Contents

1. Overview

In order to use the MLX90641 driver there are 4 files that should be included in the C project:

- *MLX90641_I2C_Driver.h* – header file containing the definitions of the I2C related functions
- *MLX90641_I2C_Driver.cpp or MLX90641_SWI2C_Driver.cpp* – file containing the I2C related functions
- *MLX90641_API.h* – header file containing the definitions of the MLX90641 specific functions
- *MLX90641_API.h* – file containing the MLX90641 specific functions

The user

2. I2C driver

This is the driver for the I2C communication. The user should change this driver accordingly so that a proper I2C with the MLX90641 is achieved. As the functions are being used by the MLX90641 API the functions definitions should not be changed. The I2C standard reads LSByte first reversing the endianness of the data. Note that the driver is also responsible to reconstruct the proper endianness. If that part of the code is changed, care should be taken so that the data is properly restored. There are two I2C drivers that could be used:

2.1. *MLX90641_I2C_Driver.cpp*

This file should be included if the hardware I2C if the user MCU is to be used. The user should adapt it in order to utilize the I2C hardware module in the chosen MCU. Most MCU suppliers offer libraries with defined functions that could be used.

2.2. *MLX90641_SWI2C_Driver.cpp*

This file implements a software I2C communication using two general purpose IOs of the MCU. The user should define the IOs (*sda* and *scl*) and ensure that the correct timing is achieved.

- Defining the IOs – I2C data pin should be defined as an InOut pin named '*sda*',
  I2C clock pin should be defined as an Output pin names '*scl*'
- Defining the IOs levels – in order to work properly with different hardware implementations the *scl* and *sda* levels could be defined as follows:
  - *#define LOW 0;* - low level on the line (default '0'), could be '1' if the line is inverted
  - *#define HIGH 1;* - high level on the line (default '1'), could be '0' if the line is inverted

  #define SCL_HIGH scl = HIGH; - I2C clock high level definition

  #define SCL_LOW scl = LOW; - I2C clock low level definition

```
#define SDA_HIGH sda.input(); - I2C data high level definition

#define SDA_LOW sda.output(); \  - I2C data low level definition

        sda = LOW;
```

The '*sda*' pin is being switched to input for high level and to output for low level in order to allow proper work for devices that do not support open drain on the pins. This approach mimics open drain behaviour. If the device supports open drain, the definitions to set the *sda* line low and /or high could be changed.

- Setting the I2C frequency – as this is a software implementation of the I2C, the instruction cycle and the MCU clock affect the code execution. Therefore, in order to have the correct speed the user should modify the code so that the '*scl*' generated when the MLX90641 is transmitting data is with the desired frequency. The default implementation of the wait function is:

```
void Wait(int freqCnt)

{

        int cnt;

        for(int i = 0;i<freqCnt;i++)

        {

           cnt = cnt++;

        }

}
```

The *Wait* function could be modified in order to better trim the frequency. For coarse setting of the frequency (or dynamic frequency change) using the dedicated function, '*freqCnt*' argument should be changed – lower value results in higher frequency.

2.3. I2C driver functions

The I2C driver has four main functions that ensure the proper communication between the user MCU and the MLX90641. Those functions might need some modifications by the user. However, it is important to keep the same function definitions.

2.3.1. void MLX90641_I2CInit(void)

This function should be used to initialize the I2C lines (sda and scl) and the I2C hardware module if needed. The initial state of the I2C lines should be high. The default implementation in the I2C driver is sending a stop condition.

Example:

1. *The initialization of the I2C should be done in the beginning of the program in order to ensure proper communication*

   main.c

   …definitions…

   ..MCU initialization

   MLX90641_I2CInit();

   …

   …MLX90641 communication

   …User code

2.3.2. void MLX90641_I2CFreqSet(int freq)

This function should be used to dynamically change the I2C frequency and/or for coarse settings. It has one parameter of type *int*. This parameter is used to set the frequency for a hardware I2C module or the number of cycles in the *Wait* function in the software I2C driver. When using I2C hardware module, the MCU supplier provides library with integrated function for changing the frequency. In that case the MLX90641 I2C driver should be changed so that it uses the library function to set the frequency. In the software I2C driver (when using two general purpose IOs) the *I2CFreqSet* function sets a global variable that is being used by the *Wait* function to set the number of loops. In order to set properly the frequency, the user should trim the *Wait* function so that the generated I2C clock has the desired frequency when a MLX90641 device is transmitting data.

Example:

1. *Setting the I2C frequency to 1MHz for frame data read when using a hardware I2C module:*

   MLX90641_I2CFreqSet(1000); //in this case the library function provided by the MCU supplier

   // requires int value in KHz -> 1000KHz = 1MHz

2. *Setting the I2C frequency to 400KHz for frame data read when using a software I2C implementation:*

   MLX90641_I2CFreqSet(20); //Depending on the instruction cycle and the clock of the MCU, 20

   //cycles in the Wait function result in 400KHz frequency of the

   //MLX90641 is transmitting data generated scl when

2.3.3. int MLX90641_I2CRead(uint8_t slaveAddr, *uint16*_t startAddress, *uint16*_t nWordsRead, uint16_t *data)

This function reads a desired number of words from a selected MLX90641 device memory starting from a given address and stores the data in the MCU memory location defined by the user. If the returned value is

0, the communication is successful, if the value is -1, NACK occurred during the communication. The function needs the following parameters:

- *uint8_t slaveAddr* – Slave address of the MLX90641 device (the default slave address is 0x33)
- *uint16_t startAddress* – First address from the MLX90641 memory to be read. The MLX90641 EEPROM is in the address range 0x2400 to 0x273F and the MLX90641 RAM is in the address range 0x0400 to 0x073F.
- *uint16_t nMemAddressRead* – Number of 16-bits words to be read from the MLX90641 memory
- *uint16_t *data* – pointer to the MCU memory location where the user wants the data to be stored

Example:

1. *Reading a single EEPORM value – MLX90641 settings for a MLX90641 device with slave address 0x33:*

   uint16_t eeValue;

   MLX90641_I2CRead(0x33, 0x240C, 1, &eeValue); //the EEPROM 0x240C cell value is stored in eeValue

2. *Reading the whole EEPROM data for a MLX90641 devices with slave address 0x33:*

   uint16_t eeMLX90641[832];

   MLX90641_I2CRead(0x33, 0x2400, 832, eeMLX90641); //the EERPOM data is stored in the eeMLX90641 array

2.3.4. int MLX90641_I2CWrite(uint8_t slaveAddr, unsigned int writeAddress, uint16_t data)

This function writes a 16-bit value to a desired memory address of a selected MLX90641 device. The function reads back the data after the write operation is done and returns 0 if the write was successful, -1 if NACK occurred during the communication and -2 if the data in the memory is not the same as the intended one. The following parameters are needed:

- *uint8_t slaveAddr* – Slave address of the MLX90641 device (the default slave address is 0x33)
- *unsigned int* writeAddress – The MLX90641 memory address to write data to
- *uint16_t* data - Data to be written in the MLX90641 memory address

Example:

1. *Writing settings – MLX90641 settings for a MLX90641 device with slave address 0x33:*

   int status;

   status = MLX90641_I2CWrite(0x33, 0x800D, 0x0901); //the desired settings are written to address 0x800D

   Variable status is 0 if the write was successful.

2.3.5. int MLX90641_I2CGeneralReset (void)

This function should implement the standard reset condition for the I2C. According to the I2C specification the reset condition is sending 0x06 to address 0x00. The function returns 0 when the communication is successful and -1 if NAK occurred during the communication. Note that this function would reset all devices on the bus that are supporting it.

Example:

1. *Resetting all devices on the bus:*

   int status;

   status = MLX90641_I2CGeneralReset(); //the MLX90641 device is reset.

   //Variable status is 0 if the communication was successful.

3. MLX90641 API

This is the driver for the MLX90641 device. The user should not change this driver.

3.1. MLX90641 configuration functions

3.1.1. int MLX90641_SetResolution(uint8_t slaveAddr, uint8_t resolution)

This function writes the desired resolution value (0x00 to 0x03) in the appropriate register in order to change the current resolution of a MLX90641 device with a given slave address. Note that after power-on reset, the resolution will revert back to the resolution stored in the EEPROM. The return value is 0 if the write was successful, -1 if NACK occurred during the communication and -2 if the written value is not the same as the intended one.

- *uint8_t slaveAddr* – Slave address of the MLX90641 device (the default slave address is 0x33)
- *uint8_t* resolution – The current resolution of MLX90641
  - 0x00 – 16-bit resolution
  - 0x01 – 17-bit resolution
  - 0x02 – 18-bit resolution
  - 0x03 – 19-bit resolution

Example:

1. *Setting MLX90641 device at slave address 0x33 to work with 19-bit resolution:*

   Int status;

   status = MLX90641_SetResolution(0x33,0x03);

3.1.2. int MLX90641_GetCurResolution (uint8_t slaveAddr)

This function returns the current resolution of a MLX90641 device with a given slave address. Note that the current resolution might differ from the one set in the EEPROM of that device. If the result is -1, NACK occurred during the communication and this is not a valid resolution data.

- *uint8_t slaveAddr* – Slave address of the MLX90641 device (the default slave address is 0x33)

Example:

1. *Getting the current resolution from a MLX90641 device at slave address 0x33 that works with 19-bit resolution, but has in the EEPROM programmed 16-bit resolution:*

   int curResolution;

   curResolution = MLX90641_GetCurResolution(0x33); //curResolution = 0x03(19-bit) as this is the actual

   //resolution the device is working with

3.1.3. int MLX90641_SetRefreshRate (uint8_t slaveAddr, uint8_t refreshRate)

This function writes the desired refresh rate value (0x00 to 0x07) in the appropriate register in order to change the current refresh rate of a MLX90641 device with a given slave address. Note that after power-on reset, the refresh rate will revert back to the refresh rate stored in the EEPROM. The return value is 0 if the write was successful, -1 if NACK occurred during the communication and -2 if the written value is not the same as the intended one.

- *uint8_t slaveAddr* – Slave address of the MLX90641 device (the default slave address is 0x33)
- *uint8_t refreshRate* – The current refresh rate of MLX90641 device
  - 0x00 – 0.5Hz
  - 0x01 – 1Hz
  - 0x02 – 2Hz
  - 0x03 – 4Hz
  - 0x04 – 8Hz

- 0x05 – 16Hz
- 0x06 – 32Hz
- 0x07 – 64Hz

Example:

1. *Setting MLX90641 device at slave address 0x33 to work with 16Hz refresh rate:*

   int status;

   status = MLX90641_SetRefreshRate *(0x33,0x05);*

3.1.4. int MLX90641_GetRefreshRate (uint8_t slaveAddr)

This function returns the current refresh rate of a MLX90641 device with a given slave address. Note that the current refresh rate might differ from the one set in the EEPROM of that device. If the result is -1, NACK occurred during the communication and this is not a valid refresh rate data.

- *uint8_t slaveAddr* – Slave address of the MLX90641 device (the default slave address is 0x33)

Example:

1. *Getting the current refresh rate from a MLX90641 device at slave address 0x33 that works with 16Hz resolution, but has in the EEPROM programmed 0.5Hz refresh rate:*

   int curRR;

   *curRR = MLX90641*_GetRefreshRate *(0x33);* *// curRR = 0x05(16Hz) as this is the actual*

   //refresh rate the device is working with

3.1.5. int MLX90641_GetSubPageNumber (uint16_t *frameData)

This function returns the sub-page for a selected frame data of a MLX90641 device.

- uint16_t *frameData – pointer to the MLX90641 frame data that is already acquired

Example:

1. *Getting the sub-page for a selected frame data of a MLX90641 device:*

   static int mlx90641Frame[242];

   int 9ubpage;

   9ubpage = MLX90641_GetSubPageNumber(mlx90641Frame); *// 9ubpage = 1 as this is the actual*

//sub-page number for that frame

3.1.6. float MLX90641_GetEmissivity(paramsMLX90641 *mlx90641);

This function returns the emissivity setting stored in the EEPROM of a MLX90641.

- paramsMLX90641 *mlx90641 – pointer to a variable of type *paramsMLX90641* in which are stored the extracted parameters for a particular device. Note that if multiple MLX90641 devices are on the line, an array of type *paramsMLX90641* could be used

Example:

1. *Get the emissivity stored in the EEPORM of a MLX90641 device – emissivity is 0.95:*

   unsigned char slaveAddress;

   static uint16_t eeMLX90641[832];

   paramsMLX90641 mlx90641;

   float emissivity;

   int status;

   MLX90641_DumpEE (slaveAddress, eeMLX90641);

   MLX90641_ExtractParameters(eeMLX90641, &mlx90641);

   emissivity = MLX90641_GetEmissivity(&mlx90641); *//emissivity = 0.95*

3.2. MLX90641 pre-processing functions

3.2.1. int MLX90641_DumpEE(uint8_t slaveAddr, uint16_t *eeData)

This function reads all the necessary EEPROM data from a MLX90641 device with a given slave address into a MCU memory location defined by the user. The allocated memory should be at least 832 words for proper operation. If the result is -1, NACK occurred during the communication and this is not a valid EEPROM data. If the result is -9, a single error has been detected and corrected in at least one EEPROM address. Note that due to the nature of Hamming code triple errors (or any error caused by odd number of bits larger than 1 being flipped) could be detected as a single error but cannot be corrected. If the result is -10, a double error has been detected in at least one EEPROM address. Note that the double errors are not corrected, therefore if one is detected that means that at least one EEPROM value is not correct.

- *uint8_t slaveAddr* – Slave address of the MLX90641 device (the default slave address is 0x33)
- *uint16_t *eeData* – pointer to the MCU memory location where the user wants the EEPROM data to be stored

Example:

1. *Dump the EEPROM of a MLX90641 device with slave address 0x33:*

   static uint16_t eeMLX90641[832];

   int status;

   status = MLX90641_DumpEE (0x33, eeMLX90641); //the whole EEPROM is stored in the eeMLX90641 array

3.2.2. int MLX90641_ExtractParameters(uint16_t * eeData, paramsMLX90641 *mlx90641)

This function extracts the parameters from a given EEPROM data array and stores values as type defined in *MLX90641_API.h.* After the parameters are extracted, the EEPROM data is not needed anymore and the memory it was stored in could be reused.

- uint16_t * eeData – pointer to the array that contains the EEPROM from which to extract the parameters
- paramsMLX90641 *mlx90641 – pointer to a variable of type *paramsMLX90641* in which to store the extracted parameters. Note that if multiple MLX90641 devices are on the line, an array of type *paramsMLX90641* could be used

Example:

2. *Extract the parameters from the EEPORM of a MLX90641 device:*

   unsigned char slaveAddress;

   static uint16_t eeMLX90641[832];

   paramsMLX90641 mlx90641;

   int status;

   MLX90641_DumpEE (slaveAddress, eeMLX90641);

   MLX90641_ExtractParameters(eeMLX90641, &mlx90641); *//The parameters are extracted from the*

   // EEPROM data stored in eeMLX90641

   //array and are stored in mlx90641 variable

   //of type paramsMLX90641

3.3. MLX90641 data acquisition functions

3.3.1. int MLX90641_SynchFrame(uint8_t slaveAddr)

This function waits for a new data to be available for a MLX90641 device with a given slave address. The purpose of the function is to synchronize with the MLX90641 device so that the data acquisition can be started right after the data is available. This would increase the time available for reading before new data is available. The function is especially useful when the time needed for data acquisition and signal processing is comparable to the refresh time of the sensor. It is recommended that the frame synchronization is being used before the very first frame read. It may be helpful for some systems to re-synchronize every once in a while. The rate of the synchronizations depends on the relation between the data processing time and the sensor refresh time. The faster the data processing time is, the slower the synchronization rate. If the result is -1, NACK occurred during the communication and the synchronization most likely failed. Note that if there is already new data available when calling the function, it will be disregarded.

- *uint8_t slaveAddr* – Slave address of the MLX90641 device (the default slave address is 0x33)

Example:

1. *Synchronize the frame data with a MLX90641 device with slave address 0x33:*

```
int status;

status = MLX90641_SynchFrame(0x33);     //if status is 0, the synchronization is successful and the
                                        //MLX90641_GetFrameData() function could be called
```

3.3.2. int MLX90641_TriggerMeasurement(uint8_t slaveAddr)

This function uses the global reset command described in the I2C standard. If the result is -2, NAK occurred during the communication, if the result is -2, memory write failed, if the result is -11, the trigger was not successful and if the result is 0 – the measurement was triggered. After the trigger the device will always first measure sub-page 0 and after that sub-page 1. Note that this function will reset all devices on the same I2C bus that support the I2C global reset command.

- *uint8_t slaveAddr* – Slave address of the MLX90641 device (the default slave address is 0x33)

Example:

1. *Trigger the measurement for a MLX90641 device with slave address 0x33:*

```
int status;

status = MLX90641_TriggerMeasurement (0x33);        //the measurement is triggered

//wait for the data for sub-page 0 to become available

//read the data for sub-page 0

//wait for the data for sub-page 1 to become available – some processing of the sub-page 0 data can be done

//read the data for sub-page 1
```

3.3.3. int MLX90641_GetFrameData(uint8_t slaveAddr, uint16_t *frameData)

This function reads all the necessary frame data from a MLX90641 device with a given slave address into a MCU memory location defined by the user. The allocated memory should be at least 242 words for proper operation. If the result is -1, NACK occurred during the communication and this is not a valid frame data, else if the result is -8, the data could not be acquired for a certain time – the most probable reason is that the I2C frequency is too low, else the result is the sub-page of the acquired data.

- *uint8_t slaveAddr* – Slave address of the MLX90641 device (the default slave address is 0x33)
- uint16_t *frameData – pointer to the MCU memory location where the user wants the frame data to be stored

Example:

2. *Dump the EEPORM of a MLX90641 device with slave address 0x33:*

```
static uint16_t mlx90641Frame[242];

int status;

status = MLX90641_GetFrameData (0x33, mlx90641Frame);      //the whole frame data  is stored in the

                                                           // mlx90641Frame array
```

3.4. MLX90641 calculation functions

3.4.1. float MLX90641_GetVdd(uint16_t *frameData, paramsMLX90641 *params)

This function returns the current Vdd from a given MLX90641 frame data and extracted parameters. The result is a float number.

- uint16_t *frameData – pointer to the MLX90641 frame data that is already acquired
- paramsMLX90641 *params – pointer to the MCU memory location where the already extracted parameters for the MLX90641 device are stored

Example:

1. *Get the Vdd of a MLX90641 device with supply voltage 3.3V:*

```
float vdd;

unsigned char slaveAddress;

static int eeMLX90641[832];

static int mlx90641Frame[242];
```

paramsMLX90641 mlx90641;

MLX90641_DumpEE (slaveAddress, eeMLX90641);

MLX90641_ExtractParameters(eeMLX90641, &mlx90641);

MLX90641_GetFrameData (0x33, mlx90641Frame);

vdd = MLX90641_GetVdd(mlx90641Frame, &mlx90641); *//vdd = 3.3*

3.4.2. *float MLX90641_GetTa(uint16_t *frameData, paramsMLX90641 *params)*

This function returns the current Ta measured in a given MLX90641 frame data and extracted parameters. The result is a float number.

- uint16_t *frameData – pointer to the MLX90641 frame data that is already acquired
- paramsMLX90641 *params – pointer to the MCU memory location where the already extracted parameters for the MLX90641 device are stored

Example:

1. *Get the Ta of a MLX90641 device that measured 27.18°C ambient temperature:*

   float Ta;

   unsigned char slaveAddress;

   static int eeMLX90641[832];

   static int mlx90641Frame[242];

   paramsMLX90641 mlx90641;

   MLX90641_DumpEE (slaveAddress, eeMLX90641);

   MLX90641_ExtractParameters(eeMLX90641, &mlx90641);

   MLX90641_GetFrameData (0x33, mlx90641Frame);

   Ta = MLX90641_GetTa (mlx90641Frame, &mlx90641); *//Ta = 27.18*

3.4.3. void MLX90641_CalculateTo(uint16_t *frameData, paramsMLX90641 *params, float emissivity, float tr, float *result)

This function calculates the object temperatures for all 192 pixels in the frame all based on the frame data read from a MLX90641 device, the extracted parameters for that particular device and the emissivity defined by the user. The allocated memory should be at least 192 words for proper operation.

- uint16_t *frameData – pointer to the MCU memory location where the user wants the frame data to be stored

- paramsMLX90641 *params – pointer to the MCU memory location where the already extracted parameters for the MLX90641 device are stored
- float emissivity – emissivity defined by the user. The emissivity is a property of the measured object
- float tr – reflected temperature defined by the user. If the object emissivity is less than 1, there might be some temperature reflected from the object. In order for this to be compensated the user should input this reflected temperature. The sensor ambient temperature could be used, but some shift depending on the enclosure might be needed. For a MLX90641 in the open air the shift is -5°C.
- *float *result* – pointer to the MCU memory location where the user wants the object temperatures data to be stored

Example:

1. *Calculate the object temperatures for all the pixels in a frame, object emissivity is 0.95 and the reflected temperature is 23.15°C (measured by the user):*

```
float emissivity = 0.95;

float tr;

unsigned char slaveAddress;

static uint16_t eeMLX90641[832];

static uint16_t mlx90641Frame[242];

paramsMLX90641 mlx90641;

static float mlx90641To[192];

MLX90641_DumpEE (slaveAddress, eeMLX90641);

MLX90641_ExtractParameters(eeMLX90641, &mlx90641);

MLX90641_GetFrameData (0x33, mlx90641Frame);

tr = 23.15;

MLX90641_CalculateTo(MLX90641Frame, &mlx90641, emissivity, tr, mlx90641To); //The object temperatures
                                                                              //for all 192 pixels in a
                                                                              //frame are stored in the
                                                                              //MLX90641To array
```

2. *Calculate the object temperatures for all the pixels in a frame, object emissivity is 0.95 and the reflected temperature is based on the sensor ambient temperature:*

```
#define TA_SHIFT 5                    //the default shift for a MLX90641 device in open air

float emissivity = 0.95;
```

float tr;

unsigned char slaveAddress;

static uint16_t eeMLX90641[832];

static uint16_t mlx90641Frame[242];

paramsMLX90641 mlx90641;

static float mlx90641To[192];

MLX90641_DumpEE (slaveAddress, eeMLX90641);

MLX90641_ExtractParameters(eeMLX90641, &mlx90641);

MLX90641_GetFrameData (0x33, mlx90641Frame);

tr = MLX90641_GetTa(mlx90641Frame, &mlx90641) – TA_SHIFT; *//reflected temperature based on the sensor*
//ambient temperature

MLX90641_CalculateTo(mlx90641Frame, &mlx90641, emissivity, tr, mlx90641To); *//The object temperatures*
//for all 192 pixels in a
//frame are stored in the
//MLX90641To array

3.4.4. void MLX90641_GetImage(uint16_t *frameData, paramsMLX90641 *params, float *result) – UNDER DEVELOPMENT

This function calculates values for all 192 pixels in the frame all based on the frame data read from a MLX90641 device and the extracted parameters for that particular device. The allocated memory should be at least 192 words for proper operation. The smaller the value, the lower the temperature in the pixels field of view. Note that these are signed values.

- uint16_t *frameData – pointer to the MCU memory location where the user wants the frame data to be stored
- paramsMLX90641 *params – pointer to the MCU memory location where the already extracted parameters for the MLX90641 device are stored
- *float *result* – pointer to the MCU memory location where the user wants the object temperatures data to be stored

Example:

1. *Get an image for a frame:*

   unsigned char slaveAddress;

   static uint16_t eeMLX90641[832];

   static uint16_t mlx90641Frame[242];

```
paramsMLX90641 mlx90641;

static float mlx90641Image[192];

MLX90641_DumpEE (slaveAddress, eeMLX90641);

MLX90641_ExtractParameters(eeMLX90641, &mlx90641);

MLX90641_GetFrameData (0x33, mlx90641Frame);

MLX90641_GetImage(mlx90641Frame, &mlx90641, emissivity, tr, mlx90641Image); //The image from the
                                                                             //frame data is extracted
                                                                             //and is stored in the
                                                                             //mlx90641Image array
```

3.4.5. void MLX90641_BadPixelsCorrection(uint16_t pixel, float *to)

This function corrects the values of the a broken pixel. The pixel that is marked as broken is already being reported in the *paramsMLX90641 brokenPixel*. Note that it is possible to choose which pixel to be corrected by calling the function with the appropriate parameters.

- *uint16_t pixel* – contains the pixel to be corrected.

  Note: If there is no broken pixel detected, the value in the *paramsMLX90641 brokenPixel* would be 0xFFFF

- *float *to* – pointer to the object temperature values array. The pixel values will be corrected by overwriting the current temperature values for the corresponding pixel index

Example:

1. *Correct the object temperature for the pixel that is marked as broken in a frame:*

```
float emissivity = 0.95;

float tr;

unsigned char slaveAddress;

static uint16_t eeMLX90641[832];

static uint16_t mlx90641Frame[242];

paramsMLX90641 mlx90641;

static float mlx90641To[192];

int status;
```

```
status = MLX90641_DumpEE (slaveAddress, eeMLX90641);

status = MLX90641_ExtractParameters(eeMLX90641, &mlx90641);

status = MLX90641_GetFrameData (0x33, mlx90641Frame);

tr = 23.15;

MLX90641_CalculateTo(MLX90641Frame, &mlx90641, emissivity, tr, mlx90641To);

MLX90641_BadPixelsCorrection(mlx90641->brokenPixel, mlx90641To);

//the corrected value is  in the mlx90641To array
```

2. Correct the object temperature for the desired pixel:

```
float emissivity = 0.95;

float tr;

unsigned char slaveAddress;

static uint16_t eeMLX90641[832];

static uint16_t mlx90641Frame[242];

paramsMLX90641 mlx90641;

static float mlx90641To[192];

uint16_t badPixel = 35;          //pixel 35 should be corrected

int status;

status = MLX90641_DumpEE (slaveAddress, eeMLX90641);

status = MLX90641_ExtractParameters(eeMLX90641, &mlx90641);

status = MLX90641_GetFrameData (0x33, mlx90641Frame);

tr = 23.15;

MLX90641_CalculateTo(mlx90641Frame, &mlx90641, emissivity, tr, mlx90641To);

MLX90641_BadPixelsCorrection(badPixel, mlx90641To);
```

4. Revision history table

05/02/2018	Initial release
07/11/2019	Reorganization for better readability
22/11/2019	Typos fixed
01/06/2020	New functions description added
24/11/2020	MLX90640_BadPixelsCorrection function description is corrected

Table 1