# DS18B20 Library Documentation

Revision 1.0

Luka Gacnik

October 2023

# Table of contents

# Table of figures

# Acronyms

| | |
|---|---|
| **MCU** | Microcontroller Unit |

# 1 Library description

The library consists of basic functions that control the DS18B20 device over the OneWire bus, made by Maxim Integrated.

## 1.1 About the DS18B20

The DS18B20 is a temperature sensor which offers a digital interface. This makes it configurable for conversion resolution, setting alarm trigger points, and handling data between its internal RAM (scratchpad) and EEPROM (non-volatile memory). Benefits and features of the DS18B20 device:

- Measures Temperatures from -55°C to +125°C
- ± 0.5Â°C Accuracy from -10°C to +85°C
- Programmable Resolution from 9 Bits to 12 Bits
- No External Components Required
- Unique 64-bit ROM Serial Code for each device
- Flexible EEPROM Settings

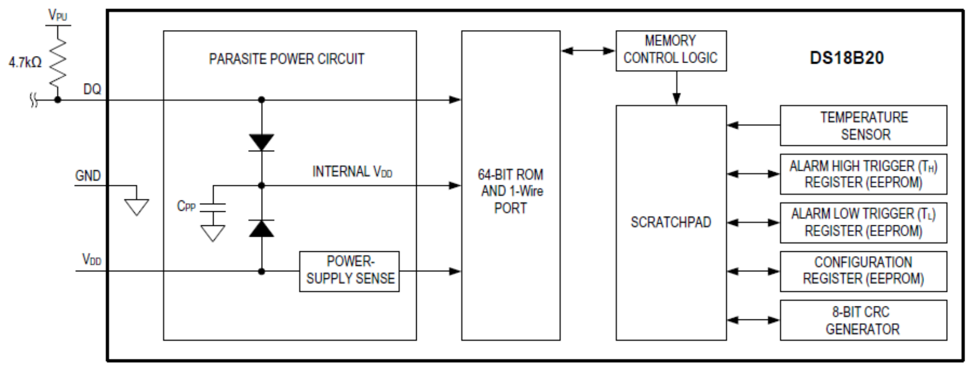

Figure 1.1: DS18B20 Block Diagram

## 1.2 Library features

Currently, the DS18B20 driver is capable of performing the following operations:

- DS18B20 search/scan over OneWire bus
- DS18B20 configuration
- DS18B20 temperature convert and read (polling and non-polling operation)

# 2 Dependencies

The library consists of the following source and header files:

- `DS18B20.c`
- `DS18B20.h`

All the external user-accessible function prototypes are provided in the `DS18B20.h` file. Their definitions and other local functions are contained in the `DS18B20.c` file.

The library depends on the following other libraries:

- `OneWire.h`
- `Edc.h`

The library `OneWire.h` provides the necessary functions that handle data transfer between MCU and DS18B20 device(s). The other library, that is `Edc.h`, provides functions that perform CRC over read data bytes from DS18B20 device(s).

Lastly, its important to provide the fact that this library was meant to be used with a specific MCU - the PIC32MX170Bxxx series made by the Microchip. The XC32 compiler was used to compile the library files, meaning there are some specifics in the library which are unique to the XC32 compiler. In case any other MCU of the PIC32 family would be used, some peripheral libraries (used by this driver) might need modifications (like different register sets, etc.).

# 3 Notes and warnings

There are few constrains which limit the capability of the DS18B20 library:

a) **Single OneWire bus only**

   The structure of the DS18B20 library code allows only for a single OneWire bus per all DS18B20 devices that are to be addressed using this library. This shouldn't be of any problem as all of DS18B20 devices in a system can be addressed using one OneWire bus. This is limited to the DS18B20 library alone and doesn't apply to the OneWire library, which is required by the DS18B20 library.

b) **Importance of System Clock Frequency**

   Many functionalities throughout this library rely on the timeout mechanism (which uses Core Timer). Each time before timeout is used a System Clock Frequency is used to calculate the proper timeout value. Therefore any changes to System Clock should be executed prior to configuring DS18B20 device(s).

   Additionally, OneWire library that is used for the DS18B20 to function properly. OneWire uses delay functions from Timer library, which requires the macro `TMR_DELAY_SYSCLK` to be set prior to using any of available delay functions.

## 4 Future ideas

None.

# 5 Custom types

The library comes with a set of custom made structure or enumeration types. For the sake of convenience these types are explained throughout this chapter. Note that individual enumeration types are not covered here because they (and their set of values) follow self-explanatory naming conventions.

## 5.1 DsConfig_t

This structure-based type contains the properties for configuration of a selected DS18B20 device. The properties of this type define parameters, which are used to configure a DS18B20 device. The type consists of the following properties:

- **`owConfig`** (*`OwConfig_t`*) - configuration structure for setting up the OneWire bus
- **`measRes`** (*`DsMeasRes_t`*) - defines resolution (and therefore time period) of a temperature conversion
- **`deviceId`** (*`uint64_t`*) - ROM identification number of DS18B20 device to be configured (irrelevant in multi-device configuration)
- **`lowAlarm`** (*`int`*) - defines the lower temperature threshold below which alarm flag is triggered
- **`highAlarm`** (*`int`*) - defines the upper temperature threshold above which alarm flag is triggered

Note that only parameters `owConfig` and `deviceId` (only when using a single device) are required to be set to proper values.

# 6 Function description

The library includes the following functions:

- `DS18B20_SearchDeviceId()`
- `DS18B20_SearchAlarm()`
- `DS18B20_ConfigDevice()`
- `DS18B20_SaveToRom()`
- `DS18B20_CopyFromRom()`
- `DS18B20_SetCorrection()`
- `DS18B20_IsDeviceFake()`
- `DS18B20_IsConvDone()`
- `DS18B20_ConvertReadTemp()`
- `DS18B20_ConvertTemp()`
- `DS18B20_ReadTemp()`
- `DS18B20_ReadRam()`

The following sub-sections describe the use and operationality of these functions.

## 6.1 DS18B20_SearchDeviceId()

**PROTOTYPE**

`DS18B20_SearchDeviceId(pinCode)`

**DESCRIPTION**

Performs ROM ID device search according to the predefined OneWire search algorithm.

**PARAMETERS**

- **`pinCode`** (*`uint32_t`*) - pre-defined pin code that passes key information about the current pin to the PIO related functions. Possible pin values (in this case `GPIO_RPBx`, where `x` is pin's number) are provided in the `Pio_sfr.h` file
- **`romIdBuff`** (*`uint64_t *`*) - an address of a reserved buffer for DS18B20 ROM IDs data storage

**RETURNS**

Returns a device count of identified DS18B20 devices.

**OPERATION**

During the first call (after device reset) CRC look-up table is generated for use of CRC check when verifying transmitted ROM IDs from DS18B20 devices. OneWire bus is set up, then OneWire search begins. Operation of this algorithm is predefined by the OneWire standard. A timeout is set up to prevent possible infinite loop in case of a broken connection during the search operation. Timeout is user-defined (in milliseconds) and may be modified via `DS_CONV_TEMP_TIMEOUT_MS` macro in the `DS18B20.h` file. Additional safety measure is used where device search is repeated for the `DS_SEARCH_DEVICE_REPEAT_COUNT` times if either of read ROM ID's CRC fails.

**NOTES**

- Any other device using the same OneWire bus will not be detected during the search operation

- For all other functions requiring ROM IDs one could omit using the `DS18B20_SearchDeviceId()` function if ROM IDs of given DS18B20 devices are already identified. In that case ROM IDs used and passed to specific functions need to be passed as 48-bit long device specific and unique codes (without generated CRC and familiy code appended to it)

## ADDITIONAL NOTE

This function could be used effectively in the following manner:

- Perform ROM ID search
- Identify individual devices by performing temperature reads and determine which is which based on read temperatures for each
- Save acquired codes into system's non-volatile memory (EEPROM)
- Next time device is reset, use EEPROM data of ROM IDs to address known devices
- Repeat this process any time a new devices is added to the system or a manual scan check is required by user

## LIMITATIONS

None.

## 6.2 DS18B20_SearchAlarm()

### PROTOTYPE

`DS18B20_SearchAlarm(pinCode)`

### DESCRIPTION

Performs ROM ID device search according to the predefined OneWire search algorithm, where only devices with alarm flag set will respond.

### PARAMETERS

- **`pinCode`** (*`uint32_t`*) - pre-defined pin code that passes key information about the current pin to the PIO related functions. Possible pin values (in this case `GPIO_RPBx`, where `x` is pin's number) are provided in the `Pio_sfr.h` file
- **`romIdBuff`** (*`uint64_t *`*) - an address of a reserved buffer for DS18B20 ROM IDs data storage

### RETURNS

Returns an alarm device count of identified DS18B20 devices, whose alarm flag has been set.

### OPERATION

The operation is identical to the operation of `DS18B20_SearchDeviceId()` expect that a different command is send to all DS18B20 devices.

### NOTES

Any other device using the same OneWire bus will not be detected during the search operation.

### LIMITATIONS

None.

## 6.3 DS18B20_ConfigDevice()

### PROTOTYPE

`DS18B20_ConfigDevice(dsConfig, isMultiMode)`

### DESCRIPTION

Configures single/multiple device(s) according to the passed configuration structure.

### PARAMETERS

- **`dsConfig`** (*`DsConfig_t`*) - contains all configuration parameters that are available for DS18B20 devices
- **`isMultiMode`** (*`Bool_t`*) - defines whether multiple devices are to be configured

### RETURNS

Returns `true` if configuration was successful, else `false`.

### OPERATION

Desired DS18B20 settings are converted into raw form required by DS18B20's registers and written to device(s). In case of multiple devices, all devices on the OneWire bus will be addressed.

### NOTES

- In case of configuring a single device it needs to be ensured that the `romId` property of the `dsConfig` structure is set to a proper 48-bit long, device unique value. The `romId` parameter is irrelevant in case of configuring multiple devices and can be passed as `NULL`
- It is advised that search function is called prior to configuring any device. Based on information obtained about available devices that are currently present on the OneWire bus, user can decide whether to configure a single or multiple devices

### LIMITATIONS

Either of alarm settings should be within range of -55°C and +125°C.

## 6.4 DS18B20_SaveToRom()

### PROTOTYPE

`DS18B20_SaveToRom(romId, isMultiMode)`

### DESCRIPTION

Issues a data transfer from DS18B20's internal scratchpad (RAM) to EEPROM.

### PARAMETERS

- **`romId`** (*`uint64_t *`*) - an address of memory where unique 48-bit long device ROM ID code(s) are stored
- **`isMultiMode`** (*`Bool_t`*) - defines whether multiple devices are to be configured

### RETURNS

Returns `true` if operation was successful, else `false`. The latter is also the case when internal timeout is reached.

### OPERATION

A specific command is issued to DS18B20 device(s) which starts moving data from internal RAM to EEPROM. If broken connection occurs during data transfer or possible device failure, then loop is exited due to timeout.

### NOTES

- In case of configuring a single device in needs to be ensured that `romId` property of the `dsConfig` structure is set to a proper value (this value should be only device's ROM ID without additional CRC or family code appended to it)
- It is advised that search function is called prior to configuring any device. Based on information obtained about available devices that are currently present on the OneWire bus, user can decide whether to configure a single or multiple devices
- Additionally, it is advised that devices are configured to the desired settings prior to calling this function

- In multi-device mode the `romId` pointer is irrelevant and can be set to `NULL` because all devices are addressed simultaneously

**LIMITATIONS**

None.

## 6.5 DS18B20_CopyFromRom()

### PROTOTYPE

`DS18B20_CopyFromRom(romId, isMultiMode)`

### DESCRIPTION

Issues a data transfer from DS18B20's internal EEPROM to scratchpad (RAM).

### PARAMETERS

- **`romId`** (*`uint64_t *`*) - an address of memory where unique 48-bit long device ROM ID code(s) are stored
- **`isMultiMode`** (*`Bool_t`*) - defines whether multiple devices are to be configured

### RETURNS

Returns `true` if operation was successful, else `false`. The latter is also the case when internal timeout is reached.

### OPERATION

A similar operation to the `DS18B20_SaveToRom()` except a different command is used to issue the data transfer.

### NOTES

- In case of configuring a single device in needs to be ensured that `romId` property of the `dsConfig` structure is set to a proper value (this value should be only device's ROM ID without additional CRC or family code appended to it)
- It is advised that search function is called prior to configuring any device. Based on information obtained about available devices that are currently present on the OneWire bus, user can decide whether to configure a single or multiple devices
- On the DS18B20 device start-up EEPROM content is automatically transferred to its scratchpad. However, since configuration function overwrites scratchpad, the function `DS18B20_CopyFromRom()` should be called afterwards

- In multi-device mode the `romId` pointer is irrelevant and can be set to `NULL` because all devices are addressed simultaneously

**LIMITATIONS**

None.

## 6.6 DS18B20_SetCorrection()

#### PROTOTYPE

`DS18B20_SetCorrection(corr)`

#### DESCRIPTION

Modifies static variable which is used throughout the DS18B20 library to add a correction to all calculations, where temperature is being calculated from raw form. Correction is also used for configuring alarm settings.

#### PARAMETERS

- **`corr`** (*`float`*) - floating point correction value

#### RETURNS

Returns `true` if correction value was within allowable limits, else it returns `false`.

#### OPERATION

Modifies a static variable.

#### NOTES

If alarm functionality is used then after calling this function also the `DS18B20_ConfigDevice()` function must be called to modify alarm values before written to a device.

#### LIMITATIONS

Correction should be within range of -55.0°C and +125.0°C.

## 6.7 DS18B20_IsDeviceFake()

**PROTOTYPE**

`DS18B20_IsDeviceFake(romId)`

**DESCRIPTION**

Verifies whether a specific DS18B20 device is a fake device.

**PARAMETERS**

- **`romId`** (*`uint64_t *`*) - an address of memory where unique 48-bit long device ROM ID code(s) are stored

**RETURNS**

Returns `true` if the device being addressed is a non-valid DS18B20 device, else function returns `false`. The latter is also returned value when device presence is not detected.

**OPERATION**

A device is configured to a 9-bit resolution and temperature conversion is executed. If device fails to complete conversion until the timeout for the 9-bit resolution is reached, then this device is considered a non-valid DS18B20 device.

**NOTES**

- Currently detected non-valid DS18B20 devices, which have a fixed conversion time regardless of their resolution settings, seems to operate properly and return valid results. However it is possible that some non-valid DS18B20 device might not be functioning properly

- Validity of a device is determined by the means of checking its temperature conversion time. It is possible that other non-valid DS18B20 devices respond differently from currently known non-valid devices, and therefore such check might return false result

- Be aware that upon detecting a fake device, one should either use polling-based temperature conversion function, or reconfigure that device to 12-bit resolution and use internal timer based functions, or use functions without internal timer and measure time which is needed for all fake devices to successfully perform conversion. If non-internal timer approach is selected then it is advised to always use `DS18B20_IsConvDone()` check as additional safety check in conjunction with external, user-defined timer

- Only one device is addressed in this case per function call, therefore address `romId` should be pointing to a specific ROM ID value

**LIMITATIONS**

None.

## 6.8 DS18B20_IsConvDone()

### PROTOTYPE

`DS18B20_IsConvDone()`

### DESCRIPTION

Verifies whether any of DS18B20 devices on OneWire bus is executing a temperature conversion.

### PARAMETERS

None.

### RETURNS

Returns `true` if neither of DS18B20 devices is currently converting temperature, else it returns `false`.

### OPERATION

Carries out a simple OneWire read bit. If any of DS18B20 devices is pulling bus low then at least one of those devices is still doing a temperature conversion.

### NOTES

This function should be used in combination with interrupt-based functions without internal timer.

### LIMITATIONS

None.

## 6.9 DS18B20_ConvertReadTemp()

### PROTOTYPE

`DS18B20_ConvertReadTemp(romId, dataBuff, deviceCount)`

### DESCRIPTION

Executes a polling-based temperature conversion with internal timeout and reads conversion results afterwards.

### PARAMETERS

- **`romId`** (*`uint64_t *`*) - an address of memory where unique 48-bit long device ROM ID code(s) are stored
- **`dataBuff`** (*`float *`*) - an address of a reserved buffer for converted temperature data storage
- **`deviceCount`** (*`uint8_t`*) - an amount of devices to be addressed

### RETURNS

Returns `true` after a successful temperature conversion, else it returns `false`. The latter is possible upon false function inputs, presence check failed, timeout, or a failed CRC check.

### OPERATION

Initially, a temperature conversion is carried out. In case of multiple devices all devices are addressed simultaneously and in case of single device, it is addressed using its unique ROM ID code. Then conversion done check is done via polling the device's output state. If loop is exited before timeout is reached, read of all devices, whose addresses were passed to this function via pointer, is executed. After end of each read a CRC is calculated and compared to the transmitted CRC code from a DS18B20 device. Lastly, all devices have their raw data converted to Celsius.

## NOTES

- The allowable amount of read restart repetitions upon failed CRC code check is determined by the user-accessible `DS_READ_RAM_REPEAT_COUNT` macro
- User must provide sufficient memory space for `dataBuff` whose size equals to `deviceCount`
- In case multiple devices are to be read, the `dataBuff` needs to provide sufficient memory space; that is at least the size of `deviceCount` value
- Converted temperature values are stored within `dataBuff` in the same order as the ROM ID codes given by `romId`

## LIMITATIONS

None.

## 6.10 DS18B20_ConvertTemp()

### PROTOTYPE

`DS18B20_ConvertTemp(romId, deviceCount)`

### DESCRIPTION

Initiates a temperature conversion.

### PARAMETERS

- **`romId`** (*`uint64_t *`*) - an address of memory where unique 48-bit long device ROM ID code(s) are stored
- **`deviceCount`** (*`uint8_t`*) - an amount of devices to be addressed

### RETURNS

Returns `true` after a successful temperature conversion has been initiated, else it returns `false`. The latter case occurs upon false function inputs or presence check failed.

### OPERATION

Initially, a temperature conversion is carried out. In case of multiple devices all devices are addressed simultaneously and in case of single device, it is addressed using its unique ROM ID code.

### NOTES

- Conversion done may be checked using `DS18B20_IsConvDone()` function
- In multi-device mode the `romId` pointer is irrelevant and can be set to `NULL` because all devices are addressed simultaneously (convert temperature operation only)

### LIMITATIONS

None.

## 6.11 DS18B20_ReadTemp()

### PROTOTYPE

`DS18B20_ReadTemp(romId, dataBuff, deviceCount)`

### DESCRIPTION

Acquires and converts raw temperature data from DS18B20 device.

### PARAMETERS

- **`romId`** (*`uint64_t *`*) - an address of memory where unique 48-bit long device ROM ID code(s) are stored
- **`dataBuff`** (*`float *`*) - an address of a reserved buffer for converted temperature data storage
- **`deviceCount`** (*`uint8_t`*) - an amount of devices to be addressed

### RETURNS

Returns `true` after successful temperature conversion, else it returns `false`. The latter is possible upon false function inputs, presence check failed, conversion done check failed, or a failed CRC check.

### OPERATION

Conversion done check is done prior to reading any data. Then read of all devices, whose addresses were passed to this function via pointer, is carried out. After end of each read, a CRC is calculated and compared to the transmitted CRC code from a DS18B20 device. Lastly, all devices have their raw data converted to Celsius.

### NOTES

- The allowable amount of read restart repetitions upon failed CRC code check is determined by the user-accessible `DS_READ_RAM_REPEAT_COUNT` macro
- In case multiple devices are to be read, the `dataBuff` needs to provide sufficient memory space; that is at least the size of `deviceCount` value

- Converted temperature values are stored within `dataBuff` in the same order as the ROM ID codes given by `romId`

**LIMITATIONS**

None.

## 6.12 DS18B20_ReadRam()

### PROTOTYPE

`DS18B20_ReadRam(romId, dataBuff, deviceCount)`

### DESCRIPTION

Reads high alarm, low alarm, and measurement resolution of a single device.

### PARAMETERS

- **`romId`** (*`uint64_t *`*) - an address of memory where unique 48-bit long device ROM ID code(s) are stored
- **`dataBuff`** (*`int *`*) - an address of a reserved buffer for scratchpad (RAM) data storage
- **`deviceCount`** (*`uint8_t`*) - an amount of devices to be addressed

### RETURNS

Returns `true` after successful device read, else it returns `false`. The latter is possible upon false function inputs, presence check failed, bus busy, or a failed CRC check.

### OPERATION

Follows the same procedure as the `DS18B20_ReadTemp` function except that only high alarm, low alarm, and measurement resolution values are read. The latter is given as *`int`* but corresponds to values of the *`DsMeasRes_t`* type.

### NOTES

- The allowable amount of read restart repetitions upon failed CRC code check is determined by the user-accessible `DS_READ_RAM_REPEAT_COUNT` macro
- In case multiple devices are to be read, the `dataBuff` needs to provide sufficient memory space; that is at least the size of 3·`deviceCount` value
- Converted temperature values are stored within `dataBuff` in the same order as the ROM ID codes given by `romId`

**LIMITATIONS**

None.

# 7 Examples

An example of using the described library is provided. Examples provide only the main operation related code. Other processes such as the start-up of an MCU, programming of the device configuration register, setting up other peripheral modules, and similar is not provided here.

## 7.1 Sample code

This example shows how a multi DS18B20 system can be handled. The shown example follows a given procedure:

- Device IDs are determined
- Check for any fake devices present
- All devices are configured to same configuration
- Store alarm settings into device's EEPROM
- Non-polling temperature conversion with internal timeout
- Read (and view) data
- Reconfigure devices to different (false) alarm setting
- Convert and read data in polling mode + check alarm flags (ROMs)
- Restore original alarm settings from device's EEPROM
- Convert and read data in polling mode + check alarm flags (ROMs)

```c
/** Required by OneWire.h for Tmr.h **/
#define TMR_DELAY_SYSCLK 40000000

/** Custom libs **/
#include "DS18B20.h"

int main (int argc, char** argv)
{
        /* Pin toggle during timeout */
    PIO_ClearPin(GPIO_RPB4);
    PIO_ConfigGpioPin(GPIO_RPB4, PIO_TYPE_DIGITAL, PIO_DIR_OUTPUT);

    OwConfig_t owConfigBus =
    {
        .pinCode = GPIO_RPB5,
        .speedMode = OW_STANDARD_SPEED
    };

    /* Configuration structure for multiple devices */
    DsConfig_t dsConfig = {
        .measRes = DS_MEAS_RES_12BIT,
        .owConfig = owConfigBus,
        .highAlarm = 40,
        .lowAlarm = 24
    };

    /* Identify all DS18B20 devices */
    uint64_t romId[10] = {0};
    uint32_t deviceCount;
    deviceCount = DS18B20_SearchDeviceId(owConfigBus.pinCode, romId);

    /* Check if any are fake - have fixed conversion time */
    for (uint8_t idx = 0; idx < deviceCount; idx++)
    {
        if (DS18B20_IsDeviceFake(&romId[idx]))
        {
            dsConfig.measRes = DS_MEAS_RES_12BIT;
            break;
        }
    }

    /* Multiple devices on OW bus check */
    Bool_t isMultiMode;
    isMultiMode = (deviceCount > 1) ? true : false;

    float data[deviceCount];

    /* Configure device */
    if (DS18B20_ConfigDevice(dsConfig, isMultiMode))
    {
        /* Store alarm settings */
        DS18B20_SaveToRom(romId, isMultiMode);

        /* Start temperature conversion with internal timeout */
        DS18B20_ConvertTemp(romId, deviceCount);

        /* Wait for timeout */
        while (!DS18B20_IsConvDone())
        {
            PIO_TogglePin(GPIO_RPB4);
        }

        /* Store results */
        DS18B20_ReadTemp(romId, data, deviceCount);
    }
    else
    {
        /* Do not proceed until device successfully configured */
    }

    /* Do alarm flag search */
    uint64_t alarmRomId[10];
    uint32_t alarmCount;
    alarmCount = DS18B20_SearchAlarm(owConfigBus.pinCode, alarmRomId);

    /* Devices at 25-40$\degree$C won't have their alarm flags set */

    /* Reconfigure to another alarm setting */
    dsConfig.lowAlarm = 0;
    dsConfig.highAlarm = 15;
    DS18B20_ConfigDevice(dsConfig, isMultiMode);

    /* Do another conversion */
    DS18B20_ConvertReadTemp(romId, data, deviceCount);

    /* Do another alarm search */
    alarmCount = DS18B20_SearchAlarm(owConfigBus.pinCode, alarmRomId);

    /* Alarm should be now triggered for devices above 15$\degree$C */

    /* Restore alarm settings */
    DS18B20_CopyFromRom(romId, isMultiMode);

    /* Check one of devices' EEPROM if successfully copied */
    int ramData[3];
    DS18B20_ReadRam(romId, ramData, 1);

    /* Do third conversion */
    DS18B20_ConvertReadTemp(romId, data, deviceCount);

    /* Do third alarm search */
    alarmCount = DS18B20_SearchAlarm(owConfigBus.pinCode, alarmRomId);

    /* This time devices at 25-40$\degree$C won't have their alarm
   flags set */

    /* Example code end */
    while (1)
    {}

    return 0;
}
```

# A Revision history

**Revision 1.0 (April 2023)**

This is the initial released version of this document.