initial Setting Processing

The initial settings section of the program (sketch) is shown in Figure 1.5.

The *setup* function specifies that the switch input pin be used for input. Also, IICA0 and IIC00 are set as the PC bus master. Then, the initial display data is set for the LCD indicator.

```c
void setup(void){
    // put your setup code here, to run once:
    pinMode(swPin, INPUT); // set D18pin to input mode
    Wire.begin(); // set IICA0 for I2C bus master
    Wire.begin(); // set IIC00 for I2C bus master
    init_LCD(); // initialize LCD display
    disp_line1[14] = D0; // set degreeC character of LCD
    printf_LCD((uint8_t *__)disp_line1, (uint8_t *__)disp_line2);
}
```

Figure 1.5 Initial setting processing section

1.6 Main Processing Part

The leading section of the main processing, which is executed repeatedly, is shown in Figure 1.6. When preparations for project startup have been set correctly, the sketch will be downloaded, then executed until the leading section of the main processing.

```c
void loop(void){
    // put your main code here, to run repeatedly:
    static char m_time = 1;
    char work; char sw_data;
    unsigned int time_work; int work_int;
    unsigned long long work;
    /*-----------------------------------------------
    * wait for 10milli seconds interval.
    */
    time_work = (int)(millis() & 0xFFFF); // read milli sec data
    if (old_time != time_work) // check 16 milli seconds passed
}
```

Figure 1.6 The leading section of the main process
1.7 Data Processing Performed by HDC1080

Normally, HDC1080 is placed in sleep mode. To obtain the humidity data and temperature data, a measurement request must be issued.

The measurement request is issued by writing the slave address of HDC1080 followed by 0x00. To obtain 14-bit humidity/temperature data from HDC1080, measurement takes 6.5 ms (TYP.) to complete.

In this application note, after issuing an measurement request, the software waits for 32 ms and then reads the measurement results.

shows an example of 4-byte data read from HDC1080. In the example, temperature data is indicated in red and humidity data is indicated in blue.

Temperature is obtained by using the following expressions. Expression ① is used to get 16-bit length data. ② is used to convert the resulting data to a value in 0.1°C degrees. ③ is used to subtract 40°C as the offset.

The temperature in units of 0.1°C degrees is obtained in this way.

1. long_work = ( hdc1080_buff[0] * 0x100UL + ( hdc1080_buff[1] );
2. long_work *= 1650; // multiply 10 times of Maximum temperature
3. temp = (int)((long_work >> 16) - 400); // adjust offset (40degreeC)

Humidity is obtained by using the following expressions. Expression ① is used to get 16-bit length data. Expression ② is used to multiply the value by 100. ③ is used to obtain the humidity data. The humidity data in percentage is obtained in this way.

1. long_work = ( hdc1080_buff[2] * 0x100UL + hdc1080_buff[3] );
2. long_work *= 100UL; // get percentage
3. humid = (unsigned char)(long_work >>16); // get humidity
2. Operating Conditions

The operation of the sample code provided with this application note has been tested under the following conditions.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller used</td>
<td>RL78/G14 (R5F104MLAFB : RL78G14_FPB)</td>
</tr>
<tr>
<td>Operating frequency</td>
<td>• High-speed on-chip oscillator clock ( f_{\text{IH}} ): 32 MHz&lt;br&gt;• CPU/peripheral hardware clock: 32 MHz</td>
</tr>
<tr>
<td>Operating voltage</td>
<td>3.3 V (can be operated at 2.75 V to 5.5 V)&lt;br&gt;LVD operation: Reset mode&lt;br&gt;LVD detection voltage ( V_{\text{LVD}} )&lt;br&gt;At rising edge: 2.81 V typ. (2.76 V to 2.87 V)&lt;br&gt;At falling edge: 2.75 V typ. (2.70 V to 2.81 V)</td>
</tr>
<tr>
<td>Integrated development&lt;br&gt;environment</td>
<td>Renesas Electronics&lt;br&gt;CS+ for CC V8.05.00&lt;br&gt;Renesas Electronics&lt;br&gt;e² studio V7.7.0&lt;br&gt;IAR Systems&lt;br&gt;IAR Embedded Workbench for RL78</td>
</tr>
<tr>
<td>C compiler</td>
<td>Renesas Electronics&lt;br&gt;CC-RL V1.10.00&lt;br&gt;IAR Systems&lt;br&gt;IAR C/C++ Compiler v4.20.1 for RL78</td>
</tr>
</tbody>
</table>
3. Related Application Notes

The application notes related to this application note are shown below.
Refer to these together with this application note.

- RL78 Family Arduino API Introduction Guide (R01AN5413)
- RL78/G14 Onboard LED Flashing Control (Arduino API) (R01AN5384)
4. Hardware

4.1 Example of Hardware Configuration

shows the hardware (FPB) that is used in this application note.

Figure 4.1 Hardware configuration example

Note: This conceptual diagram is simplified in order to summarize the connections.

As the power supply voltage, 3.3 V is supplied via USB.

4.2 List of Pins Used

Table 4.1 shows the pins used and their functions.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Port Name</th>
<th>I/O</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>D14</td>
<td>P61</td>
<td>Input/Output</td>
<td>SDA (Data line of I2C-bus)</td>
</tr>
<tr>
<td>D15</td>
<td>P60</td>
<td>Input/Output</td>
<td>SCL (Clock line of I2C-bus)</td>
</tr>
<tr>
<td>-</td>
<td>P50</td>
<td>Input/Output</td>
<td>SDA of PMOD1 connector (Data line of I2C-bus)</td>
</tr>
<tr>
<td>-</td>
<td>P30</td>
<td>Input/Output</td>
<td>SCL of PMOD1 connector (Clock line of I2C-bus)</td>
</tr>
<tr>
<td>-</td>
<td>P14</td>
<td>Input/Output</td>
<td>SDA of PMOD2 connector (Data line of I2C-bus)</td>
</tr>
<tr>
<td>-</td>
<td>P15</td>
<td>Input/Output</td>
<td>SCL of PMOD2 connector (Clock line of I2C-bus)</td>
</tr>
<tr>
<td>D18</td>
<td>P137</td>
<td>Input</td>
<td>Switch (SW_USR) input</td>
</tr>
</tbody>
</table>
5. Software

5.1 Summary of Operation

In this application note, when the software completes initial setup (pin setup) and the main processing (loop) starts, the LCD indicator is placed in the initial display status.

Data is obtained from the temperature/humidity sensor HDC1080 on a regular basis at one-minute intervals or by pressing a switch. The temperature and humidity are calculated from the obtained data, and then the calculation results are displayed on the LCD indicator.

Details are explained in (1) to (2) below.

(1) Use the "setup" function to specify the settings of the pins to be used.
- The software sets the read pin of the on-board SW_USR (swPin) to digital input.
- To control the I2C bus by using the D14 and D15 pins, the software sets IIC0A0 to Master.
- To control the I2C bus by using Pmod connector 1, the software sets IIC00.
- The software initializes the LCD indicator connected to the I2C bus to reset the indication.

(2) Use the "loop" function to perform the main processing.
- The software obtains the data of 12 bits (in units of 16 milliseconds) from bits 15 to 4 as the time elapsed since startup in milliseconds.
- The software checks whether the obtained data has changed from the old data (old_time).
- If the data has not changed, the software terminates processing and returns control to the beginning of the "loop" function.
- If the data has changed (16 milliseconds elapsed), the software replaces the data in old_time by the obtained data.
- The software starts the counter that is reset at one-minute intervals (0xEA6).
- The software checks the status of the switch connected to D18.
- If the switch is not pressed and one minute has not elapsed, the software returns control to the beginning of the "loop" function. Note
- If the switch is pressed or one minute has elapsed, the software measures the temperature and humidity. Note
- The temperature/humidity sensor HDC1080 exits standby mode and the software starts measurement.
  ◆ The software waits for about 16 milliseconds until the data that can be obtained becomes stable.
  ◆ The software reads data from the sensor.
  ◆ The software calculates the temperature and humidity from the data that was read.
- The software transfers the calculation results to the LCD indicator.
- The software returns control to the beginning of the "loop" function.

Note: Immediately after startup, the software uses the sensor to measure the temperature and humidity.
5.2 List of Constants

Table 5.1 shows constants that are used in the sample code.

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Setting Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>swPin</td>
<td>18</td>
<td>Number of the pin from which SW_USR is read</td>
</tr>
<tr>
<td>DUMMY_DATA</td>
<td>0xFF</td>
<td>Data to be written for starting reception during master reception</td>
</tr>
<tr>
<td>RELEASE</td>
<td>1</td>
<td>Specifies that stop conditions are generated when communication is completed.</td>
</tr>
<tr>
<td>RESTART</td>
<td>0</td>
<td>Specifies that restart conditions are generated when communication is completed.</td>
</tr>
<tr>
<td>SLADDR_HC1080</td>
<td>0x40</td>
<td>Slave address of the sensor (7 bits)</td>
</tr>
<tr>
<td>SLADDR_LCD</td>
<td>0x50</td>
<td>Slave address of the LCD indicator (7 bits)</td>
</tr>
<tr>
<td>COMBYTE</td>
<td>0x00</td>
<td>Data specifying that a command is sent to the LCD indicator</td>
</tr>
<tr>
<td>DATABYTE</td>
<td>0x80</td>
<td>Data specifying that data is transferred to the LCD indicator</td>
</tr>
<tr>
<td>CLRDISP</td>
<td>0x01</td>
<td>Command that clears the indication of the LCD indicator</td>
</tr>
<tr>
<td>HOMEPOSI</td>
<td>0x02</td>
<td>Moves the cursor of the LCD indicator to the home position.</td>
</tr>
<tr>
<td>LCD_Mode</td>
<td>0x38</td>
<td>Specifies that each character is displayed with 5x8 dots on two lines.</td>
</tr>
<tr>
<td>DISPON</td>
<td>0x0F</td>
<td>Turns on and blinks the cursor.</td>
</tr>
<tr>
<td>ENTRY_Mode</td>
<td>0x06</td>
<td>Moves the display position each time one character is transferred.</td>
</tr>
<tr>
<td>LOOPLIMIT</td>
<td>1000</td>
<td>Sets the maximum number of times starts and stops can be detected to 1,000.</td>
</tr>
<tr>
<td>SUCCESS</td>
<td>0x00</td>
<td>Indicates that processing of the I2C bus terminated normally.</td>
</tr>
<tr>
<td>BUS_FREE</td>
<td>0x00</td>
<td>The I2C bus is idle.</td>
</tr>
<tr>
<td>BUS_ERROR</td>
<td>0x8F</td>
<td>The I2C bus could not be secured.</td>
</tr>
<tr>
<td>GET_BUS</td>
<td>0x10</td>
<td>The I2C bus was secured.</td>
</tr>
<tr>
<td>GET_BUS4TX</td>
<td>0x20</td>
<td>The I2C bus was secured for transmission.</td>
</tr>
<tr>
<td>TX_MODE</td>
<td>0x30</td>
<td>Transmission mode</td>
</tr>
<tr>
<td>TX_END</td>
<td>0x40</td>
<td>Transmission was completed.</td>
</tr>
<tr>
<td>GET_BUS4RX</td>
<td>0x50</td>
<td>The I2C bus was secured for reception.</td>
</tr>
<tr>
<td>RX_MODE</td>
<td>0x60</td>
<td>Reception mode</td>
</tr>
<tr>
<td>RX_END</td>
<td>0x70</td>
<td>Reception was completed.</td>
</tr>
<tr>
<td>BUFF_OVER</td>
<td>0x81</td>
<td>The number of bytes that were send exceeded the buffer capacity.</td>
</tr>
<tr>
<td>NO_SLAVE</td>
<td>0x82</td>
<td>The relevant slave does not exist.</td>
</tr>
<tr>
<td>NO_ACK</td>
<td>0x83</td>
<td>NACK was replied to the data that was sent.</td>
</tr>
<tr>
<td>NO_DATA</td>
<td>0x84</td>
<td>The number of bytes that were received is 0.</td>
</tr>
<tr>
<td>MINUTE</td>
<td>60000/16</td>
<td>The counter is incremented by 16 milliseconds to obtain 1 minute.</td>
</tr>
<tr>
<td>TX_DELAY1</td>
<td>1</td>
<td>Transmission interval of Wire1 of Pmod connector 1 (1 μs)</td>
</tr>
<tr>
<td>RX_DELAY1</td>
<td>10</td>
<td>Reception interval of Wire1 of Pmod connector 1 (10 μs)</td>
</tr>
<tr>
<td>TX_DELAY2</td>
<td>1</td>
<td>Transmission interval of Wire2 of Pmod connector 2 (1 μs)</td>
</tr>
<tr>
<td>RX_DELAY2</td>
<td>10</td>
<td>Reception interval of Wire2 of Pmod connector 2 (10 μs)</td>
</tr>
</tbody>
</table>
### 5.3 List of Variables

Table 5.2 to Note: This is shown by the name of the internal processing function, not the Arduino API.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value 1</td>
<td>Internal processing function 1</td>
</tr>
<tr>
<td>Value 2</td>
<td>Internal processing function 2</td>
</tr>
<tr>
<td>Value 3</td>
<td>Internal processing function 3</td>
</tr>
<tr>
<td>Value 4</td>
<td>Internal processing function 4</td>
</tr>
<tr>
<td>Value 5</td>
<td>Internal processing function 5</td>
</tr>
<tr>
<td>Value 6</td>
<td>Internal processing function 6</td>
</tr>
</tbody>
</table>

Note: This is shown by the name of the internal processing function, not the Arduino API.
Table 5.2 Global variables (1/3)

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable Name</th>
<th>Description</th>
<th>Function used</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int</td>
<td>old_time</td>
<td>Time elapsed since the previous startup (in milliseconds)</td>
<td>loop()</td>
<td></td>
</tr>
<tr>
<td>unsigned char</td>
<td>hdc1080_buff[4]</td>
<td>Buffer for the data read from the sensor</td>
<td>loop()</td>
<td></td>
</tr>
<tr>
<td>unsigned int</td>
<td>humid</td>
<td>Humidity data</td>
<td>loop()</td>
<td></td>
</tr>
<tr>
<td>unsigned int</td>
<td>temp</td>
<td>Temperature data in units of 0.1 °C</td>
<td>loop()</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>disp_line1[40]</td>
<td>Data displayed on the LCD indicator (line 1)</td>
<td>loop()</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>disp_line2[40]</td>
<td>Data displayed on the LCD indicator (line 2)</td>
<td>loop()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>count16ms</td>
<td>The counter is incremented by 16 milliseconds to obtain 1 minute.</td>
<td>loop()</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>sw_work</td>
<td>Variable for checking the switch status every 16 milliseconds</td>
<td>loop()</td>
<td></td>
</tr>
<tr>
<td>unsigned char</td>
<td>g_lcd_command[2]</td>
<td>Variable for setting a command on the LCD indicator</td>
<td>set_command()</td>
<td></td>
</tr>
<tr>
<td>unsigned char</td>
<td>g_lcd_data[2]</td>
<td>Variable for setting data on the LCD indicator</td>
<td>set_dat()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>gp_tx_set</td>
<td>Pointer for writing data to the transmission buffer (maximum: 255)</td>
<td>Wire_begin(), Wire_beginTransmission(), Wire_write()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>gp_tx_get</td>
<td>Pointer for reading data from the transmission buffer</td>
<td>Wire_begin(), Wire_beginTransmission(), r_IICA0_interrupt()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_tx_buff[256]</td>
<td>Transmission buffer</td>
<td>Wire_begin(), r_IICA0_interrupt()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>gp_rx_set</td>
<td>Pointer for writing data to the transmission buffer (maximum: 255)</td>
<td>Wire_begin(), Wire_requestFrom(), r_IICA0_interrupt()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>gp_rx_get</td>
<td>Pointer for reading data from the reception buffer</td>
<td>Wire_begin(), Wire_requestFrom(), Wire_read()</td>
<td></td>
</tr>
<tr>
<td>uint16_t</td>
<td>g_rx_buff[256]</td>
<td>Reception buffer</td>
<td>r_IICA0_interrupt(), Wire_read()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>sladdr8</td>
<td>8-bit slave address</td>
<td>Wire_requestFrom(), r_IICA0_interrupt()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_stop_flag</td>
<td>Flag indicating whether to generate stop conditions at termination 0: Performs nothing, 1: Generates stop conditions at termination.</td>
<td>Wire_endTransmission(), Wire_requestFrom(), r_IICA0_interrupt(), r_operation_end()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_status</td>
<td>I2C0 status flag</td>
<td>r_IICA0_interrupt(), Wire_beginTransmission(), Wire_endTransmission(), Wire_requestFrom(), Wire_requestFromSub(), r_IICA0_interrupt(), r_operation_end()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_erflag</td>
<td>0x00: Success 0x01: Buffer overflow 0x02: No slave exists. 0x03: NACK was replied to the data that was sent. 0x04: Other errors</td>
<td>Wire_endTransmission()</td>
<td></td>
</tr>
</tbody>
</table>

Note: This is shown by the name of the internal processing function, not the Arduino API.
### Table 5.3 Global variables (2/3)

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable Name</th>
<th>Description</th>
<th>Function used</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_t</td>
<td>gp_tx1_set</td>
<td>Pointer for writing data to the transmission buffer (max: 255)</td>
<td>Wire1_begin(), Wire1_beginTransmission(), Wire1_write()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>gp_tx1_get</td>
<td>Pointer for reading data from the transmission buffer</td>
<td>Wire1_begin(), Wire1_beginTransmission(), r_IIC00_interrupt()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_tx1_buff[256]</td>
<td>Transmission buffer</td>
<td>Wire1_write(), r_IIC00_interrupt()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>gp_rx1_set</td>
<td>Pointer for writing data to the reception buffer (max: 255)</td>
<td>Wire1_begin(), Wire1_requestFrom(), r_IIC00_interrupt()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>gp_rx1_get</td>
<td>Pointer for reading data from the reception buffer</td>
<td>Wire1_begin(), Wire1_requestFrom(), Wire1_read()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_rx1_buff[256]</td>
<td>Reception buffer</td>
<td>r_IIC00_interrupt(), Wire1_read()</td>
<td></td>
</tr>
<tr>
<td>uint16_t</td>
<td>g_rx1_num</td>
<td>Number of bytes received</td>
<td>Wire1_requestFrom(), r_IIC00_interrupt()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>sladdr8_1</td>
<td>8-bit slave address</td>
<td>Wire1_beginTransmission(), Wire1_requestFromSub(), Wire1_requestFromSubb()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_stop_flag_1</td>
<td>Flag indicating whether to generate stop conditions at termination</td>
<td>Wire1_endTransmission(), Wire1_requestFrom(), r_IIC00_interrupt(), r_operation_end_1()</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_status_1</td>
<td>IIC00 status flag</td>
<td>Wire1_beginTransmission(), Wire1_endTransmission(), Wire1_requestFrom(), Wire1_requestFromSub(), r_IIC00_interrupt(), r_operation_end_1()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x00: BUS FREE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x8F: BUS Error</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x10: Get bus</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x20: Get bus to transmit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x30: Transmit operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x40: Transmit end</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x50: Get bus to receive</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x60: Receive operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x70: Receive end</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x81: Data size over buffer size</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x82: NACK for slave address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x83: No ACK for data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_erflag_1</td>
<td>0x00: Success</td>
<td>Wire1_endTransmission()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x01: Buffer overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x02: No slave exists.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x03: NACK was replied to the data that was sent.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x04: Other errors</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: This is shown by the name of the internal processing function, not the Arduino API.
### Table 5.4 Global variables (3/3)

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable Name</th>
<th>Description</th>
<th>Function used Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_t</td>
<td>gp_tx2_set</td>
<td>Pointer for writing data to the transmission buffer (max: 255)</td>
<td>Wire2_begin(), Wire2_beginTransmission(), Wire2_write()</td>
</tr>
<tr>
<td>uint8_t</td>
<td>gp_tx2_get</td>
<td>Pointer for reading data from the transmission buffer</td>
<td>Wire2_begin(), Wire2_beginTransmission(), r_IIC20_interrupt()</td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_tx2_buff[256]</td>
<td>Transmission buffer</td>
<td>Wire2_write(), r_IIC20_interrupt()</td>
</tr>
<tr>
<td>uint8_t</td>
<td>gp_rx2_set</td>
<td>Pointer for writing data to the reception buffer (max: 255)</td>
<td>Wire2_begin(), Wire2_requestFrom(), r_IIC20_interrupt()</td>
</tr>
<tr>
<td>uint8_t</td>
<td>gp_rx2_get</td>
<td>Pointer for reading data from the reception buffer</td>
<td>Wire2_begin(), Wire2_requestFrom(), Wire2_read()</td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_rx2_buff[256]</td>
<td>Reception buffer</td>
<td>r_IIC20_interrupt(), Wire2_read()</td>
</tr>
<tr>
<td>uint16_t</td>
<td>g_rx2_num</td>
<td>Number of bytes received</td>
<td>Wire2_requestFrom(), r_IIC20_interrupt()</td>
</tr>
<tr>
<td>uint8_t</td>
<td>sladdr8_2</td>
<td>8-bit slave address</td>
<td>Wire2_beginTransmission(), Wire2_requestFromSub(), Wire2_requestFromb()</td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_stop_flag_2</td>
<td>Flag indicating whether to generate stop conditions at termination</td>
<td>Wire2_endTransmission(), Wire2_requestFrom(), r_IIC20_interrupt(), r_operation_end_2()</td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_status_2</td>
<td>IIC00 status flag</td>
<td>Wire2_beginTransmission(), Wire2_endTransmission(), Wire2_requestFromb(), Wire2_requestFromSub(), r_IIC20_interrupt(), r_operation_end_2()</td>
</tr>
<tr>
<td>uint8_t</td>
<td>g_erflag_2</td>
<td>0x00: Success 0x01: Buffer overflow 0x02: No slave exists. 0x03: NACK was replied to the data that was sent. 0x04: Other errors</td>
<td>Wire2_endTransmission()</td>
</tr>
</tbody>
</table>

Note: This is shown by the name of the internal processing function, not the Arduino API.
### 5.4 List of Functions

Table 5.5 to Table 5.6 shows a list of functions.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>Main processing (sketch)</td>
</tr>
<tr>
<td>setup</td>
<td>Initialization function (sketch)</td>
</tr>
<tr>
<td>pinMode</td>
<td>Specifies the operation mode (input mode, output mode, or input mode with an internal pull-up resistor enabled) for the pin.</td>
</tr>
<tr>
<td>digitalWrite</td>
<td>Outputs data to the pin.</td>
</tr>
<tr>
<td>digitalRead</td>
<td>Reads the pin state.</td>
</tr>
<tr>
<td>micros</td>
<td>Returns the time in microseconds from the start of program execution to the present time.</td>
</tr>
<tr>
<td>millis</td>
<td>Returns the time in milliseconds from the start of program execution to the present time.</td>
</tr>
<tr>
<td>delay</td>
<td>Stops the program for the time specified in milliseconds.</td>
</tr>
<tr>
<td>delayMicroseconds</td>
<td>Stops the program for the time specified in microseconds.</td>
</tr>
<tr>
<td>Wire.begin</td>
<td>Initializes the I2C library and connects it as the master.</td>
</tr>
<tr>
<td>Wire.requestFrom</td>
<td>Starts reading data from the specified slave. The read operation is processed as an interrupt.</td>
</tr>
<tr>
<td>Wire_requestFromS</td>
<td>Starts reading data from the specified slave. The read operation is processed as an interrupt. It is possible to specify that stop conditions are generated when reception is completed. This is an internal processing function for Wire.requestFrom.</td>
</tr>
<tr>
<td>Wire_requestFromSub</td>
<td>This is an internal processing function for Wire.requestFrom.</td>
</tr>
<tr>
<td>Wire.available</td>
<td>Returns the number of bytes that can be read by using Wire.read from the reception buffer.</td>
</tr>
<tr>
<td>Wire.read</td>
<td>Reads data from the reception buffer.</td>
</tr>
<tr>
<td>Wire.beginTransmission</td>
<td>Prepares for sending data to the specified slave.</td>
</tr>
<tr>
<td>Wire.write</td>
<td>Writes the send data to the transmission buffer.</td>
</tr>
<tr>
<td>Wire_writec</td>
<td>Adds one-character data to the transmission buffer. This is an internal processing function for Wire.write.</td>
</tr>
<tr>
<td>Wire_writeb</td>
<td>Adds a data block to the transmission buffer. This is an internal processing function for Wire.write.</td>
</tr>
<tr>
<td>Wire.endTransmission</td>
<td>Actually sends the send data that is set in the buffer via the I2C bus. It is possible to specify that stop conditions are generated when transmission is completed.</td>
</tr>
<tr>
<td>init_LCD</td>
<td>Initializes the LCD indicator.</td>
</tr>
<tr>
<td>print_LCD</td>
<td>Displays 1-screen data (16 characters x 2 lines) for the LCD indicator.</td>
</tr>
<tr>
<td>move_cursor</td>
<td>Specifies the cursor position at which to set the data to be displayed on the LCD indicator.</td>
</tr>
<tr>
<td>set_2digit</td>
<td>Displays 1-byte data with 2 digits.</td>
</tr>
<tr>
<td>set_1digit</td>
<td>Displays the lower-bit data with one digit.</td>
</tr>
<tr>
<td>set_command</td>
<td>Sends a command to the LCD indicator.</td>
</tr>
<tr>
<td>set_data</td>
<td>Sends the data to be displayed to the LCD indicator.</td>
</tr>
<tr>
<td>Function Name</td>
<td>Overview</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Wire1.begin</td>
<td>Initializes the I2C library of Pmod connector 1 and connects it as the master.</td>
</tr>
<tr>
<td>Wire1.requestFrom</td>
<td>Starts reading data from the specified slave. The read operation is processed as an interrupt.</td>
</tr>
<tr>
<td>Wire1_requestFromS</td>
<td>Starts reading data from the specified slave. The read operation is processed as an interrupt. It is possible to specify that stop conditions are generated when reception is completed. This is an internal processing function for Wire1.requestFrom.</td>
</tr>
<tr>
<td>Wire1_requestFromSub</td>
<td>This is an internal processing function for Wire1.requestFrom.</td>
</tr>
<tr>
<td>Wire1.available</td>
<td>Returns the number of bytes that can be read by using Wire1.read from the reception buffer.</td>
</tr>
<tr>
<td>Wire1.read</td>
<td>Reads data from the reception buffer.</td>
</tr>
<tr>
<td>Wire1.beginTransmission</td>
<td>Prepares for sending data to the specified slave.</td>
</tr>
<tr>
<td>Wire1.write</td>
<td>Writes the send data to the transmission buffer.</td>
</tr>
<tr>
<td>Wire1_writec</td>
<td>Adds one-character data to the transmission buffer. This is an internal processing function for Wire1.write.</td>
</tr>
<tr>
<td>Wire1_writeb</td>
<td>Adds a data block to the transmission buffer. This is an internal processing function for Wire1.write.</td>
</tr>
<tr>
<td>Wire1.endTransmission</td>
<td>Actually sends the send data that is set in the buffer via the I2C bus. It is possible to specify that stop conditions are generated when transmission is completed.</td>
</tr>
<tr>
<td>Wire2.begin</td>
<td>Initializes the I2C library of Pmod connector 2 and connects it as the master.</td>
</tr>
<tr>
<td>Wire2.requestFrom</td>
<td>Starts reading data from the specified slave. The read operation is processed as an interrupt.</td>
</tr>
<tr>
<td>Wire2_requestFromS</td>
<td>Starts reading data from the specified slave. The read operation is processed as an interrupt. It is possible to specify that stop conditions are generated when reception is completed. This is an internal processing function for Wire2.requestFrom.</td>
</tr>
<tr>
<td>Wire2_requestFromSub</td>
<td>This is an internal processing function for Wire2.requestFrom.</td>
</tr>
<tr>
<td>Wire2.available</td>
<td>Returns the number of bytes that can be read by using Wire2.read from the reception buffer.</td>
</tr>
<tr>
<td>Wire2.read</td>
<td>Reads data from the reception buffer.</td>
</tr>
<tr>
<td>Wire2.beginTransmission</td>
<td>Prepares for sending data to the specified slave.</td>
</tr>
<tr>
<td>Wire2.write</td>
<td>Writes the send data to the transmission buffer.</td>
</tr>
<tr>
<td>Wire2_writec</td>
<td>Adds one-character data to the transmission buffer. This is an internal processing function for Wire2.write.</td>
</tr>
<tr>
<td>Wire2_writeb</td>
<td>Adds a data block to the transmission buffer. This is an internal processing function for Wire2.write.</td>
</tr>
<tr>
<td>Wire2.endTransmission</td>
<td>Actually sends the send data that is set in the buffer via the I2C bus. It is possible to specify that stop conditions are generated when transmission is completed.</td>
</tr>
</tbody>
</table>
### 5.5 Specification of Functions

The function specifications of the sample code are shown below.

<table>
<thead>
<tr>
<th>Function name</th>
<th>loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>Main function</td>
</tr>
<tr>
<td>Header</td>
<td>AR_LIB_PORT.h, AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, AR_SKETCH.h, r_cg_userdefine.h, LCD_LIB.h</td>
</tr>
<tr>
<td>Declaration</td>
<td>void loop(void);</td>
</tr>
<tr>
<td>Description</td>
<td>After startup, this function checks the status of the switch at 16-ms intervals. If the switch is pressed or 1 minute elapses, the function starts the sensor (HDC1080). After the sensor starts, when about 16 milliseconds elapse, the function reads the measurement results of the sensor. The function then calculates the temperature and humidity from the measurement results and displays the calculation results on the LCD indicator.</td>
</tr>
<tr>
<td>Argument</td>
<td>None</td>
</tr>
<tr>
<td>Return value</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function name</th>
<th>setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>Initialization function</td>
</tr>
<tr>
<td>Header</td>
<td>AR_LIB_PORT.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h</td>
</tr>
<tr>
<td>Declaration</td>
<td>void setup(void);</td>
</tr>
<tr>
<td>Description</td>
<td>This function sets up the pins, I2CA0, and LCD indicator used by the program (sketch).</td>
</tr>
<tr>
<td>Argument</td>
<td>None</td>
</tr>
<tr>
<td>Return value</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function name</th>
<th>pinMode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>Function to set the pin function</td>
</tr>
<tr>
<td>Header</td>
<td>AR_LIB_PORT.h, r_cg_macrodriver.h, r_cg_userdefine.h</td>
</tr>
<tr>
<td>Declaration</td>
<td>void pinMode(uint8_t pin, uint8_t mode);</td>
</tr>
<tr>
<td>Description</td>
<td>The pin indicated by the first argument is set to the mode indicated by the second argument.</td>
</tr>
</tbody>
</table>
| Argument      | uint8_t pin : Number of the pin to be specified  
uint8_t mode : Specifies the pin mode with OUTPUT/INPUT/INPUT_PULLUP |
| Return value  | None |

<table>
<thead>
<tr>
<th>Function Name</th>
<th>digitalWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>Function to output digital data to a pin</td>
</tr>
<tr>
<td>Header</td>
<td>AR_LIB_PORT.h, r_cg_macrodriver.h, r_cg_userdefine.h</td>
</tr>
<tr>
<td>Declaration</td>
<td>void digitalWrite(uint8_t pin, uint8_t t value);</td>
</tr>
<tr>
<td>Description</td>
<td>The data indicated by the second argument is output to the pin indicated by the first argument.</td>
</tr>
</tbody>
</table>
| Argument      | uint8_t t pin : Number of the pin for data output  
uint8_t t value : Data to output (HIGH/LOW) |
| Return value  | None |
### [Function Name] pinMode

**Overview**
Function to make a specific pin as an input or output

**Header**
AR_LIB_PORT.h, r_cg_macrodriver.h, r_cg_userdefine.h

**Declaration**
void pinMode(uint8_t pin, int mode);

**Description**
Sets the mode of the pin specified by the argument.

**Argument**
- uint8_t pin:
  Number of the pin to be configured
- int mode:
  Mode to be set (INPUT, OUTPUT, INPUT_PULLUP)

**Return value**
None

---

### [Function Name] digitalRead

**Overview**
Function to read out digital data from a pin

**Header**
AR_LIB_PORT.h, r_cg_macrodriver.h, r_cg_userdefine.h

**Declaration**
uint8_t digitalRead(uint8_t pin);

**Description**
The state of the pin specified by the argument is read out.

**Argument**
- uint8_t pin:
  Number of the pin to be read out

**Return value**
uint8_t:
Data that was read out (HIGH/LOW)

---

### [Function Name] micros

**Overview**
Function to obtain the elapsed time in microsecond units

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, r_cg_userdefine.h

**Declaration**
uint32_t micros(void);

**Description**
Returns the time elapsed from startup, in microsecond units.

**Argument**
None

**Return value**
uint32_t:
Elapsed time in microsecond units

---

### [Function Name] millis

**Overview**
Function to obtain the elapsed time in millisecond units

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, r_cg_userdefine.h

**Declaration**
uint32_t millis(void);

**Description**
Returns the time elapsed from startup, in millisecond units.

**Argument**
None

**Return value**
uint32_t:
Elapsed time in millisecond units

---

### [Function Name] delay

**Overview**
A function that waits for a certain length of time in milliseconds

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, r_cg_userdefine.h

**Declaration**
uint32_t delay(uint32_t time);

**Description**
This function waits for the length of time specified for an argument in milliseconds.

**Argument**
- uint32_t time:
  Wait time (in milliseconds)

**Return value**
None

---

### [Function Name] delayMicroseconds

**Overview**
A function that waits for a certain length of time in microseconds

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, r_cg_userdefine.h

**Declaration**
void delayMicroseconds(uint32_t time);

**Description**
This function waits for the length of time specified for an argument in microseconds.

**Argument**
- uint32_t time:
  Wait time (in microseconds)

**Return value**
None
### Wire.begin

**Overview**
Function that prepares for using the I2C bus

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h

**Declaration**
void Wire.begin(void);

**Description**
This function initializes I2C0 as a preparation for using the I2C bus.

**Argument**
None

**Return value**
None

### Wire.requestFrom

**Overview**
Function that prepares for receiving data from the slave

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h

**Declaration**
void Wire.requestFrom(uint8_t saddr7, uint16_t bytes, uint8_t stop);

**Description**
This function issues the start condition and sends the slave address so that data can be received under the conditions specified by using arguments. The subsequent processing is performed as forms of interrupts. When this function ends, it performs the processing specified by the third argument.

**Argument**
- uint8_t saddr7: 7-bit slave address
- uint16_t bytes: Number of bytes to be received
- uint8_t stop: Processing to be performed when the function ends (If this argument is omitted, the function releases the bus.)
  0: Issues the restart condition. (The bus is held.)
  1: Issues the stop condition. (The bus is released.)

**Return value**
uint8_t
0x00: Normal
0x01: Buffer overflow
0x04: Other errors

**Remarks**
g_status: Communication status
- If the value that is set is 0x50, startup is successful. Afterward, the value changes to 0x60 (now receiving), and then to 0x70 (reception completed).
- The other values are as follows: 0x81: buffer error, 0x84: no data received, 0x8F: startup failed
- Processing that starts communication with the I2C bus must not be performed during execution of this function.
### Wire_requestFromS

**Overview**  
Function that prepares for receiving data from the slave

**Header**  
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h

**Declaration**  
void Wire_requestFromS(uint8_t saddr7, uint16_t bytes);

**Description**  
This function issues the start condition and sends the slave address so that data can be received under the conditions specified by using arguments. The subsequent processing is performed as forms of interrupts. When this function ends, it issues the stop condition and releases the bus.  
(This function is used for the internal processing of Wire.requestFrom.)

**Argument**
- uint8_t saddr7  
  7-bit slave address
- uint16_t bytes  
  Number of bytes to be received

**Return value**
- uint8_t  
  0x00: Normal  
  0x01: Buffer overflow  
  0x04: Other errors

**Remarks**
- g_status: Communication status
  - If the value that is set is 0x50, startup is successful. Afterward, the value changes to 0x60 (now receiving), and then to 0x70 (reception completed).
  - The other values are as follows: 0x81: buffer error, 0x84: no data received, 0x8F: startup failed
  - Processing that starts communication with the I2C bus must not be performed during execution of this function.

### Wire_requestFromSub

**Overview**  
Internal function that prepares for receiving data from the slave

**Header**  
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h

**Declaration**  
void Wire_requestFromSub(uint8_t saddr7, uint16_t bytes, uint8_t stop);

**Description**  
This function issues the start condition and sends the slave address so that data can be received under the conditions specified by using arguments. The subsequent processing is performed as forms of interrupts. When this function ends, it performs the processing specified by the third argument.  
(This function is an internal function for Wire.requestFrom.)

**Argument**
- uint8_t saddr7  
  7-bit slave address
- uint16_t bytes  
  Number of bytes to be received
- uint8_t stop  
  Processing to be performed when the function ends (If this argument is omitted, the function releases the bus.)
  - 0: Issues the restart condition. (The bus is held.)
  - 1: Issues the stop condition. (The bus is released.)

**Return value**  
None

**Remarks**
- g_status: Communication status
  - If the value that is set is 0x50, startup is successful. Afterward, the value changes to 0x60 (now receiving), and then to 0x70 (reception completed).
  - The other values are as follows: 0x81: buffer error, 0x84: no data received, 0x8F: startup failed
- g_erflag: Error flag
  - 0x00: normal, 0x01: buffer overflow, 0x04: other errors
  - Processing that starts communication with the I2C bus must not be performed during execution of this function.
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Wire.available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>Function that returns the number of bytes that can be read</td>
</tr>
<tr>
<td>Header</td>
<td>AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h</td>
</tr>
<tr>
<td>Declaration</td>
<td>uint8_t Wire.available(void);</td>
</tr>
<tr>
<td>Description</td>
<td>This function uses the Wire.requestFrom function to receive data and then returns the number of bytes of the data stored in a buffer.</td>
</tr>
<tr>
<td>Argument</td>
<td>None</td>
</tr>
<tr>
<td>Return value</td>
<td>uint8_t Number of bytes that can be read from the buffer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Wire.read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>Function that reads data from the receive buffer</td>
</tr>
<tr>
<td>Header</td>
<td>AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h</td>
</tr>
<tr>
<td>Declaration</td>
<td>uint8_t Wire.read(void);</td>
</tr>
<tr>
<td>Description</td>
<td>This function reads data from the buffer.</td>
</tr>
<tr>
<td>Argument</td>
<td>None</td>
</tr>
<tr>
<td>Return value</td>
<td>uint8_t Data read from the buffer (or 0x00)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Wire.beginTransmission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>Function that prepares for sending data to the slave</td>
</tr>
<tr>
<td>Header</td>
<td>AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h</td>
</tr>
<tr>
<td>Declaration</td>
<td>void Wire.beginTransmission(uint8_t saddr7);</td>
</tr>
<tr>
<td>Description</td>
<td>This function converts the slave address to an 8-bit address, stores it in the &quot;sladdr8&quot; variable, and then issues the start condition to secure the bus.</td>
</tr>
<tr>
<td>Argument</td>
<td>uint8_t saddr7 7-bit slave address</td>
</tr>
<tr>
<td>Return value</td>
<td>uint8_t 0x00: Normal</td>
</tr>
<tr>
<td>Remarks</td>
<td>g_erflag: Communication status</td>
</tr>
<tr>
<td></td>
<td>If the value that is set is 0x00, startup is successful.</td>
</tr>
<tr>
<td></td>
<td>If the value is 0x04, the function failed to secure the I2C bus.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Wire.write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>Function that sets the send data</td>
</tr>
<tr>
<td>Header</td>
<td>AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h</td>
</tr>
<tr>
<td>Declaration</td>
<td>uint8_t Wire.write( uint8_t data); uint8_t Wire.write(uint8_t *buff, uint8_t _bytes);</td>
</tr>
<tr>
<td>Description</td>
<td>This function stores one character specified for argument 1 or the data block specified for argument 2 in the send buffer.</td>
</tr>
<tr>
<td>Argument 1</td>
<td>uint8_t data Data to be sent</td>
</tr>
<tr>
<td>Argument 2</td>
<td>uint8_t *buff Data block to be sent</td>
</tr>
<tr>
<td>Return value</td>
<td>uint8_t Number of bytes stored in the buffer</td>
</tr>
<tr>
<td>Remarks</td>
<td>g_erflag: Communication status</td>
</tr>
<tr>
<td></td>
<td>If the value of &quot;g_erflag&quot; is 0x01, the send buffer has overflowed. If the value is 0x04, the function failed to secure the I2C bus.</td>
</tr>
</tbody>
</table>
### Wire_writec

#### Overview
Function that sets the send data

#### Header
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h

#### Declaration
uint8_t Wire_writec(uint8_t data);

#### Description
This function stores one character specified for argument 1. (This function is an internal function that processes 1 character in the Wire.write function.)

#### Argument
- **uint8_t data**
  - Data to be sent

#### Return value
- **uint8_t**
  - Number of bytes stored in the buffer

#### Remarks
If the value of "g_erflag" is 0x01, the send buffer has overflowed. If the value is 0x04, the function failed to secure the I²C bus.

### Wire_writeb

#### Overview
Function that sets the send data

#### Header
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, and r_cg_userdefine.h

#### Declaration
uint8_t Wire_writeb(uint8_t *buff, uint8_t bytes);

#### Description
This function stores the data of the block specified for an argument in the send buffer. (This function is an internal function that processes a block in the Wire.write function.)

#### Argument
- **uint8_t *buff**
  - Address of the data block to be sent
- **uint8_t bytes**
  - Number of bytes to be sent

#### Return value
- **uint8_t**
  - Number of bytes stored in the buffer

#### Remarks
If the value of "g_erflag" is 0x01, the send buffer has overflowed. If the value is 0x04, the function failed to secure the I²C bus.

### Wire.endTransmission

#### Overview
Function that sends data to the slave

#### Header
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, and r_cg_userdefine.h

#### Declaration
void Wire.endTransmission(uint8_t STOP);

#### Description
This function sends data from the send buffer to the slave.

#### Argument
- **uint8_t STOP**
  - Processing performed when sending is completed:
  0: Issues the restart condition to secure the bus.
  1: Releases the bus.

#### Return value
- **uint8_t**
  - Result of sending:
  0: Success
  1: The number of bytes exceeded the buffer size.
  2: NACK was replied to the slave address.
  3: NACK was replied to the send data.
  4: Other errors

---
**init_LCD**

**Overview**
Function that initializes the LCD indicator

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, and r_cg_userdefine.h

**Declaration**
uint8_t init_LCD(void);

**Description**
This function sets the LCD indicator in 16 (characters) x 2 (lines) mode and clears the display.

**Argument**
None

**Return value**
uint8_t Communication result:
0: Success
2: The LCD indicator does not respond.

**print_LCD**

**Overview**
Function that sets 1-screen data (16 characters x 2 lines) for the LCD indicator

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, and r_cg_userdefine.h

**Declaration**
void print_LCD(uint8_t *point, uint8_t *point2);

**Description**
This function displays 32 characters from the address passed by an argument on two lines of the LCD indicator.

**Argument**
uint8_t *point Specifies the start address of disp_line1.
uint8_t *point2 Specifies the start address of disp_line2.

**Return value**
None

**move_cursor**

**Overview**
Function that sets the cursor position on the LCD indicator

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, and r_cg_userdefine.h

**Declaration**
void move_cursor(uint8_t col, uint8_t row);

**Description**
This function moves the cursor to the position passed by an argument.

**Argument**
uint8_t col Specifies the column position on the line.
uint8_t row Specifies the line position.

**Return value**
None

**Remarks**
After this function is run, the next write operation must not be performed before 60 microseconds elapse.

**set_2digit**

**Overview**
Function that displays a numeric value with 2-digit ASCII codes on the LCD indicator

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, and r_cg_userdefine.h

**Declaration**
void set_2digit(uint8_t datacode);

**Description**
This function receives hexadecimal or BCD data via arguments, converts the data to 2-digit ASCII codes, and then sends the ASCII codes as display data to the LCD indicator.

**Argument**
uint8_t datacode 8-bit data code (hexadecimal or BCD data) to be sent to the LCD indicator

**Return value**
None
**set_1digit**

**Overview**
Function that displays a numeric value with a 1-digit ASCII code on the LCD indicator

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, and r_cg_userdefine.h

**Declaration**
void set_1digit(uint8_t datacode);

**Description**
This function receives data via an argument, converts the last 4 bits of the data to an ASCII code, and then sends the conversion results as display data to the LCD indicator.

**Argument**
uint8_t datacode

**Return value**
None

**set_command**

**Overview**
Function that sends a command to the LCD indicator

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h

**Declaration**
uint8_t set_command(uint8_t lcd_command);

**Description**
This function receives a command code via an argument and sends it as a command to the LCD indicator.

**Argument**
uint8_t lcd_command

**Return value**
uint8_t
Communication result:
0: Success
2: The LCD indicator does not respond.

**Remarks**
After this function is run, the next write operation must not be performed before 60 microseconds elapse.

**set_data**

**Overview**
Function that sends display data to the LCD indicator

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE.h, r_cg_userdefine.h

**Declaration**
uint8_t set_data(uint8_t datacode);

**Description**
This function receives a data code via an argument and sends it as data to the LCD indicator.

**Argument**
uint8_t datacode

**Return value**
uint8_t
Communication result:
0: Success
2: The LCD indicator does not respond.

**Remarks**
After this function is run, the next write operation must not be performed before 60 microseconds elapse.
### Wire1.begin

**Overview**
Function that prepares for using the I2C bus of PMOD1 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE1.h, r_cg_userdefine.h

**Declaration**
void Wire1.begin(void);

**Description**
This function initializes IICA0 as a preparation for using the I2C bus.

**Argument**
None

**Return value**
None

---

### Wire1.requestFrom

**Overview**
Function that prepares for receiving data from the slave using the I2C bus of PMOD1 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE1.h, r_cg_userdefine.h

**Declaration**
void Wire1.requestFrom(uint8_t saddr7, uint16_t bytes, uint8_t stop);

**Description**
This function issues the start condition and sends the slave address so that data can be received under the conditions specified by using arguments. The subsequent processing is performed as forms of interrupts. When this function ends, it performs the processing specified by the third argument.

**Argument**
- `uint8_t saddr7` 7-bit slave address
- `uint16_t bytes` Number of bytes to be received
- `uint8_t stop` Processing to be performed when the function ends (If this argument is omitted, the function releases the bus.)
  - 0: Issues the restart condition. (The bus is held.)
  - 1: Issues the stop condition. (The bus is released.)

**Return value**
`uint8_t`
- 0x00: Normal
- 0x01: Buffer overflow
- 0x04: Other errors

**Remarks**
g_status: Communication status
If the value that is set is 0x50, startup is successful. Afterward, the value changes to 0x60 (now receiving), and then to 0x70 (reception completed).

The other values are as follows:
- 0x81: buffer error
- 0x84: no data received
- 0xF: startup failed

Processing that starts communication with the I2C bus must not be performed during execution of this function.
### Wire1_requestFromS

**Overview**
Function that prepares for receiving data from the slave using the I2C bus of PMOD1 connector.

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE1.h, r_cg_userdefine.h

**Declaration**
void Wire1_requestFromS(uint8_t saddr7, uint16_t bytes);

**Description**
This function issues the start condition and sends the slave address so that data can be received under the conditions specified by using arguments. The subsequent processing is performed as forms of interrupts. When this function ends, it issues the stop condition and releases the bus.

(This function is used for the internal processing of Wire1.requestFrom.)

**Argument**
- uint8_t saddr7: 7-bit slave address
- uint16_t bytes: Number of bytes to be received

**Return value**
- uint8_t: 0x00: Normal
  0x01: Buffer overflow
  0x04: Other errors

**Remarks**
g_status: Communication status
If the value that is set is 0x50, startup is successful. Afterward, the value changes to 0x60 (now receiving), and then to 0x70 (reception completed).

The other values are as follows: 0x81: buffer error, 0x84: no data received, 0x8F: startup failed
Processing that starts communication with the I2C bus must not be performed during execution of this function.

### Wire1_requestFromSub

**Overview**
Internal function that prepares for receiving data from the slave using the I2C bus of PMOD1 connector.

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE1.h, r_cg_userdefine.h

**Declaration**
void Wire1_requestFromSub(uint8_t saddr7, uint16_t bytes, uint8_t stop);

**Description**
This function issues the start condition and sends the slave address so that data can be received under the conditions specified by using arguments. The subsequent processing is performed as forms of interrupts. When this function ends, it performs the processing specified by the third argument.

(This function is an internal function for Wire1.requestFrom.)

**Argument**
- uint8_t saddr7: 7-bit slave address
- uint16_t bytes: Number of bytes to be received
- uint8_t stop: Processing to be performed when the function ends (If this argument is omitted, the function releases the bus.)
  - 0: Issues the restart condition. (The bus is held.)
  - 1: Issues the stop condition. (The bus is released.)

**Return value**
None

**Remarks**
g_status: Communication status
If the value that is set is 0x50, startup is successful. Afterward, the value changes to 0x60 (now receiving), and then to 0x70 (reception completed).

The other values are as follows: 0x81: buffer error, 0x84: no data received, 0x8F: startup failed
g_erflag: Error flag
0x00: normal, 0x01: buffer overflow, 0x04: other errors
Processing that starts communication with the I2C bus must not be performed during execution of this function.
### Wire1.available

**Overview**
Function that returns the number of bytes that can be read from the receive buffer of Wire1.

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE1.h, r_cg_userdefine.h

**Declaration**
uint8_t Wire1.available(void);

**Description**
This function uses the Wire1.requestFrom function to receive data and then returns the number of bytes of the data stored in a buffer.

**Argument**
None

**Return value**
uint8_t Number of bytes that can be read from the buffer

---

### Wire1.read

**Overview**
Function that reads data from the receive buffer

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE1.h, r_cg_userdefine.h

**Declaration**
uint8_t Wire1.read(void);

**Description**
This function reads data from the buffer.

**Argument**
None

**Return value**
uint8_t Data read from the buffer (or 0x00)

---

### Wire1.beginTransmission

**Overview**
Function that prepares for sending data to the slave on the I2C bus of the Pmod1 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE1.h, r_cg_userdefine.h

**Declaration**
void Wire1.beginTransmission(uint8_t saddr7);

**Description**
This function converts the slave address to an 8-bit address, stores it in the "sladdr8" variable, and then issues the start condition to secure the bus.

**Argument**
uint8_t saddr7 7-bit slave address

**Return value**
uint8_t 0x00: Normal
0x04: Other errors

**Remarks**
g_erflag: Communication status
- If the value that is set is 0x00, startup is successful.
- If the value is 0x04, the function failed to secure the I2C bus.

---

### Wire1.write

**Overview**
Function that sets the send data to the slave on the I2C bus of the Pmod1 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE1.h, r_cg_userdefine.h

**Declaration**
uint8_t Wire1.write( uint8_t data); uint8_t Wire1.write(uint8_t *buff, uint8_t bytes);

**Description**
This function stores one character specified for argument 1 or the data block specified for argument 2 in the send buffer.

**Argument 1**
uint8_t data Data to be sent

**Argument 2**
uint8_t *buff Data block to be sent
uint8_t byte Number of bytes to be sent

**Return value**
uint8_t Number of bytes stored in the buffer

**Remarks**
If the value of "g_erflag" is 0x01, the send buffer has overflowed. If the value is 0x04, the function failed to secure the I2C bus.
### Wire1_writec

**Overview**
Function that sets the send data to the slave on the I2C bus of the Pmod1 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE1.h, r_cg_userdefine.h

**Declaration**
uint8_t Wire1_writec(uint8_t data);

**Description**
This function stores one character specified for argument 1.
(This function is an internal function that processes 1 character in the Wire1.write function.)

**Argument**
- **uint8_t data**
  - Data to be sent

**Return value**
- **uint8_t**
  - Number of bytes stored in the buffer

**Remarks**
If the value of "g_erflag" is 0x01, the send buffer has overflowed. If the value is 0x04, the function failed to secure the I²C bus.

### Wire1_writeb

**Overview**
Function that sets the send data to the slave on the I2C bus of the Pmod1 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE1.h, r_cg_userdefine.h

**Declaration**
uint8_t Wire1_writeb(uint8_t *buff, uint8_t bytes);

**Description**
This function stores the data of the block specified for an argument in the send buffer.
(This function is an internal function that processes a block in the Wire1.write function.)

**Argument**
- **uint8_t *buff**
  - Address of the data block to be sent
- **uint8_t bytes**
  - Number of bytes to be sent

**Return value**
- **uint8_t**
  - Number of bytes stored in the buffer

**Remarks**
If the value of "g_erflag" is 0x01, the send buffer has overflowed. If the value is 0x04, the function failed to secure the I²C bus.

### Wire1.endTransmission

**Overview**
Function that sends data to the slave on the I2C bus of the Pmod1 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE1.h, r_cg_userdefine.h

**Declaration**
void Wire1_endTransmission(uint8_t STOP);

**Description**
This function sends data from the send buffer to the slave.

**Argument**
- **uint8_t STOP**
  - Processing performed when sending is completed:
    0: Issues the restart condition to secure the bus.
    1: Releases the bus.

**Return value**
- **uint8_t**
  - Result of sending:
    0: Success
    1: The number of bytes exceeded the buffer size.
    2: NACK was replied to the slave address.
    3: NACK was replied to the send data.
    4: Other errors
[Function Name] Wire2.begin

Overview  Function that prepares for using the I2C bus of PMOD1 connector
Header   AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE2.h, r_cg_userdefine.h
Declaration  void Wire2.begin(void);
Description  This function initializes I2C0 as a preparation for using the I2C bus.
Argument  None
Return value  None

[Function Name] Wire2.requestFrom

Overview  Function that prepares for receiving data from the slave using the I2C bus of PMOD1 connector
Header   AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE2.h, r_cg_userdefine.h
Declaration  void Wire2.requestFrom(uint8_t saddr7, uint16_t bytes, uint8_t stop);
Description  This function issues the start condition and sends the slave address so that data can be received under the conditions specified by using arguments. The subsequent processing is performed as forms of interrupts. When this function ends, it performs the processing specified by the third argument.
Argument  uint8_t saddr7  7-bit slave address
          uint16_t bytes  Number of bytes to be received
          uint8_t stop   Processing to be performed when the function ends (If this argument is omitted, the function releases the bus.)
          0: Issues the restart condition. (The bus is held.)
          1: Issues the stop condition. (The bus is released.)
Return value  uint8_t  0x00: Normal
              0x01: Buffer overflow
              0x04: Other errors
Remarks  g_status: Communication status
         If the value that is set is 0x50, startup is successful. Afterward, the value changes to 0x60 (now receiving), and then to 0x70 (reception completed).
         The other values are as follows: 0x81: buffer error, 0x84: no data received, 0x8F: startup failed
         Processing that starts communication with the I2C bus must not be performed during execution of this function.
[Function Name] Wire2_requestFromS

**Overview**
Function that prepares for receiving data from the slave using the I2C bus of PMOD1 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE2.h, r_cg_userdefine.h

**Declaration**
void Wire2_requestFromS(uint8_t saddr7, uint16_t bytes);

**Description**
This function issues the start condition and sends the slave address so that data can be received under the conditions specified by using arguments. The subsequent processing is performed as forms of interrupts. When this function ends, it issues the stop condition and releases the bus.

(This function is used for the internal processing of Wire2.requestFrom.)

**Argument**
- uint8_t saddr7: 7-bit slave address
- uint16_t bytes: Number of bytes to be received

**Return value**
- uint8_t: Normal
- 0x01: Buffer overflow
- 0x04: Other errors

**Remarks**
- g_status: Communication status
  - If the value that is set is 0x50, startup is successful. Afterward, the value changes to 0x60 (now receiving), and then to 0x70 (reception completed).
  - The other values are as follows: 0x81: buffer error, 0x84: no data received, 0x89: startup failed

Processing that starts communication with the I2C bus must not be performed during execution of this function.

[Function Name] Wire2_requestFromSub

**Overview**
Internal function that prepares for receiving data from the slave using the I2C bus of PMOD1 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE2.h, r_cg_userdefine.h

**Declaration**
void Wire2_requestFromSub(uint8_t saddr7, uint16_t bytes, uint8_t stop);

**Description**
This function issues the start condition and sends the slave address so that data can be received under the conditions specified by using arguments. The subsequent processing is performed as forms of interrupts. When this function ends, it performs the processing specified by the third argument.

(This function is an internal function for Wire2.requestFrom.)

**Argument**
- uint8_t saddr7: 7-bit slave address
- uint16_t bytes: Number of bytes to be received
- uint8_t stop: Processing to be performed when the function ends (If this argument is omitted, the function releases the bus.)
  - 0: Issues the restart condition. (The bus is held.)
  - 1: Issues the stop condition. (The bus is released.)

**Return value**
None

**Remarks**
- g_status: Communication status
  - If the value that is set is 0x50, startup is successful. Afterward, the value changes to 0x60 (now receiving), and then to 0x70 (reception completed).
  - The other values are as follows: 0x81: buffer error, 0x84: no data received, 0x89: startup failed

g_erflag: Error flag
- 0x00: normal, 0x01: buffer overflow, 0x04: other errors

Processing that starts communication with the I2C bus must not be performed during execution of this function.
**[Function Name]** Wire2.available

**Overview**
Function that returns the number of bytes that can be read from the receive buffer of Wire2.

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE2.h, r_cg_userdefine.h

**Declaration**
uint8_t Wire2.available(void);

**Description**
This function uses the Wire2.requestFrom function to receive data and then returns the number of bytes of the data stored in a buffer.

**Argument**
None

**Return value**
uint8_t Number of bytes that can be read from the buffer

---

**[Function Name]** Wire2.read

**Overview**
Function that reads data from the receive buffer.

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE2.h, r_cg_userdefine.h

**Declaration**
uint8_t Wire2.read(void);

**Description**
This function reads data from the buffer.

**Argument**
None

**Return value**
uint8_t Data read from the buffer (or 0x00)

---

**[Function Name]** Wire2.beginTransmission

**Overview**
Function that prepares for sending data to the slave on the I2C bus of the Pmod2 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE2.h, r_cg_userdefine.h

**Declaration**
void Wire2.beginTransmission(uint8_t saddr7);

**Description**
This function converts the slave address to an 8-bit address, stores it in the "sladdr8" variable, and then issues the start condition to secure the bus.

**Argument**
uint8_t saddr7 7-bit slave address

**Return value**
uint8_t 0x00: Normal 0x04: Other errors

**Remarks**
g_erflag: Communication status
If the value that is set is 0x00, startup is successful.
If the value is 0x04, the function failed to secure the I2C bus.

---

**[Function Name]** Wire2.write

**Overview**
Function that sets the send data to the slave on the I2C bus of the Pmod2 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE2.h, r_cg_userdefine.h

**Declaration**
uint8_t Wire2.write( uint8_t data); uint8_t Wire2.write(uint8_t *buff, uint8_t bytes);

**Description**
This function stores one character specified for argument 1 or the data block specified for argument 2 in the send buffer.

**Argument 1**
uint8_t data Data to be sent

**Argument 2**
uint8_t *buff Data block to be sent uint8_t byte Number of bytes to be sent

**Return value**
uint8_t Number of bytes stored in the buffer

**Remarks**
If the value of "g_erflag" is 0x01, the send buffer has overflowed. If the value is 0x04, the function failed to secure the I2C bus.
**[Function Name]** Wire2_writec

**Overview**
Function that sets the send data to the slave on the I2C bus of the Pmod2 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE2.h, r_cg_userdefine.h

**Declaration**
uint8_t Wire2_writec(uint8_t data);

**Description**
This function stores one character specified for argument 1.
(This function is an internal function that processes 1 character in the Wire2.write function.)

**Argument**
uint8_t data  Data to be sent

**Return value**
uint8_t  Number of bytes stored in the buffer

**Remarks**
If the value of "g_erflag" is 0x01, the send buffer has overflowed. If the value is 0x04, the function failed to secure the I²C bus.

---

**[Function Name]** Wire2_writeb

**Overview**
Function that sets the send data to the slave on the I2C bus of the Pmod2 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE2.h, r_cg_userdefine.h

**Declaration**
uint8_t Wire2_writeb(uint8_t *buff, uint8_t bytes);

**Description**
This function stores the data of the block specified for an argument in the send buffer.
(This function is an internal function that processes a block in the Wire2.write function.)

**Argument**
uint8_t *buff  Address of the data block to be sent
uint8_t bytes  Number of bytes to be sent

**Return value**
uint8_t  Number of bytes stored in the buffer

**Remarks**
If the value of "g_erflag" is 0x01, the send buffer has overflowed. If the value is 0x04, the function failed to secure the I²C bus.

---

**[Function Name]** Wire2.endTransmission

**Overview**
Function that sends data to the slave on the I2C bus of the Pmod2 connector

**Header**
AR_LIB_TIME.h, r_cg_macrodriver.h, AR_LIB_WIRE2.h, r_cg_userdefine.h

**Declaration**
void Wire2_endTransmission(uint8_t STOP);

**Description**
This function sends data from the send buffer to the slave.

**Argument**
uint8_t STOP  Processing performed when sending is completed:
0: Issues the restart condition to secure the bus.
1: Releases the bus.

**Return value**
uint8_t  Result of sending:
0: Success
1: The number of bytes exceeded the buffer size.
2: NACK was replied to the slave address.
3: NACK was replied to the send data.
4: Other errors
5.6 Flowcharts

5.6.1 Initial setting function

Figure 5.1 shows the flowchart of the initial setting.

```
setup

Set the internal switch control pin: pinMode()

swPin is set to INPUT.

Set the I2C bus to the master: Wire.begin()

The I2C bus is set to the master in standard mode.

Set the I2C bus to the master: Wire1.begin()

The I2C bus of the Pmod1 connector is set to 380 kbps.

Initialize the LCD indicator: init_LCD()

The LCD indicator is set to 16 (chars) x 2 (lines) mode.

Set the font for the degree display:

0xDF is assigned to variable disp_line1[14].

Display the initial values on the LCD indicator: print_LCD()

15°C is displayed as the temperature and 50% is displayed as the humidity.

return
```

Figure 5.1 Initial setting function
5.6.2 Main Processing Function

Figure 5.2 to Figure 5.5 show a flowchart of the main processing function.

![Flowchart of main processing function](image)

Figure 5.2 Main Function (1/4)

- **Bits 15 to 4 are read as the elapsed time in milliseconds.**
  
  "millis() & 0x0FFF0" is assigned to variable `time_work`.
  
  If the difference from the previous value is less than 16 ms, the function ends without performing processing.

- **Variable `time_work` is assigned to variable `old_time`**.
  
  "count16ms + 1" is assigned to variable `count16ms`.
  
  Variable `sw_work` shifted to the left by 1 bit.
  "sw_work + swPin" is assigned to variable `sw_work`.

  "sw_work & 0x03" is assigned to variable `sw_data`: Lower 2 bits are extracted.

  0x01 is assigned to variable `m_time`:
  The measurement start flag is set.

  0x01 is assigned to variable `m_time`:
  The measurement start flag is set.

  0x0000 is assigned to variable `count16ms`:
  The counter is cleared.
Transmission start processing is performed for HDC1080.

0x00 is set as the data to be sent to HDC1080.

0x00 is written to the HDC1080 register pointer (measurement start processing of H1080)

0x02 is assigned to variable m_time.

Processing that starts HDC1080 (m_time=1)

Wait for a specific length of time (m_time=2)

0x03 is assigned to variable m_time.

HDC1080 processing that starts reading the conversion results (m_time=3)

Processing to read 4 bytes is started from HDC1080.

0x04 is assigned to variable m_time.

HDC1080 processing that reads the conversion results (m_time=4)

The number of bytes in the reception buffer is checked.

If 4-byte measurement data has been received, the measurement data is read.

4-byte measurement data is read by 4 operations.

Figure 5.3 Main Function (2/4)
**Figure 5.4 Main Function (3/4)**

**Calculating the temperature and humidity from the measurement data**

The calculation result (in units of 0.1°C) obtained from the 1st and 2nd data is assigned to variable "temp".

The calculation result (in units of 0.1°C) obtained from the 1st and 2nd data is assigned to variable "humid".

**Creating an image of temperature indication displayed on the LCD indicator**

"-" is assigned to variable disp_line1[9]: A minus sign (-) is set.

"0 - variable temp" is assigned to Work_int.

Variable temp is assigned to Work_int: The value is copied.

" " is assigned to variable disp_line1[9]: The

"1" is assigned to variable disp_line1[9]: The hundreds place (°C) is set to 1.

"Variable Work_int - 1000" is assigned to variable Work_int.
Creating an image of temperature indication displayed on the LCD indicator

- "((work_int % 10) + 0x30)" is assigned to variable disp_line1[13].
- "work_int / 10" is assigned to variable work_int.

- "((work_int % 10) + 0x30)" is assigned to variable disp_line1[11].
- "work_int / 10" is assigned to variable work_int.

- "(work_int + 0x30)" is assigned to variable disp_line1[10].

Creating an image of humidity indication displayed on the LCD indicator

If variable "humid" is 100, the 100% indication is displayed.
- "1" is assigned to variable disp_line2[12].
- "0" is assigned to variable disp_line2[13].
- "0" is assigned to variable disp_line2[14].

- " " is set for variable disp_line2[12]: A blank is set.

- "(humid % 10) + 0x30" is assigned to variable disp_line2[14].
- "(humid / 10) + 0x30" is assigned to variable disp_line2[13].

The information displayed on the LCD indicator is updated.

0x00 is assigned to variable m_time: It is not when to perform measurement.
5.6.3 LCD Indicator Initialization Function
Figure 5.6 shows a flowchart of the LCD indicator initialization function.

```
init_LCD

Clear the LCD indicator: set_command()

Communication error occurred?

Yes

Set a return value indicating an error

No

The CLRDISP command is set for the LCD indicator.

2 is set as the return value and processing ends.

Wait for 3 ms: delay()

Set the operation mode: set_command()

Wait for 60 µs: delayMicroseconds()

Permit display: set_command()

Wait for 60 µs: delayMicroseconds()

Set the input mode: set_command()

Wait for 60 µs: delayMicroseconds()

The function waits for a time longer than the timeout value of the clear command.

The LCD_Mode command is set for the LCD indicator: 5x8 dots on 2 lines

The function waits for a time longer than the timeout value of the operation mode command.

The DISPON command is set for the LCD indicator: Indication is turned on but no cursor is displayed.

The function waits for a time longer than the timeout value of the DISPON command.

The ENTRY_Mode command is set for the LCD indicator: After data is set, control moves to the right.

Processing ends with the return value 0.
```

Figure 5.6 LCD Indicator Initialization Function
5.6.4 Function that Sets Full-Screen Display for the LCD Indicator

Figure 5.7 and Figure 5.8 show a flowchart of the function that sets full-screen display for the LCD indicator.

```
print_LCD

Set a data display pointer

Move the pointer to the beginning of line 1:
move_cursor()

Transfer the data to be displayed:
work = 0, work = 16, +1

Read the data to be displayed

Is the data null?

Yes

Replace the indication by a space

" " is assigned to variable "data": The value is replaced by a space (0x20).

No

Send the data to be displayed:
set_data()

Update the pointer

Wait for 60 µs:
delayMicroseconds()

Transfer the data to be displayed:

Set a data display pointer

Move the pointer to the beginning of line 2:
move_cursor()

A
```

The argument is assigned to variable p_addr.

Set the data display pointer at the beginning of line 1.

The function continues to send data to be displayed while the "work" counter is incremented from 0 to 15 and exits the loop when the counter value becomes 16.

"p_addr + 1" is assigned to variable p_addr.

The data to be displayed is sent to the LCD indicator.

The function waits until a data write becomes possible again.

"Argument + 40" is assigned to variable p_addr.

Set the data display pointer at the beginning of line 2.
The function continues to send data to be displayed while the "work" counter is incremented from 0 to 15 and exits the loop when the counter value becomes 16.

*p_addr is assigned to variable data.

" " is assigned to variable "data": The value is replaced by a space (0x20).

The data to be displayed is sent to the LCD indicator.

"p_addr + 1" is assigned to variable p_addr.

The function waits until a data write becomes possible again.

Figure 5.8  Function that Sets Full-Screen Display for the LCD Indicator (2/2)
5.6.5 Function that Sets the Data Display Position for the LCD Indicator

Figure 5.9 shows a flowchart of the function that sets the data display position for the LCD indicator.

![Flowchart for setting data display position](image)

- move_cursor
  - Create a cursor movement command
    - Send the command: `set_command()`
    - Wait for 60 µs: `delayMicroseconds()`
  - return

The argument is included in the DDRAM address specification command of the LCD indicator.
The DDRAM address specification command is set for the LCD indicator.
The function waits until a data write becomes possible again.

5.6.6 Function that Sets a Command for the LCD Indicator

Figure 5.10 shows a flowchart of the function that sets a command for the LCD indicator.

![Flowchart for setting command](image)

- set_command
  - Set the command byte
d  - Set the command code
  - Specify the slave: `Wire.beginTransmission()`
  - Set the send data: `Wire.write()`
  - Send data to the LCD indicator: `Wire.endTransmission()`
  - return

COMBYTE is assigned to variable `g_lcd_command[0]`.
The argument is assigned to variable `g_lcd_command[1]`.
The slave address of the LCD indicator is specified.
The content of variable `g_lcd_command` is set in the buffer.
Transmission to the LCD indicator is performed.
The function sets the processing status (transmission result) as a return value and control returns.
5.6.7 Function that Sets Data for the LCD Indicator

Figure 5.11 shows a flowchart of the function that sets data for the LCD indicator.

```
set_data

Set the command byte

Set the command code

Specify the slave:
Wire.beginTransmission()

Set the send data:
Wire.write()

Send data to the LCD indicator:
Wire.endTransmission()

return

DATABYTE is assigned to variable g_lcd_data[0].

The argument is assigned to variable g_lcd_data[1].

The slave address of the LCD indicator is specified.

The content of variable g_lcd_data is set in the buffer.

The content of variable g_lcd_data is set in the buffer.

The function sets the processing status (transmission result) as a return value and control returns.
```

Figure 5.11 Function that Sets Data for the LCD Indicator
6. Sample Code
Sample code can be downloaded from the Renesas Electronics website.

7. Reference Documents
RL78/G14 User's Manual: Hardware (R01UH0186)
RL78 family User's Manual: Software (R01US0015)
RL78/G14 Fast Prototyping Board User’s Manual (R20UT4573)
(The latest versions can be downloaded from the Renesas Electronics website.)

Technical Update/Technical News
(The latest versions can be downloaded from the Renesas Electronics website.)

Website and Support
Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/contact/

All trademarks and registered trademarks are the property of their respective owners.
<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Description</th>
<th>Page</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>2021.11.01</td>
<td>—</td>
<td>—</td>
<td>First Edition</td>
</tr>
</tbody>
</table>
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 be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.

5. 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.

6. 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 systems; underwater 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.

7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.

8. 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.

9. 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.

10. 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.

11. 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.

12. 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.

13. This document shall not be reproduced, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.

14. 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.

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

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/

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