Contents

1. Overview

In order to use the MLX90621 driver there are 4 files that should be included in the C project:

- *MLX90621_I2C_Driver.h* – header file containing the definitions of the I2C related functions
- *MLX90621_I2C_Driver.cpp or MLX90621_SWI2C_Driver.cpp* – file containing the I2C related functions
- *MLX90621_API.h* – header file containing the definitions of the MLX90621 specific functions
- *MLX90621_API.h* – file containing the MLX90621 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 MLX90621 is achieved. As the functions are being used by the MLX90621 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. *MLX90621_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. *MLX90621_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 named '*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 MLX90621 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 five main functions that ensure the proper communication between the user MCU and the MLX90621. Those functions might need some modifications by the user. However, it is important to keep the same function definitions.

2.3.1. void MLX90621_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

   MLX90621_I2CInit();

   …

   …MLX90621 communication

   …User code

2.3.2. void MLX90621_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 MLX90621 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 MLX90621 device is transmitting data.

Example:

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

   MLX90621_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:*

```
MLX90621_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
                            //MLX90621 is transmitting data generated scl when
```

2.3.3. MLX90621_I2CReadEEPROM(uint8_t slaveAddr, uint8_t startAddress, uint16_t nMemAddressRead, uint8_t *data);

This function reads a desired number of bytes from a selected MLX90621 device EEPROM 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 MLX90621 device – always 0x50
- *uint8_t startAddress* – First address from the MLX90621 EEPROM to be read
- *uint16_t nMemAddressRead* – Number of bytes to be read from the MLX90621 EEPROM
- *uint8_t *data* – pointer to the MCU memory location where the user wants the data to be stored

Example:

1. *Reading the whole EEPROM of a MLX90621 device:*

```
static uint8_t eeData[256];

status = MLX90621_I2CReadEEPROM(0x50, 0, 256, eeData);    //the EEPROM data is stored in the eeData
                                                          //array
```

2.3.4. int MLX90621_I2CRead(uint8_t slaveAddr, uint8_t command, uint8_t startAddress, uint8_t addressStep, uint8_t nMemAddressRead, uint16_t *data)

This function reads a desired number of words from a selected MLX90621 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 MLX90621 device – always 0x60

- *uint8_t command* – I2C command for the MLX90621 device – refer to the MLX90621 datasheet for more details
- *uint8_t startAddress* – First address from the MLX90621 memory to be read. The IR data is stored in address 0x00 to 0x3F; the PTAT data is stored at address 0x40 and the compensation pixel data is at address 0x41
- *uint8_t addressStep* – Read step – useful when a certain read pattern is desired (read a line, a column or some other pattern)
- *uint8_t nMemAddressRead* – Number of 16-bits words to be read from the MLX90621 memory
- *uint16_t *data* – pointer to the MCU memory location where the user wants the data to be stored

Example:

2. *Reading the configuration register value:*

```
uint16_t cfgReg;

status = MLX90621_I2CRead(0x60, 0x02, 0x92, 0, 1, &cfgReg);     //the configuration register value is stored
                                                                //in cfgReg
```

3. *Reading a whole frame data for a MLX90621:*

```
static uint16_t frameData[66];

status = MLX90621_I2CRead(0x60, 0x02, 0, 1, 66, frameData);     //the frame data is stored in the

                                                                //frameData array
```

2.3.5. int MLX90621_I2CWrite(uint8_t slaveAddr, uint8_t command, uint8_t checkValue, uint16_t data);

This function writes a 16-bit value to a desired register (configuration or trim) of a selected MLX90621 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 MLX90621 device – always 0x60
- *uint8_t command* – command for the MLX90621 register to write data to
- *uint8_t checkValue* – 0xAA when writing to the trim register and 0x55 when writing to the configuration registes
- *uint16_t* data - Data to be written in the MLX90621 register

Example:

1. *Writing settings – MLX90621 settings for a MLX90621 device:*

int status;

uint16_t value;

status = MLX90621_I2CWrite(0x60, 0x03, 0x55, value); //the desired settings are written to the //configuration register

Variable status is 0 if the write was successful.

3. MLX90621 API

This is the driver for the MLX90621 device. The user should not change this driver.

3.1. MLX90621 API functions

3.1.1. int MLX90621_GetOscillatorTrim(uint16_t *oscTrim)

This function returns the current oscillator trim value of a MLX90621 device. Note that the current oscillator trim value may 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.

- *uint16_t * oscTrim* – pointer to the memory location where the oscillator trim value will be stored

Example:

1. *Getting the current oscillator trim value from a MLX90621 device:*

   int osc;

   *status = MLX90621_*GetOscillatorTrim *(&osc); // the oscillator trim value is stored at osc*

3.1.2. int MLX90621_GetConfiguration(uint16_t *cfgReg)

This function returns the current configuration value of a MLX90621 device. Note that the current configuration value may 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.

- *uint16_t * cfgReg* – pointer to the memory location where the configuration value will be stored

Example:

1. *Getting the current configuration value from a MLX90621 device:*

```
int cfg;

status = MLX90621_GetConfiguration (&cfg);          // the configuration value is stored at cfg
```

3.1.3. int MLX90621_Configure(uint8_t * eeData)

This function extracts the oscillator trim value and the configuration value from the EEPROM data and writes these values into the appropriate registers. 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 * eeData* – pointer to the array that contains the EEPROM from which to extract the oscillator trim and the configuration values

Example:

1. *Configure a MLX90621 device:*

```
static uint8_t eeMLX90621[256];

int status;

status = MLX90621_DumpEE (eeMLX90621);

status = MLX90621_Configure(eeMlx90621);          //The MLX90621 device oscillator trim and
                                                  // configuration values are written into the
                                                  // appropriate registers
```

3.1.4. int MLX90621_SetResolution(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 MLX90621 device. 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* resolution – The current resolution of MLX90621
    - 0x00 – 15-bit resolution
    - 0x01 – 16-bit resolution
    - 0x02 – 17-bit resolution

- 0x03 – 18-bit resolution

Example:

1. *Setting MLX90621 device to work with 18-bit resolution:*

```
Int status;

status = MLX90621_SetResolution(0x03);
```

3.1.5. int MLX90621_GetCurResolution ()

This function returns the current resolution of a MLX90621 device. 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.

Example:

1. *Getting the current resolution from a MLX90621 device that works with 18-bit resolution, but has in the EEPROM programmed 16-bit resolution:*

```
int curResolution;

curResolution = MLX90621_GetResolution();      //curResolution = 0x03(18-bit) as this is the actual

                                               //resolution the device is working with
```

3.1.6. int MLX90621_SetRefreshRate (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 MLX90621 device. 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 refreshRate* – The current resolution of MLX90621 device
  - 0x00 to 0x05 – 512Hz
  - 0x06 – 256Hz
  - 0x07 – 128Hz
  - 0x08 – 64Hz
  - 0x09 – 32Hz
  - 0x0A – 16Hz
  - 0x0B – 8Hz

- 0x0C – 4Hz
- 0x0D – 2Hz
- 0x0E – 1Hz (default)
- 0x0F – 0.5Hz

Example:

1. *Setting MLX90621 device to work with 16Hz refresh rate:*

```
int status;

status = MLX90621_SetRefreshRate (0x0A);
```

3.1.7. int MLX90621_GetRefreshRate ()

This function returns the current refresh rate of a MLX90621 device. 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.

Example:

1. *Getting the current refresh rate from a MLX90621 device that works with 16Hz resolution, but has in the EEPROM programmed 0.5Hz refresh rate:*

```
int curRR;

curRR = MLX90621_GetRefreshRate ();          // curRR = 0x0A(16Hz) as this is the actual

                                             //refresh rate the device is working with
```

3.1.8. int MLX90621_DumpEE(uint8_t *eeData)

This function reads all the necessary EEPROM data from a MLX90621 device into a MCU memory location defined by the user. The allocated memory should be at least 256 bytes for proper operation. If the result is -1, NACK occurred during the communication and this is not a valid EEPROM data.

- *Uint8_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 MLX90621 device:*

```
static uint8_t eeMLX90621[256];

int status;
```

status = MLX90621_DumpEE (eeMLX90621); //the whole EEPROM is stored in the eeMLX90621 array

3.1.9. int MLX90621_ExtractParameters(uint8_t *eeData, paramsMLX90621 *mlx90621)

This function extracts the parameters from a given EEPROM data array and stores values as type defined in *MLX90621_API.h.* After the parameters are extracted, the EEPROM data is not needed anymore and the memory it was stored in could be reused.

- *uint8_t * eeData* – pointer to the array that contains the EEPROM from which to extract the parameters
- *paramsMLX90621 *mlx90621* – pointer to a variable of type *paramsMLX90621* in which to store the extracted parameters. Note that if multiple MLX90621 devices are on the line, an array of type *paramsMLX90621* could be used

Example:

1. *Extract the parameters from the EEPROM of a MLX90621 device:*

   static uint8_t eeMLX90621[256];

   paramsMLX90621 mlx90621;

   int status;

   status = MLX90621_DumpEE (eeMLX90621);

   status = ***MLX90621_ExtractParameters(eeMLX90621, &mlx90621);*** *//The parameters are extracted from the*

   // EEPROM data stored in eeMLX90621

   // array and are stored in mlx90621 variable

   //of type paramsMLX90621

3.1.10. int MLX90621_GetFrameData(uint16_t *frameData)

This function reads all the necessary frame data from a MLX90621 into a MCU memory location defined by the user. The allocated memory should be at least 66 words for proper operation. If the result is -1, NACK occurred during the communication and this is not a valid EEPROM data.

- 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 MLX90621 device:*

   static uint16_t mlx90621Frame[66];

   int status;

```
status = MLX90621_GetFrameData (mlx90621Frame);        //the whole frame data  is stored in the
                                                       // mlx90621Frame array
```

3.1.11. *float MLX90621_GetTa(uint16_t *frameData, const paramsMLX90621 *params)*

This function returns the current Ta measured in a given MLX90621 frame data and extracted parameters. The result is a float number.

- *uint16_t *frameData* – pointer to the MLX90621 frame data that is already acquired
- *paramsMLX90621 *params* – pointer to the MCU memory location where the already extracted parameters for the MLX90621 device are stored

Example:

1. *Get the Ta of a MLX90621 device that measured 27.18°C ambient temperature:*

```
float Ta;

static int eeMLX90621[256];

static int mlx90621Frame[66];

paramsMLX90621 mlx90621;

int status;

status = MLX90621_DumpEE (eeMLX90621);

status = MLX90621_ExtractParameters(eeMLX90621, &mlx90621);

status = MLX90621_GetFrameData (mlx90621Frame);

Ta = MLX90621_GetTa (mlx90621Frame, &mlx90621);        //Ta = 27.18
```

3.1.12. void MLX90621_CalculateTo(uint16_t *frameData, const paramsMLX90621 *params, float emissivity, float tr, float *result)

This function calculates the object temperatures for all 64 pixels in the frame all based on the frame data read from a MLX90621 device, the extracted parameters for that particular device and the emissivity defined by the user. The allocated memory should be at least 64 words for proper operation.

- *uint16_t *frameData* – pointer to the MCU memory location where the frame data is stored
- *paramsMLX90621 *params* – pointer to the MCU memory location where the already extracted parameters for the MLX90621 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 MLX90621 in the open air the shift is 0°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;

static uint8_t eeMLX90621[256];

static uint16_t mlx90621Frame[66];

paramsMLX90621 mlx90621;

static float mlx90621To[64];

int status;

status = MLX90621_DumpEE (eeMLX90621);

status = MLX90621_ExtractParameters(eeMLX90621, &mlx90621);

status = MLX90621_GetFrameData (mlx90621Frame);

tr = 23.15;

MLX90621_CalculateTo(mlx90621Frame, &mlx90621, emissivity, tr, mlx90621To);//The object temperatures
                                                                            //for all 64 pixels in a
                                                                            //frame are stored in the
                                                                            //mlx90621To array
```

3.1.13. void MLX90621_GetImage(uint16_t *frameData, const paramsMLX90621 *params, float *result)

This function calculates values for all 64 pixels in the frame all based on the frame data read from a MLX90621 device and the extracted parameters for that particular device. The allocated memory should be at least 64 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 frame data is stored

- *paramsMLX90621 *params* – pointer to the MCU memory location where the already extracted parameters for the MLX90621 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:*

```
static uint8_t eeMLX90621[256];

static uint16_t mlx90621Frame[66];

paramsMLX90621 mlx90621;

static float mlx90621Image[64];

int status;

status = MLX90621_DumpEE (eeMLX90621);

status = MLX90621_ExtractParameters(eeMLX90621, &mlx90621);

status = MLX90621_GetFrameData (mlx90621Frame);

MLX90621_GetImage(mlx90621Frame, &mlx90621, mlx90621Image);     //The image from the
                                                                //frame data is extracted
                                                                //and is stored in the
                                                                //mlx90621Image array
```

4. Revision history table

23/05/2019	Initial release

Table 1