Contents

1. Overview

In order to use the MLX90640 driver there are 4 files that should be included in the C project:

- *MLX90640_I2C_Driver.h* – header file containing the definitions of the I2C related functions
- *MLX90640_I2C_Driver.cpp or MLX90640_SWI2C_Driver.cpp* – file containing the I2C related functions
- *MLX90640_API.h* – header file containing the definitions of the MLX90640 specific functions
- *MLX90640_API.cpp* – file containing the MLX90640 specific functions

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 MLX90640 is achieved. As the functions are being used by the MLX90640 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. *MLX90640_I2C_Driver.cpp*

This file should be included if the hardware I2C in 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. *MLX90640_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 MLX90640 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 MLX90640. Those functions might need some modifications by the user. However, it is important to keep the same function definitions.

2.3.1. void MLX90640_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

    MLX90640_I2CInit();

    …

    …MLX90640 communication

    …User code

2.3.2. void MLX90640_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 MLX90640 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 MLX90640 device is transmitting data.

Example:

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

    MLX90640_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:*

    MLX90640_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

//MLX90640 is transmitting data generated scl when

2.3.3. int MLX90640_I2CRead(uint8_t slaveAddr, uint16_t startAddress, uint16_t nMemAddressRead, uint16_t *data)

This function reads a desired number of words from a selected MLX90640 device memory starting from a given address and stores the data in the MCU memory location defined by the user. Note that the address location is being auto incremented while an I2C read communication is in progress, but if a new read is initiated the address will be reset. Thus if a large memory should be dumped in more than one I2C read command, special care should be taken so that the appropriate start address is used. 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 MLX90640 device (the default slave address is 0x33)
- *uint16_t startAddress* – First address from the MLX90640 memory to be read. The MLX90640 EEPROM is in the address range 0x2400 to 0x273F and the MLX90640 RAM is in the address range 0x0400 to 0x073F.
- *uint16_t nMemAddressRead* – Number of 16-bits words to be read from the MLX90640 memory
- *uint16_t *data* – pointer to the MCU memory location where the user wants the data to be stored

Example:

1. *Reading a single EEPROM value – MLX90640 settings for a MLX90640 device with slave address 0x33:*

   uint16_t eeValue;

   MLX90640_I2CRead (0x33, 0x240C, 1, &eeValue); *//the EEPROM 0x240C cell value is stored in eeValue*

2. *Reading a whole frame data for a MLX90640 devices with slave address 0x33:*

   static uint16_t frameData[832];

   MLX90640_I2CRead(0x33, 0x0400, 832, frameData); *//the frame data is stored in the frameData array*

2.3.4. int MLX90640_I2CWrite (uint8_t slaveAddr, uint16_t writeAddress, uint16_t data)

This function writes a 16-bit value to a desired memory address of a selected MLX90640 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 MLX90640 device (the default slave address is 0x33)

- *uint16_t* writeAddress – The MLX90640 memory address to write data to
- *uint16_t* data - Data to be written in the MLX90640 memory address

Example:

1. *Writing settings – MLX90640 settings for a MLX90640 device with slave address 0x33:*

   int status;

   status = MLX90640_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 MLX90640_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. *Reseting all devices on the bus:*

   int status;

   status = MLX90640_I2CGeneralReset();   //the MLX90640 device is reset.

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

3. MLX90640 API

This is the driver for the MLX90640 device. The user should not change this driver.

3.1. MLX90640 configuration functions

API functions that allow setting and/or getting different MLX90640 configurations.

3.1.1. int MLX90640_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 MLX90640 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 MLX90640 device (the default slave address is 0x33)
- *uint8_t* resolution – The current resolution of MLX90640
  - 0x00 – 16-bit resolution
  - 0x01 – 17-bit resolution
  - 0x02 – 18-bit resolution
  - 0x03 – 19-bit resolution

Example:

1. *Setting MLX90640 device at slave address 0x33 to work with 19-bit resolution:*

   Int status;

   status = MLX90640_SetResolution(0x33,0x03);

3.1.2. int MLX90640_GetCurResolution (uint8_t slaveAddr)

This function returns the current resolution of a MLX90640 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 MLX90640 device (the default slave address is 0x33)

Example:

1. *Getting the current resolution from a MLX90640 device at slave address 0x33 that works with 19-bit resolution, but has in the EEPROM programmed 16-bit resolution:*

int curResolution;

curResolution = MLX90640_GetCurResolution(0x33); //curResolution = 0x03(19-bit) as this is the actual

//resolution the device is working with

3.1.3. int MLX90640_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 MLX90640 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 MLX90640 device (the default slave address is 0x33)
- *uint8_t refreshRate* – The current resolution of MLX90640 device
    - 0x00 – 0.5Hz
    - 0x01 – 1Hz
    - 0x02 – 2Hz
    - 0x03 – 4Hz
    - 0x04 – 8Hz
    - 0x05 – 16Hz
    - 0x06 – 32Hz
    - 0x07 – 64Hz

Example:

1. *Setting MLX90640 device at slave address 0x33 to work with 16Hz refresh rate:*

    int status;

    status = MLX90640_SetRefreshRate *(0x33,0x05);*

3.1.4. int MLX90640_GetRefreshRate (uint8_t slaveAddr)

This function returns the current refresh rate of a MLX90640 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 MLX90640 device (the default slave address is 0x33)

Example:

1. *Getting the current refresh rate from a MLX90640 device at slave address 0x33 that works with 16Hz resolution, but has in the EEPROM programmed 0.5Hz refresh rate:*

   int curRR;

   *curRR = MLX90640_*GetRefreshRate *(0x33);*   *// curRR = 0x05(16Hz) as this is the actual*

   //refresh rate the device is working with

3.1.5. int MLX90640_GetSubPageNumber (uint16_t *frameData)

This function returns the sub-page for a selected frame data of a MLX90640 device.

- *uint16_t *frameData* – pointer to the MLX90640 frame data that is already acquired

Example:

1. *Getting the sub-page for a selected frame data of a MLX90640 device:*

   static int mlx90640Frame[834];

   int 10ubpage;

   10ubpage = MLX90640_GetSubPageNumber(mlx90640Frame);   *// 10ubpage = 1 as this is the actual*

   //sub-page number for that frame

3.1.6. int MLX90640_SetInterleavedMode (uint8_t slaveAddr)

This function sets to interleaved mode a MLX90640 device with a given slave address. Note that after power-on reset, the mode will revert back to the one 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 MLX90640 device (the default slave address is 0x33)

Example:

1. *Setting MLX90640 device at slave address 0x33 to work in interleaved mode:*

   int status;

   status = MLX90640_SetInterleavedMode (0x33);

3.1.7. int MLX90640_SetChessMode (uint8_t slaveAddr)

This function sets to chess pattern mode a MLX90640 device with a given slave address. Note that after power-on reset, the mode will revert back to the one 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 MLX90640 device (the default slave address is 0x33)

Example:

1. *Setting MLX90640 device at slave address 0x33 to work in chess pattern mode:*

   int status;

   status = MLX90640_SetChessMode (0x33);

3.1.8. int MLX90640_GetCurMode (uint8_t slaveAddr)

This function returns the working mode of a MLX90640 device.

- *uint8_t slaveAddr* – Slave address of the MLX90640 device (the default slave address is 0x33)

Example:

1. *Getting the working mode of a MLX90640 device with slave address 0x33:*

   int mode;

   mode = MLX90640_GetCurMode(0x33); // mode = 0 if interleaved mode is set

   // mode = 1 if chess pattern mode is set

3.2. MLX90640 pre-processing functions

API functions that are normally called just once after power-on reset in order to extract all the parameters required for proper temperature calculations

3.2.1. int MLX90640_DumpEE(uint8_t slaveAddr, uint16_t *eeData)

This function reads all the necessary EEPROM data from a MLX90640 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.

- *uint8_t slaveAddr* – Slave address of the MLX90640 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 MLX90640 device with slave address 0x33:*

```
static uint16_t eeMLX90640[832];

int status;

status = MLX90640_DumpEE (0x33, eeMLX90640);     //the whole EEPROM is stored in the eeMLX90640 array
```

3.2.2. int MLX90640_ExtractParameters(uint16_t * eeData, paramsMLX90640 *mlx90640)

This function extracts the parameters from a given EEPROM data array and stores values as type defined in *MLX90640_API.h.* After the parameters are extracted, the EEPROM data is not needed anymore and the memory it was stored in could be reused. If the returned value is -7, the EEPROM data at the specified location is not a valid MLX90640 EEPROM and the parameters extraction is aborted.

- *uint16_t * eeData* – pointer to the array that contains the EEPROM from which to extract the parameters
- *paramsMLX90640 *mlx90640* – pointer to a variable of type *paramsMLX90640* in which to store the extracted parameters. Note that if multiple MLX90640 devices are on the line, an array of type *paramsMLX90640* could be used

Example:

1. *Extract the parameters from the EEPROM of a MLX90640 device:*

```
unsigned char slaveAddress;

static uint16_t eeMLX90640[832];

paramsMLX90640 mlx90640;

int status;

status = MLX90640_DumpEE (slaveAddress, eeMLX90640);

status = MLX90640_ExtractParameters(eeMLX90640, &mlx90640); //The parameters are extracted from the
                                                            // EEPROM data stored in eeMLX90640
```

// array and are stored in mlx90640 variable

//of type paramsMLX90640

3.3. MLX90640 data acquisition functions

API functions that ensure proper data acquisition.

3.3.1. int MLX90640_SynchFrame(uint8_t slaveAddr)

This function waits for a new data to be available for a MLX90640 device with a given slave address. The purpose of the function is to synchronize with the MLX90640 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 those 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 MLX90640 device (the default slave address is 0x33)

Example:

1. *Synchronize the frame data with a MLX90640 device with slave address 0x33:*

   int status;

   status = MLX90640_SynchFrame(0x33); *//if status is 0, the synchronization is successful and the*
   //MLX90640_GetFrameData() function could be called

3.3.2. int MLX90640_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 -9, 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 that command.

- *uint8_t slaveAddr* – Slave address of the MLX90640 device (the default slave address is 0x33)

Example:

1. *Trigger the measurement for a MLX90640 device with slave address 0x33:*

   int status;

status = MLX90640_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 MLX90640_GetFrameData(uint8_t slaveAddr, uint16_t *frameData)

This function reads all the necessary frame data from a MLX90640 device with a given slave address into a MCU memory location defined by the user. The allocated memory should be at least 834 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 and the frame data is corrupted – 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 MLX90640 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:

1. *Read the frame data of a MLX90640 device with slave address 0x33:*

   static uint16_t mlx90640Frame[834];

   int status;

   status = MLX90640_GetFrameData (0x33, mlx90640Frame); *//the whole frame data is stored in the*

   // mlx90640Frame array

2. *Read the frame data of a MLX90640 device with slave address 0x33 using measurement triggering:*

   static uint16_t mlx90640Frame[834];

   int status;

   status = MLX90640_TriggerMeasurement (0x33);

   status = MLX90640_GetFrameData (0x33, mlx90640Frame); *//the data for sub-page 0 is fetched*

   //some waiting or data processing might be included

   status = MLX90640_GetFrameData (0x33, mlx90640Frame); *//the whole frame data is stored in the*

// mlx90640Frame array

3.4. MLX90640 calculation functions

API functions that should be called in a certain sequence in order to get properly calculated temperatures.

3.4.1. float MLX90640_GetVdd(uint16_t *frameData, const paramsMLX90640 *params)

This function returns the current Vdd from a given MLX90640 frame data and extracted parameters. The result is a float number.

- *uint16_t *frameData* – pointer to the MLX90640 frame data that is already acquired
- *paramsMLX90640 *params* – pointer to the MCU memory location where the already extracted parameters for the MLX90640 device are stored

Example:

1. *Get the Vdd of a MLX90640 device with supply voltage 3.3V:*

```
float vdd;

uint8_t slaveAddress;

static int eeMLX90640[832];

static int mlx90640Frame[834];

paramsMLX90640 mlx90640;

int status;

status = MLX90640_DumpEE (slaveAddress, eeMLX90640);

status = MLX90640_ExtractParameters(eeMLX90640, &mlx90640);

status = MLX90640_GetFrameData (0x33, mlx90640Frame);

vdd = MLX90640_GetVdd(mlx90640Frame, &mlx90640); //vdd = 3.3
```

3.4.2. *float MLX90640_GetTa(uint16_t *frameData, const paramsMLX90640 *params)*

This function returns the current Ta measured in a given MLX90640 frame data and extracted parameters. The result is a float number.

- *uint16_t *frameData* – pointer to the MLX90640 frame data that is already acquired

- *paramsMLX90640 *params* – pointer to the MCU memory location where the already extracted parameters for the MLX90640 device are stored

Example:

1. *Get the Ta of a MLX90640 device that measured 27.18°C ambient temperature:*

```
float Ta;

unsigned char slaveAddress;

static int eeMLX90640[832];

static int mlx90640Frame[834];

paramsMLX90640 mlx90640;

int status;

status = MLX90640_DumpEE (slaveAddress, eeMLX90640);

status = MLX90640_ExtractParameters(eeMLX90640, &mlx90640);

status = MLX90640_GetFrameData (0x33, mlx90640Frame);

Ta = MLX90640_GetTa (mlx90640Frame, &mlx90640); //Ta = 27.18
```

3.4.3. void MLX90640_CalculateTo(uint16_t *frameData, const paramsMLX90640 *params, float emissivity, float tr, float *result)

This function calculates the object temperatures for all 768 pixel in the frame all based on the frame data read from a MLX90640 device, the extracted parameters for that particular device and the emissivity defined by the user. The allocated memory should be at least 768 words for proper operation.

Note: If absolute temperature values are not needed, the *MLX90640_GetImage()* function could be used instead. In that case the *MLX90640_CalculateTo()* function should not be called.

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

- *paramsMLX90640 *params* – pointer to the MCU memory location where the already extracted parameters for the MLX90640 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 MLX90640 in the open air the shift is -8°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 eeMLX90640[832];

static uint16_t mlx90640Frame[834];

paramsMLX90640 mlx90640;

static float mlx90640To[768];

int status;

status = MLX90640_DumpEE (slaveAddress, eeMLX90640);

status = MLX90640_ExtractParameters(eeMLX90640, &mlx90640);

status = MLX90640_GetFrameData (0x33, mlx90640Frame);

tr = 23.15;

MLX90640_CalculateTo(mlx90640Frame, &mlx90640, emissivity, tr, mlx90640To);//The object temperatures
                                                                            //for all 768 pixels in a
                                                                            //frame are stored in the
                                                                            //mlx90640To array
```

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

```
#define TA_SHIFT 8                     //the default shift for a MLX90640 device in open air

float emissivity = 0.95;

float tr;

unsigned char slaveAddress;

static uint16_t eeMLX90640[832];
```

```
static uint16_t mlx90640Frame[834];

paramsMLX90640 mlx90640;

static float mlx90640To[768];

int status;

status = MLX90640_DumpEE (slaveAddress, eeMLX90640);

status = MLX90640_ExtractParameters(eeMLX90640, &mlx90640);

status = MLX90640_GetFrameData (0x33, mlx90640Frame);

tr = MLX90640_GetTa(mlx90640Frame, &mlx90640) – TA_SHIFT; //reflected temperature based on the sensor
                                                          //ambient temperature

MLX90640_CalculateTo(mlx90640Frame, &mlx90640, emissivity, tr, mlx90640To);//The object temperatures
                                                                          //for all 768 pixels in a
                                                                          //frame are stored in the
                                                                          //mlx90640To array
```

3.4.4. void MLX90640_GetImage(uint16_t *frameData, const paramsMLX90640 *params, float *result)

This function calculates values for all 768 pixels in the frame all based on the frame data read from a MLX90640 device and the extracted parameters for that particular device. The allocated memory should be at least 768 words for proper operation. The smaller the value, the lower the temperature in the pixels field of view. Note that these are signed values.

Note: If absolute temperature values are needed, the *MLX90640_CalculateTo()* should be used instead. In that case the function *MLX90640_GetImage()* should not be called.

- *uint16_t *frameData* – pointer to the MCU memory location where the user wants the frame data to be stored
- *paramsMLX90640 *params* – pointer to the MCU memory location where the already extracted parameters for the MLX90640 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 eeMLX90640[832];

static uint16_t mlx90640Frame[834];

paramsMLX90640 mlx90640;

static float mlx90640Image[768];

int status;

status = MLX90640_DumpEE (slaveAddress, eeMLX90640);

status = MLX90640_ExtractParameters(eeMLX90640, &mlx90640);

status = MLX90640_GetFrameData (0x33, mlx90640Frame);

MLX90640_GetImage(mlx90640Frame, &mlx90640, mlx90640Image);    //The image from the
                                                               //frame data is extracted
                                                               //and is stored in the
                                                               //mlx90640Image array
```

3.5. MLX90640 supporting image processing functions

API functions that could be used for image processing.

3.5.1. void MLX90640_BadPixelsCorrection(uint16_t *pixels, float *to, int mode, paramsMLX90640 *params)

This function corrects the values of the broken pixels and/or the outlier pixels. The values of all pixels indexes in the *pixels* array (until value 0xFFFF is read) will be corrected using different filtering methods. The pixels that are marked as broken or outliers are already being reported in the *paramsMLX90640* arrays *brokenPixels* and *outlierPixels*. Note that it is possible to choose which pixels to be corrected by creating a custom list with the indexes of those pixels. The list should end with 0xFFFF.

- *uint16_t *pixels* – pointer to the array that contains the pixels to be corrected.

  Note: the array should contain the indexes of the pixels (0 to 767) and 0xFFFF should be used as a termination value

- *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
- *int mode* – the current working mode of the MLX90640 device
  - 0 – interleaved pattern mode

- 1 – chess pattern mode

- *paramsMLX90640 *params* – pointer to the MCU memory location where the already extracted parameters for the MLX90640 device are stored

Example:

1. *Correct the object temperatures for all the pixels that are marked as broken or outliers in a frame:*

```
float emissivity = 0.95;

float tr;

unsigned char slaveAddress;

static uint16_t eeMLX90640[832];

static uint16_t mlx90640Frame[834];

paramsMLX90640 mlx90640;

static float mlx90640To[768];

int status;

status = MLX90640_DumpEE (slaveAddress, eeMLX90640);

status = MLX90640_ExtractParameters(eeMLX90640, &mlx90640);

mode = MLX90640_GetCurMode(slaveAddress);

status = MLX90640_GetFrameData (0x33, mlx90640Frame);

tr = 23.15;

MLX90640_CalculateTo(mlx90640Frame, &mlx90640, emissivity, tr, mlx90640To);

MLX90640_BadPixelsCorrection((&mlx90640)->brokenPixels, mlx90640To, mode, &mlx90640);

MLX90640_BadPixelsCorrection((&mlx90640)->outlierPixels, mlx90640To, mode, &mlx90640);


//the corrected values are in the mlx90640To array
```

2. *Correct the object temperatures for all the desired pixels by using a custom list:*

```
float emissivity = 0.95;

float tr;
```

```
unsigned char slaveAddress;

static uint16_t eeMLX90640[832];

static uint16_t mlx90640Frame[834];

paramsMLX90640 mlx90640;

static float mlx90640To[768];

static uint16_t badPixels[5] = {0, 35, 150, 500, 0xFFFF};        //create a custom list of pixels to correct

int status;

status = MLX90640_DumpEE (slaveAddress, eeMLX90640);

status = MLX90640_ExtractParameters(eeMLX90640, &mlx90640);

mode = MLX90640_GetCurMode(slaveAddress);

status = MLX90640_GetFrameData (0x33, mlx90640Frame);

tr = 23.15;

MLX90640_CalculateTo(mlx90640Frame, &mlx90640, emissivity, tr, mlx90640To);

MLX90640_BadPixelsCorrection(badPixels, mlx90640To, mode, &mlx90640);


//the corrected values are in the mlx90640To array
```

4. Revision history table

02/03/2018	Initial release
04/06/2018	Typos fixed
05/06/2018	Typos fixed I2C read and write functions definitions changed
07/06/2018	Added a note for I2C communication when limited number of addresses can be read at a time
19/06/2018	Added const modifier to paramsMLX90640 in some functions
24/08/2018	Typos fixed
29/10/2018	Added function for correction of broken pixels and outliers
12/06/2019	Reorganization for better readability
07/11/2019	Typos fixed
22/11/2019	Typos fixed
10/04/2020	New functions description added

Table 1