MLX90640 driver library

Guide

Visibility and dissemination of the document:

Can be widely distributed

Language: **C**
Language version: **C99**
Endianness: **Little endian only**

Synchronous: **yes**
Asynchronous: **yes**
OS: **need adaptation**

MCU compatibility:
no limit except endianness

Tests: **partial, ~80% coverage**

MISRA C: **to be determined**
CERT C: **to be determined**

PBIT: **yes**
CBIT: **possible**

MLX90640 driver library

Version: 1.1.0 date: 19 December 2021

This library is compatible with MLX90640 components.
The MLX90640 component is a far infrared thermal sensor array (32x24 pix).

Established by	Reviewed	Approved
Name: FMA	Name: FMA	Name: FMA
Date: 19 December 2021	Date: 19 December 2021	Date: 19 December 2021

Change history

Issue	Date	Nature / comment	Paragraph	Writer
1.1.0	19 december 2021	I2C interface rework for I2C DMA use and polling	8, 14	
1.0.2	18 April 2021	**Initial release**	-	FMA

Content

1. Introduction 5
   1.1. Purpose 5
   1.2. Documents, References, and abbreviations 5
      1.2.1. Applicable documents 5
      1.2.2. Reference documents 5
      1.2.3. Abbreviations and Acronyms 5
2. Features 6
3. Limitations 6
4. Presentation 6
   4.1. Setup 6
5. Driver configuration 6
   5.1. MLX90640_PRECALCULATE_PIXELS_COEFFS define 6
   5.2. MLX90640_MOVING_AVERAGE_FILTER_VALUES_COUNT define 7
   5.3. CHECK_NULL_PARAM define 7
6. Parameters extraction 7
7. Using the device without DMA (Synchronized) 8
   7.1. Device configuration 8
      7.1.1. Example 8
   7.2. Get a frame 9
      7.2.1. Example 9
8. Using the device with DMA (Asynchronized) 10
   8.1. Device configuration 10
      8.1.1. Example 10
   8.2. Get a frame 11
      8.2.1. Example 11
9. Summary of Driver Functions 12
   9.1. Initialization and availability 12
   9.2. Read from EEPROM, RAM and registers 12
   9.3. Write to registers 12
   9.4. Device configuration 12
   9.5. Device's parameters 13
   9.6. Calculus and processing 13
   9.7. Polling (asynchronous) 13
10. Configuration structures 14
   10.1. Device object structure 14
      10.1.1. Data fields 14
      10.1.2. TMLX90640DriverInternal type and InternalConfig variable 15
      10.1.3. Driver interface handle functions 16
   10.2. MLX90640 configuration object structure 17
      10.2.1. Data fields 17
      10.2.2. Enumerators 18
   10.3. Function's return error enumerator 19
11. Driver's functions 20
   11.1. Initialization and availability 20
      11.1.1. Enumerators 21
   11.2. Read from EEPROM, RAM and registers 22
      11.2.1. Structures 23
   11.3. Write to registers 24
   11.4. Device configuration 25
   11.5. Device's parameters 26

11.6. Calculus and processing 27
11.7. Polling (asynchronous) 27
11.7.1. FrameTo structure **28**
12. Example of "Conf_MLX90640.h" file 29
13. Example of driver interface handle functions (without DMA) 30
14. Chronogram of DMA transfer (Asynchronous) 33

Figure summary

Figure 1 - Driver use without DMA overview 8
Figure 2 - Driver use with DMA overview 10

Table summary

Aucune entrée de table d'illustration n'a été trouvée.

1. INTRODUCTION

1.1. Purpose

The purpose of this document is to explain how the driver library works and how to use it. It can work with either MLX90640BAA devices or MLX90640BAB devices or both.

The driver features are:

- Can be use with any MCU (little or big endian)
- Only take care of the controller, not the communication with it
- All functions and functionalities are implemented
- Configuration is very simplified
- Can communicate with virtually an infinite count of devices (max 126 devices per I2C port)
- Different configurations can be used with different devices (no duplication of the driver needed)
- Direct communication with the devices, the driver has no buffer
- Can use the driver defines, enums, structs to create your own functions

1.2. Documents, References, and abbreviations

1.2.1. Applicable documents

IDENTIFICATION	TITLE	DATE
I2C_Interface	This I2C interface definitions for all the https://github.com/Emandhal drivers and developments	Oct 2021

1.2.2. Reference documents

IDENTIFICATION	TITLE	DATE
3901090640	MLX90640 Datasheet Rev 12 – 32x24 IR array	Dec 2019

1.2.3. Abbreviations and Acronyms

This is the list of all the abbreviations and acronyms used in this document and their definitions. They are arranged in alphabetical order.

CBIT	Continuous Built-In Test
CERT	Computer Emergency Response Team
CLK	Clock
DMA	Direct Access Memory
FIFO	First In First Out
I2C	Inter-Integrated Circuit
MCU	Micro-Controller Unit
MISRA	Motor Industry Software Reliability Association
OS	Operating System
PBIT	Power Up Built-In Test
PIO	Programmable Input/Output
RAM	Random Access Memory

2. FEATURES

This driver has been designed to:

- Be fully configurable (all known features of the MLX90640 are managed)
- Detect which one of the MLX90640BAA or MLX90640BAB is connected
- Have no limit of configuration except the ones imposed by the device
- Manage devices completely independently
- Prevent all configuration errors
- Can be used with EEPROM data already extracted and saved in flash (reduce the use of RAM used)
- Can be used with Parameters already calculated and saved in flash (reduce the use of RAM used)
- Can be used with a DMA

3. LIMITATIONS

To use this driver and device, you need:

- At least 20k of RAM. The driver itself needs 1664 bytes for EEPROM, 106+10704 bytes for parameters, 1666 bytes for the data frame, and 3096 bytes for the frame result
- A FPU on the CPU else the driver will be very slow and only slow refresh speed can be achieved

4. PRESENTATION

This driver only takes care of configuration and check of the internal registers and the formatting of the communication with the device. That means it does not directly take care of the physical communication, there are functions interfaces to do that.

Each driver's functions need a device structure that indicate with which device he must threat and communicate. Each device can have its own configuration.

4.1. Setup

To set up one or more devices in the project, you must:

- Configure the driver with "Conf_MLX90640.h" file which will be the same for all devices but modify only its behavior in the project
- Create and define the configuration of as many device structures as there are devices to use
- Declare EEPROM (MLX90640_EEPROM) memory (used to fill the parameters of a device, after it can be discarded)
- Declare Parameters (MLX90640_Parameters) memory (one per devices)

If driver used without DMA (synchronous):

- Declare Frame (MLX90640_FrameData) data memory (used to get frame data from device and calculate the frame result)
- Declare the Frame T_o (MLX90640_FrameTo) result (one per devices, will be updated at each valid frame data)

If driver used with DMA (asynchronous):

- Declare Frame polling (MLX90640_FramePolling) data memory (used to manage the DMA transfer and the frame T_o processing)

5. DRIVER CONFIGURATION

The configuration is done by use of "Conf_MLX90640.h" file.

This file contains the some defines:

- `#define MLX90640_PRECALCULATE_PIXELS_COEFFS`
- `#define MLX90640_MOVING_AVERAGE_FILTER_VALUES_COUNT (…)`
- `#define CHECK_NULL_PARAM`

See example in §12.

5.1. MLX90640_PRECALCULATE_PIXELS_COEFFS define

This define specify to the driver to pre-calculate Offset, Sensitivity, Kta, and Kv of each pixel and save them in the MLX90640_Parameters.

If the following define is set, the driver will take in addition $(768 \times 2) + \big((768 \times 4) \times 3\big) = 10704\ bytes$ of RAM to store theses values. If unset then the driver will take less RAM but more time to calculate T_o of each pixels.

5.2. MLX90640_MOVING_AVERAGE_FILTER_VALUES_COUNT define

In order to limit the noise in the final T_o calculation it is advisable to filter the CP readings at this point of calculation. This filter is only useful with device with thermal gradient compensation. A good practice would be to apply a Moving Average Filter with length of 16 or higher. If set to < 2, the filter will be disabled.

5.3. CHECK_NULL_PARAM define

This define enables check of pointing parameters are not NULL. It checks if function parameters point to something.

This define can be enabled only in debug. Normally in a static pure C programming, these parameters and function pointer are set and fix, so always checking them is not useful.

6. PARAMETERS EXTRACTION

To work, the driver needs parameters extraction which consists on pre-calculated values from the device EEPROM dump. Using theses pre-calculated values speeds up the frame calculation. To extract the parameters you need to use the `MLX90640_ExtractDeviceParameters()` function right after the `Init_MLX90640()` function with the dumpEEPROM set to 'true' or use the `MLX90640_ExtractDeviceParameters()` function with an EEPROM dump already done with the dumpEEPROM set to 'false'.

7. USING THE DEVICE WITHOUT DMA (SYNCHRONIZED)

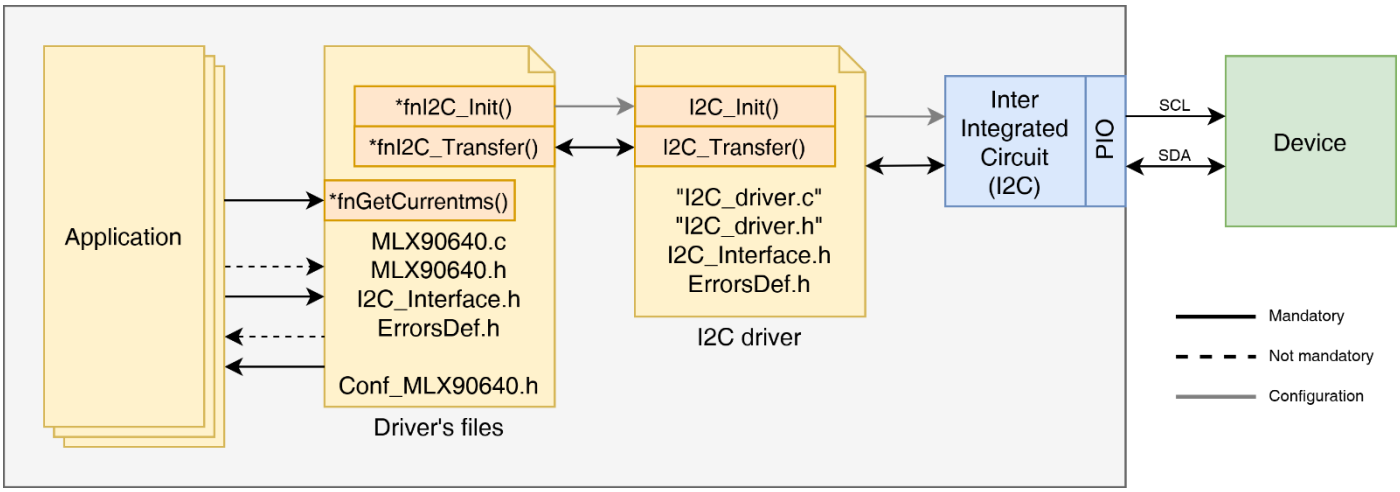


Figure 1 - Driver use without DMA overview

7.1. Device configuration

Each first parameter of a function is the device configuration. The purpose of this device configuration is to specify to the driver how and by which interface to communicate with the device selected. The device configuration type is MLX90640.

The MLX90640.fnI2C_Init, MLX90640.fnI2C_Transfer, and MLX90640.fnGetCurrentms are explained at §0 and an example at §13.

The MLX90640.I2Caddress is the I2C address of the device needed to access the device.

The MLX90640.I2CclockSpeed is the desired frequency of the I2C clock in Hertz. This allows the driver to change the I2C speed and return to the specify clock by its own when necessary. The maximum clock speed is 1MHz.

The MLX90640.UserDriverData is a generic pointer to what the user need. It can be used as context or bringing information for the interface functions for example. This variable is never touch by the driver.

The configuration of the device is only used at initialization by the Init_MLX90640() function. The configuration type is MLX90640_Config. It configures the subpage configuration and the I2C configuration.

The configuration can be change later by using Init_MLX90640() function (if you need to reconfigure the entire device) or by using MLX90640_ConfigureDeviceI2C() and/or MLX90640_ConfigureDevice() functions.

7.1.1. Example

Example of driver configuration in a .c file:

```
MLX90640_EEPROM IrCAM_EEPROM;       // Can be reused for all devices
MLX90640_Parameters IrCAM_Params;   // One per device
MLX90640_FrameData IrCAM_FrameData; // One per device
MLX90640_FrameTo IrFrame;           // One per device


struct MLX90640 MLX90640_V71 =
{
  .UserDriverData  = NULL,
  //--- Interface driver params and call functions ---
  .I2Caddress      = MLX90640_CHIPADDRESS_DEFAULT,
  .InterfaceDevice = I2C0,
  .I2CclockSpeed   = BOARD_I2C_CLK_SPEED_HZ,
  .fnI2C_Init      = TCA9543A_I2CInit,
  .fnI2C_Transfer  = TCA9543A_I2CTranfert,
  //--- Time call function ---
  .fnGetCurrentms  = GetCurrentms_V71, // Can be NULL if MLX90640_ChangeI2Caddress() will not be used
  //--- Device EEPROM ---
#if !defined(MLX90640_PRECALCULATE_PIXELS_COEFFS)
  .EEPROM          = &IrCAM_EEPROM,
#endif
  //--- Device parameters ---
  .Params          = &IrCAM_Params,
};
```

```
//---------------------------------------------------------------------------
MLX90640_Config IrCAM_Config =
{
  //--- Subpage configuration ---
  .SubpageMode    = MLX90640_MEASURE_ALTERNATE_SUBPAGES,
  .RefreshRate    = MLX90640_IR_REFRESH_RATE_16Hz,
  .ReadingPattern = MLX90640_READING_CHESS_PATTERN_MODE,
  .ADCresolution  = MLX90640_ADC_RESOLUTION_18bits,
  //--- I2C configuration ---
  .I2C_FMpEnable            = false,
  .SetThresholdTo1V8        = false,
  .SetSDAdriverCurrentLimit = false,
};
```

Example of driver configuration in a .h file:

```
extern MLX90640_EEPROM IrCAM_EEPROM;       // Can be reused for all devices
extern MLX90640_Parameters IrCAM_Params;   // One per device
extern MLX90640_FrameData IrCAM_FrameData; // One per device
extern MLX90640_FrameTo IrFrame;           // One per device

extern struct MLX90640 MLX90640_V71;
#define IrCAM  &MLX90640_V71


//---------------------------------------------------------------------------
extern MLX90640_Config IrCAM_Config;
```

7.2. Get a frame

You first need to ask the device if there is a frame available by calling the `MLX90640_IsFrameAvailable()` function.

If 'true', call the `MLX90640_GetFrameData()` to extract the subframe data from its RAM. After you need to calculate the temperature object (T_o) of each pixel of the subframe and store it into a `MLX90640_FrameTo` structure which contains all new pixels data of each subpages, this is done by calling the `MLX90640_CalculateTo()` function. This function will only modify the pixels of the current subpage.

After you need to correct the bad pixels by calculating the defective pixel value by averaging its neighboring pixels, this is done by calling the `MLX90640_CorrectBadPixels()` function.

After you can exploit the `MLX90640_FrameTo` variable and do what you want.

7.2.1. Example

Example of frame get and extract:

```
//--- Frame available ? ---
if (MLX90640_IsFrameAvailable(IrCAM))
{
  do
  {
    //--- Get subframe frame data ---
    Error = MLX90640_GetFrameData(IrCAM, &IrCAM_FrameData);
    if (Error != ERR_OK) break;

    //--- Calculate the subframe To
    Error = MLX90640_CalculateTo(IrCAM, &IrCAM_FrameData, 1.0f, Tr, &IrFrame);
    if (Error != ERR_OK) break;
    Error = MLX90640_CorrectBadPixels(IrCAM, &IrFrame);
    if (Error != ERR_OK) break;


    //--- Do what you want with the frame ---
    Tr = IrFrame.Ta - 8.0f; // Calculate next Tr: Tr = Ta - 8°
    // Do stuff


  } while (Error != ERR_OK);
  if (Error != ERR_OK) ShowError(Error);
}
```

8. USING THE DEVICE WITH DMA (ASYNCHRONIZED)

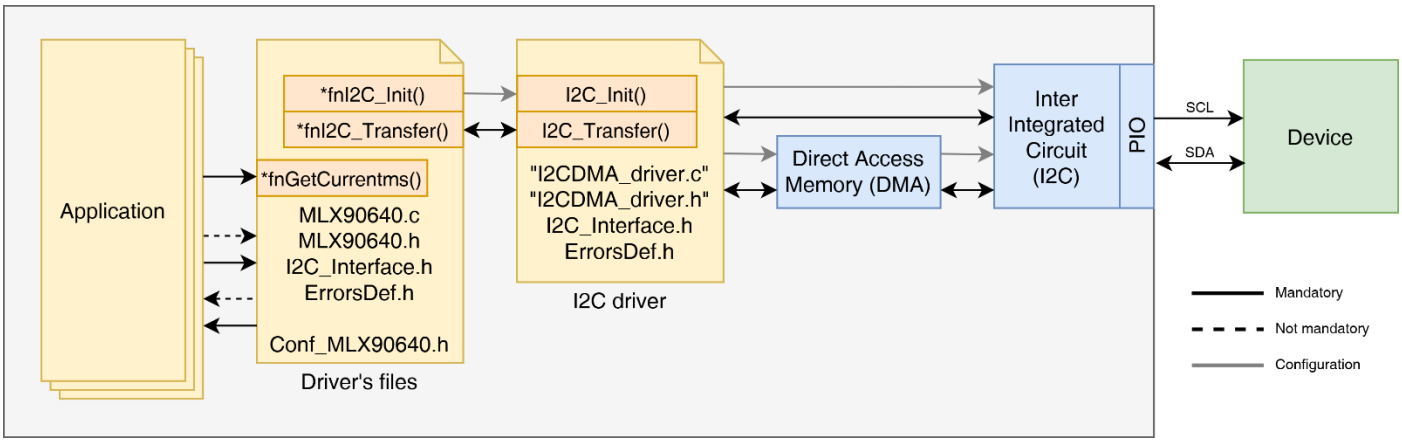


Figure 2 - Driver use with DMA overview

8.1. Device configuration

Each first parameter of a function is the device configuration. The purpose of this device configuration is to specify to the driver how and by which interface to communicate with the device selected. The device configuration type is MLX90640.

The MLX90640.fnI2C_Init, MLX90640.fnI2C_Transfer, and MLX90640.fnGetCurrentms are explained at §0 and an example at §13.

The MLX90640.I2Caddress is the I2C address of the device needed to access the device.

The MLX90640.I2CclockSpeed is the desired frequency of the I2C clock in Hertz. This allows the driver to change the I2C speed and return to the specify clock by its own when necessary. The maximum clock speed is 1MHz.

The MLX90640.UserDriverData is a generic pointer to what the user need. It can be used as context or bringing information for the interface functions for example. This variable is never touch by the driver.

The configuration of the device is only used at initialization by the Init_MLX90640() function. The configuration type is MLX90640_Config. It configures the subpage configuration and the I2C configuration.

The configuration can be change later by using Init_MLX90640() function (if you need to reconfigure the entire device) or by using MLX90640_ConfigureDeviceI2C() and/or MLX90640_ConfigureDevice() functions.

8.1.1. Example

Example of driver configuration in a .c file:

```
MLX90640_EEPROM IrCAM_EEPROM;         // Can be reused for all devices
MLX90640_Parameters IrCAM_Params;     // One per device
MLX90640_FramePolling IrFramePolling; // One per device

struct MLX90640 MLX90640_V71 =
{
  .UserDriverData  = NULL,
  //--- Interface driver params and call functions ---
  .I2Caddress      = MLX90640_CHIPADDRESS_DEFAULT,
  .InterfaceDevice = I2C0,
  .I2CclockSpeed   = BOARD_I2C_CLK_SPEED_HZ,
  .fnI2C_Init      = TCA9543A_I2CInit,
  .fnI2C_Transfer  = TCA9543A_I2CTranfert,
  //--- Time call function ---
  .fnGetCurrentms  = GetCurrentms_V71, // Can be NULL if MLX90640_ChangeI2Caddress() will not be used
  //--- Device EEPROM ---
#if !defined(MLX90640_PRECALCULATE_PIXELS_COEFFS)
  .EEPROM          = &IrCAM_EEPROM,
#endif
  //--- Device parameters ---
  .Params          = &IrCAM_Params,
};
```

```
//---------------------------------------------------------------------------
MLX90640_Config IrCAM_Config =
{
  //--- Subpage configuration ---
  .SubpageMode    = MLX90640_MEASURE_ALTERNATE_SUBPAGES,
  .RefreshRate    = MLX90640_IR_REFRESH_RATE_16Hz,
  .ReadingPattern = MLX90640_READING_CHESS_PATTERN_MODE,
  .ADCresolution  = MLX90640_ADC_RESOLUTION_18bits,
  //--- I2C configuration ---
  .I2C_FMpEnable            = false,
  .SetThresholdTo1V8        = false,
  .SetSDAdriverCurrentLimit = false,
};
```

Example of driver configuration in a .h file:

```
extern MLX90640_EEPROM IrCAM_EEPROM;         // Can be reused for all devices
extern MLX90640_Parameters IrCAM_Params;     // One per device
extern MLX90640_FramePolling IrFramePolling; // One per device

extern struct MLX90640 MLX90640_V71;
#define IrCAM  &MLX90640_V71


//---------------------------------------------------------------------------
extern MLX90640_Config IrCAM_Config;
```

8.2. Get a frame

First, you need to set the Emissivity and T_r of the MLX90640_FramePolling structure.

```
IrFramePolling.Emissivity = 1.0f;
IrFramePolling.Tr = 25.0f - 8.0f; // Set the first Tr at 25-8°. Next will be get from the last calculated frame
```

In the application program, call as often as possible the MLX90640_PollFrameTo() function.

When the function returns:

- ERR__BUSY, that indicates there is no new update to the MLX90640_FramePolling.Result (MLX90640_FrameTo)
- ERR_OK, that indicates the MLX90640_FramePolling.Result (MLX90640_FrameTo) have been updated
- Other result is an error

This function takes care of retrieving the new subframes by using DMA, and during the transfer, the function performs these operations on the last subframe received sequentially on each call of the polling function:

1. Call MLX90640_BigEndianToLittleEndian() on the subframe data (only if the DMA driver do not perform endianness transform, for example the processor is already with big-endianness)
2. A check of the received data
3. Call the MLX90640_CalculateTo() function
4. Call the MLX90640_CorrectBadPixels() function

So to get a refresh, you need at least 5 call of the MLX90640_PollFrameTo() function

After you can exploit the MLX90640_FramePolling.Result variable and do what you want.

8.2.1. Example

Example of frame get and extract:

```
//--- Frame available ? ---
Error = MLX90640_PollFrameTo(IrCAM, &IrFramePolling); // Call often this function
if (Error == ERR_OK)
{
  //--- Do what you want with the frame ---
  // Do stuff with the IrFramePolling.Result

  IrFramePolling.Tr = IrFramePolling.Result.Ta - 8.0f; // Calculate reflected signal for next subframeTr: Tr = Ta - 8°
}
else if ((Error != ERR__BUSY) && (Error != ERR__I2C_BUSY)) ShowError(Error);
```

See the result in §14.

9. SUMMARY OF DRIVER FUNCTIONS

9.1. Initialization and availability

eERRORRESULT ***Init_MLX90640****(MLX90640 **pComp**, const MLX90640_Config **pConf**)*
→ MLX90640 device initialization

bool ***MLX90640_PollDevice****(MLX90640 **pComp**)*
→ Poll the acknowledge from the MLX90640

bool ***MLX90640_IsFrameAvailable****(MLX90640 **pComp**)*
→ Poll the acknowledge from the MLX90640

eERRORRESULT ***MLX90640_GetDeviceID****(MLX90640 **pComp**, eMLX90640_Devices* **device**, uint16_t* **deviceId1**, uint16_t* **deviceId2**, uint16_t* **deviceId3**)*
→ Get actual device of the MLX90640

9.2. Read from EEPROM, RAM and registers

eERRORRESULT ***MLX90640_ReadData****(MLX90640 **pComp**, const uint16_t **address**, uint16_t* **data**, size_t **size**)*
→ Read data from the MLX90640

eERRORRESULT ***MLX90640_ReadDataWithDMA****(MLX90640 **pComp**, const uint16_t **address**, uint16_t* **data**, size_t **size**, I2C_Conf **configResult**)*
→ This function reads data using DMA from the MLX90640 device

eERRORRESULT ***MLX90640_ReadRegister****(MLX90640 **pComp**, const uint16_t **address**, uint16_t* **data**)*
→ This function reads a register from the MLX90640 device

eERRORRESULT ***MLX90640_DumpEEPROM****(MLX90640 **pComp**, MLX90640_EEPROM **eepromDump**)*
→ This function reads the entire address range of the internal EEPROM of the MLX90640 device

eERRORRESULT ***MLX90640_GetFrameData****(MLX90640 **pComp**, MLX90640_FrameData **frameData**)*
→ Get the last frame data on the MLX90640 device

9.3. Write to registers

eERRORRESULT ***MLX90640_WriteData****(MLX90640 **pComp**, const uint16_t **address**, const uint16_t* **data**, size_t **size**)*
→ Write data to the MLX90640

eERRORRESULT ***MLX90640_WriteRegister****(MLX90640 **pComp**, const uint16_t **address**, const uint16_t* **data**)*
→ This function writes data to a register of the MLX90640 device

9.4. Device configuration

eERRORRESULT ***MLX90640_ConfigureDeviceI2C****(MLX90640 **pComp**, bool **i2cFMpEnable**, bool **setThresholdTo1V8**, bool **setSDAdriverCurrentLimit**)*
→ Configure the I2C on the MLX90640 device

eERRORRESULT ***MLX90640_ConfigureDevice****(MLX90640 **pComp**, eMLX90640_SubpageMode **subpageMode**, eMLX90640_RefreshRate **refreshRate**, eMLX90640_ReadingPattern **readingPattern**, eMLX90640_ADCresolution **adcResolution**)*
→ Configure the MLX90640 device

eERRORRESULT ***MLX90640_ChangeI2Caddress****(MLX90640 **pComp**, uint8_t **newAddress**)*
→ Change the I2C address of the MLX90640 device

9.5. Device's parameters

eERRORRESULT ***MLX90640_ExtractDeviceParameters****(MLX90640 **pComp**, MLX90640_EEPROM **eepromDump**, bool **dumpEEPROM**)*

→ Extract device's parameters from an EEPROM dump of a MLX90640 device

eERRORRESULT ***MLX90640_BigEndianToLittleEndian****(uint16_t **data**, size_t **size**)*

→ Invert the endianness of the data array from big endian to little endian

9.6. Calculus and processing

eERRORRESULT ***MLX90640_CalculateTo****(MLX90640 **pComp**, MLX90640_FrameData **frameData**, float **emissivity**, float **tr**, MLX90640_FrameTo **result**)*

→ This function calculates the subframe T_o (Object Temperature) of the frame data previously extracted

eERRORRESULT ***MLX90640_CorrectBadPixels****(MLX90640 **pComp**, MLX90640_FrameTo **result**)*

→ This function corrects the defective pixel value by replacing its value by an interpolation of its neighboring pixels

9.7. Polling (asynchronous)

eERRORRESULT ***MLX90640_PollFrameTo****(MLX90640 **pComp**, MLX90640_FramePolling **result**)*

→ This function, when called regularly, process a subframe while the other is retrieved by DMA

10. CONFIGURATION STRUCTURES

10.1. Device object structure

The MLX90640 device object structure contains all information that is mandatory to work with a device. It is always the first parameter of each function of the driver.

Source code:

```
typedef struct MLX90640 MLX90640;
typedef uint16_t TMLX90640DriverInternal;

struct MLX90640
{
  void *UserDriverData;
  TMLX90640DriverInternal InternalConfig;

  //--- Interface driver params and call functions ---
  uint8_t I2Caddress;
  void *InterfaceDevice;
  uint32_t I2CclockSpeed;
  MLX90640_I2CInit_Func fnI2C_Init;
  MLX90640_I2CTransfer_Func fnI2C_Transfer;

  //--- Time call function ---
  GetCurrentms_Func fnGetCurrentms;

  //--- Device EEPROM ---
#if !defined(MLX90640_PRECALCULATE_PIXELS_COEFFS)
  MLX90640_EEPROM* EEPROM;
#endif

  //--- Device parameters ---
  MLX90640_Parameters* Params;
};
```

10.1.1. Data fields

***void* *UserDriverData**

Optional, can be used to store driver data related to the project and the device or NULL. This field is not used or modified by the driver.

***TMLX90640DriverInternal* InternalConfig**

This is the internal driver configuration. The user should not change the value, only the driver can change values. The value is changed following driver usage.

Type

typedef *uint16_t* TMLX90640DriverInternal

Initial value / default

Regardless of the value set when filling the struct, the value will be modified at device initialization when using the function Init_MLX90640()

***uint8_t* I2Caddress**

Device address set into the I2C address (0x800F) register. Use MLX90640_CHIPADDRESS_DEFAULT if you do not have change the address. To change the device address, use the MLX90640_ChangeI2Caddress() function and reset the device.

***void* *InterfaceDevice**

This is the pointer that will be in the first parameter of all interface call functions.

***uint32_t* I2CclockSpeed**

Clock frequency of the I2C interface in Hertz.

MLX90640_I2CInit_Func **fnI2C_Init**

This function will be called at driver initialization to configure the I2C interface driver.

Type

typedef *eERRORRESULT (*MLX90640_I2CInit_Func)(void *, const uint32_t)*

Initial value / default

This function must point to a function else a ERR__PARAMETER_ERROR is returned by the function Init_MLX90640().

MLX90640_I2CTransfer_Func **fnI2C_Transfer**

This function will be called when the driver needs to transfer data over the I2C communication with the device.

Type

typedef *eERRORRESULT (*MLX90640_I2CTransfer_Func)(void *, const uint8_t, uint8_t *, size_t, bool, bool)*

Initial value / default

This function must point to a function else an ERR__PARAMETER_ERROR is returned when using a function that require to communicate with the device.

GetCurrentms_Func **fnGetCurrentms**

This function will be called when the driver needs to get current millisecond. Some functions need a timeout, without, they can be stuck forever.

Type

typedef *uint32_t (*GetCurrentms_Func)(void)*

Initial value / default

This function has to point to a function else an ERR__PARAMETER_ERROR is returned by the functions MLX90640_ChangeI2Caddress(). If the MLX90640_ChangeI2Caddress() will not be used, this parameter can be set to NULL.

MLX90640_EEPROM ***EEPROM**

Device EEPROM. Need to store a dump of the device's EEPROM for further calculations.

This parameter only exists if the driver configuration has MLX90640_PRECALCULATE_PIXELS_COEFFS not defined

Type

struct MLX90640_EEPROM

MLX90640_Parameters ***Params**

Device parameters. Will be filled after calling the function MLX90640_ExtractDeviceParameters().

Type

struct MLX90640_Parameters

10.1.2. TMLX90640DriverInternal type and InternalConfig variable

Warning: This variable should never be changed by the application.

This variable is used by the driver to estimate the state of the device and some information that is impossible to retrieve directly from device. This saves some unnecessary transfer communications.

For information, the TMLX90640DriverInternal type is defined from a uint16_t and is constituted as this `...PAA.ttttttdep` where:

P	'0' → Interleaved (TV) mode (set by using the define MLX90640_READING_INTERLEAVED) '1' → Chess pattern (set by using the define MLX90640_READING_CHESS_PATTERN)
A	'00' → ADC set to 16 bit resolution (enum eMLX90640_ADCresolution::MLX90640_ADC_RESOLUTION_16bits) '01' → ADC set to 17 bit resolution (enum eMLX90640_ADCresolution::MLX90640_ADC_RESOLUTION_17bits) '10' → ADC set to 18 bit resolution (enum eMLX90640_ADCresolution::MLX90640_ADC_RESOLUTION_18bits) '11' → ADC set to 19 bit resolution (enum eMLX90640_ADCresolution::MLX90640_ADC_RESOLUTION_19bits)
.	Not used
t	Current DMA transaction number (updated by the I2C+DMA driver)
d	'0' → No DMA transfer in progress '1' → DMA transfer in progress (set by using the define MLX90640_DMA_TRANSFER_IN_PROGRESS)
e	'0' → CPU is little endian (set by using the define MLX90640_LITTLE_ENDIAN) '1' → CPU is big endian (set by using the define MLX90640_BIG_ENDIAN)
p	'0' → Device is not parameterized '1' → Device is parameterized (set by using the define MLX90640_DEV_PARAMETERIZED)

10.1.2.1. Defines

```
#define MLX90640_DEV_PARAMETERIZED
```

Select the little endianness.

Value

0x0001u

```
#define MLX90640_LITTLE_ENDIAN
```

Select the little endianness.

Value

0x0000u

```
#define MLX90640_BIG_ENDIAN
```

Select the big endianness.

Value

0x0002u

```
#define MLX90640_DMA_TRANSFER_IN_PROGRESS
```

Select the little endianness.

Value

0x0004u

10.1.3. Driver interface handle functions

```
eERRORRESULT (*MLX90640_I2CInit_Func)(
        void *pIntDev,
        const uint32_t sclFreq)
```

Function for interface driver initialization of the MLX90640. This function will be called at driver initialization to configure the interface driver.

Parameters

Input	*pIntDev	Is the MLX90640.InterfaceDevice of the device that call the interface initialization
Input	sclFreq	Is the SCL frequency in Hz to set at the interface initialization

Return

Returns an *eERRORRESULT* value enumerator dependent of how the return error is implemented by the user. It is recommended, during the implement of the pointer interface function, to return only ERR__I2C_PARAMETER_ERROR, ERR__I2C_COMM_ERROR, ERR__I2C_CONFIG_ERROR, ERR__I2C_TIMEOUT, or ERR__I2C_FREQUENCY_ERROR when there is an error and ERR_OK when all went fine.

```
eERRORRESULT (*MLX90640_I2CTransfer_Func)(
        void *pIntDev,
        const uint8_t deviceAddress,
        uint8_t *data,
        size_t byteCount,
        bool start,
        bool stop)
```

Function for interface transfer of the MLX90640. This function will be called when the driver needs to transfer data over the I2C communication with the device. Can be a read of data or a transmit of data. It also indicates if it needs a start and/or a stop.

Parameters

Input	*pIntDev	Is the MLX90640.InterfaceDevice of the device that call the I2C transfer
Input	deviceAddress	Is the device address on the bus (8-bits only). The LSB bit indicate if it is a I2C Read (bit at '1') or a I2C Write (bit at '0')
In/Out	*data	Is a pointer to memory data to write in case of I2C Write, or where the data received will be stored in case of I2C Read (can be NULL if no data transfer other than chip address)
Input	byteCount	Is the byte count to write over the I2C bus or the count of byte to read over the bus
Input	start	Indicate if the transfer needs a start (in case of a new transfer) or restart (if the previous transfer has not been stopped)
Input	stop	Indicate if the transfer needs a stop after the last byte sent

Return

Returns an *eERRORRESULT* value enumerator dependent of how the return error is implemented by the user. It is recommended, during the implement of the pointer interface function, to return only ERR__I2C_NACK, ERR__I2C_NACK_ADDR, ERR__I2C_NACK_DATA, ERR__I2C_PARAMETER_ERROR, ERR__I2C_COMM_ERROR, ERR__I2C_CONFIG_ERROR, ERR__I2C_TIMEOUT, ERR__I2C_DEVICE_NOT_READY, ERR__I2C_INVALID_ADDRESS, ERR__I2C_INVALID_COMMAND, ERR__I2C_FREQUENCY_ERROR, ERR__I2C_OVERFLOW_ERROR, ERR__I2C_BUSY, ERR__I2C_UNDERFLOW_ERROR, or ERR__I2C_OTHER_BUSY when there is an error and ERR_OK when all went fine.

*uint32_t (***GetCurrentms_Func**)(void)*

Function that gives the current millisecond of the system to the driver. This function will be called when the driver needs to get current millisecond.

Return

Returns the current millisecond of the system

10.2. MLX90640 configuration object structure

The MLX90640 configuration object structure contains all information that is mandatory to configure the device.

Source code:

```
typedef struct MLX90640_Config
{
  //--- Subpage configuration ---
  eMLX90640_SubpageMode SubpageMode;
  eMLX90640_RefreshRate RefreshRate;
  eMLX90640_ReadingPattern ReadingPattern;
  eMLX90640_ADCresolution ADCresolution;

  //--- I2C configuration ---
  bool I2C_FMp_Enable;
  bool SetThresholdTo1V8;
  bool SetSDAdriverCurrentLimit;
} MLX90640_Config;
```

10.2.1. Data fields

eMLX90640_SubpageMode **SubpageMode**

Determine the working mode of the subpages.

Type

enum eMLX90640_SubpageMode

eMLX90640_RefreshRate **RefreshRate**

IR refresh rate of subpages.

Type

enum eMLX90640_RefreshRate

eMLX90640_ReadingPattern **ReadingPattern**

Reading pattern of the IR array.

Type

enum eMLX90640_ReadingPattern

eMLX90640_ADCresolution **ADCresolution**

Select the ADC resolution of the IR sensors.

Type

enum eMLX90640_ADCresolution

bool **I2C_FMp_Enable**

I2C FM+: 'true' = Enable (SCL @ 1MHz) ; 'false' = Disable (SCL @ 400kHz).

bool **SetThresholdTo1V8**

I2C threshold level to 1.8V: 'true' = threshold to 1.8V ; 'false' = threshold to Vdd.

bool **SetSDAdriverCurrentLimit**

I2C driver current limit: 'true' = Enable ; 'false' = Disable.

10.2.2. Enumerators

enum **eMLX90640_SubpageMode**

Enumerator of all possible subpage modes.

Enumerator

MLX90640_MEASURE_ONLY_SUBPAGE0	**0x00**	Measure only subpage 0
MLX90640_MEASURE_ONLY_SUBPAGE1	**0x01**	Measure only subpage 1
MLX90640_MEASURE_ALTERNATE_SUBPAGES	**0x02**	Measure alternates between subpage 0 and subpage 1 (0->1->0->1...)

enum **eMLX90640_RefreshRate**

Enumerator of all possible refresh rate control.

Enumerator

MLX90640_IR_REFRESH_RATE_0Hz5	**0b000**	IR refresh rate = 0.5Hz
MLX90640_IR_REFRESH_RATE_1Hz	**0b001**	IR refresh rate = 1Hz
MLX90640_IR_REFRESH_RATE_2Hz	**0b010**	IR refresh rate = 2Hz (default)
MLX90640_IR_REFRESH_RATE_4Hz	**0b011**	IR refresh rate = 4Hz
MLX90640_IR_REFRESH_RATE_8Hz	**0b100**	IR refresh rate = 8Hz
MLX90640_IR_REFRESH_RATE_16Hz	**0b101**	IR refresh rate = 16Hz
MLX90640_IR_REFRESH_RATE_32Hz	**0b110**	IR refresh rate = 32Hz (Only possible with I2C clock @ 1MHz)
MLX90640_IR_REFRESH_RATE_64Hz	**0b111**	IR refresh rate = 64Hz (Only possible with I2C clock @ 1MHz with DMA)

enum **eMLX90640_ReadingPattern**

Enumerator of all possible reading pattern.

Enumerator

MLX90640_READING_INTERLEAVE_MODE	**0x00**	Reading in Interleaved (TV) mode
MLX90640_READING_CHESS_PATTERN_MODE	**0x01**	Reading in Chess pattern (default)

enum **eMLX90640_ADCresolution**

Enumerator of all possible ADC resolution control.

Enumerator

MLX90640_ADC_RESOLUTION_16bits	**0b00**	ADC set to 16-bit resolution
MLX90640_ADC_RESOLUTION_17bits	**0b01**	ADC set to 17-bit resolution
MLX90640_ADC_RESOLUTION_18bits	**0b10**	ADC set to 18-bit resolution (default)
MLX90640_ADC_RESOLUTION_19bits	**0b11**	ADC set to 19-bit resolution

10.3. Function's return error enumerator

enum **eERRORRESULT**

There is only one error code at the same time returned by the functions. The only code that indicates that all went fine is ERR_OK.

Enumerator

ERR_OK	**0**	Succeeded
ERR__NO_DEVICE_DETECTED	**1**	No device detected
ERR__UNKNOWN_ELEMENT	**3**	Unknown element (type or value)
ERR__PARAMETER_ERROR	**9**	Parameter error
ERR__DATA_NOT_INITIALIZED	**28**	Data not initialized
ERR__UNKNOWN_DEVICE	**30**	Unknown device error
ERR__NULL_BUFFER	**31**	Null buffer parameter
ERR__TOO_MANY_BAD	**32**	Too many bad things
ERR__TWO_BAD_SIDE_BY_SIDE	**33**	Two bad things side by side
ERR__BAD_DATA	**37**	Bad data
ERR__BUSY	**38**	Busy
ERR__I2C_NACK	**210**	Received a I2C not acknowledge
ERR__I2C_NACK_ADDR	**211**	Received a I2C not acknowledge while transferring addr
ERR__I2C_NACK_DATA	**212**	Received a I2C not acknowledge while transferring data
ERR__I2C_PARAMETER_ERROR	**213**	I2C parameter error
ERR__I2C_COMM_ERROR	**214**	I2C communication error
ERR__I2C_CONFIG_ERROR	**215**	I2C configuration error
ERR__I2C_TIMEOUT	**216**	I2C communication timeout
ERR__I2C_DEVICE_NOT_READY	**217**	I2C device not ready
ERR__I2C_INVALID_ADDRESS	**218**	I2C invalid address
ERR__I2C_INVALID_COMMAND	**219**	I2C invalid command
ERR__I2C_FREQUENCY_ERROR	**220**	I2C frequency error
ERR__I2C_OVERFLOW_ERROR	**221**	I2C overflow error
ERR__I2C_UNDERFLOW_ERROR	**222**	I2C underflow error
ERR__I2C_BUSY	**223**	I2C busy
ERR__I2C_OTHER_BUSY	**224**	I2C busy by other transfer
ERR__TEST_ERROR	**255**	Test error

11. DRIVER'S FUNCTIONS

Here is the use of all functions related to the driver.

11.1. Initialization and availability

```
eERRORRESULT Init_MLX90640(
        MLX90640 *pComp,
        const MLX90640_Config *pConf)
```

This function initializes the MLX90640 driver and call the initialization of the interface driver (I2C). Next it checks parameters and configures the MLX90640.

Parameters

Input	*pComp	Is the pointed structure of the device to be initialized
Input	*pConf	Is the pointed structure of the device configuration

Return

Returns an *eERRORRESULT* value enumerator. Below are some returned by the function itself but not errors returned by called functions.

- ERR__PARAMETER_ERROR when pComp or pConf is NULL, or Interface functions are NULL
- ERR__I2C_FREQUENCY_ERROR when I2CclockSpeed parameter is out of range frequency
- ERR__NO_DEVICE_DETECTED when no communication with the device is possible

```
bool MLX90640_PollDevice(MLX90640 *pComp)
```

Poll the acknowledge from the MLX90640.

Parameters

Input	*pComp	Is the pointed structure of the device to be used

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or data is NULL, or Interface functions are NULL

```
bool MLX90640_IsFrameAvailable(MLX90640 *pComp)
```

Poll the flag of a new data available in RAM from the MLX90640.

Parameters

Input	*pComp	Is the pointed structure of the device to be used

Return

Returns 'true' if ready else 'false'

```
eERRORRESULT MLX90640_GetDeviceID(MLX90640 *pComp,
      eMLX90640_Devices* device,
      uint16_t* deviceId1,
      uint16_t* deviceId2,
      uint16_t* deviceId3)
```

Get actual device of the MLX90640.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*device	Is the device found (MLX90640BAA or MLX90640BAB)
Output	*deviceId1	Is the returned device ID1 (This parameter can be NULL if not needed)
Output	*deviceId2	Is the returned device ID2 (This parameter can be NULL if not needed)
Output	*deviceId3	Is the returned device ID3 (This parameter can be NULL if not needed)

Return

Returns an *eERRORRESULT* value enumerator. Below are some returned by the function itself but not errors returned by called functions.

- ERR__PARAMETER_ERROR when pComp or device is NULL, or Interface functions are NULL
- ERR__UNKNOWN_DEVICE when the device field of view which indicate the BAA or BAB version cannot be found

11.1.1. Enumerators

```
enum eMLX90640_Devices
```

List of supported devices.

Enumerator

MLX90640BAA	**0x00**	Device MLX90640BAA - FOV = 110°x75°
MLX90640BAB	**0x01**	Device MLX90640BAB - FOV = 55°x35°

11.2. Read from EEPROM, RAM and registers

```
eERRORRESULT MLX90640_ReadData(
        MLX90640 *pComp,
        const uint16_t address,
        uint16_t* data,
        size_t size)
```

Read data from the MLX90640. This function will convert the big-endian data (the device communicates in big endian) to the endianness of the CPU.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address where data will be read in the MLX90640 (address will be incremented automatically)
Output	*data	Is where the data will be stored
Input	size	Is the size of the data array to read (count of 16-bits data)

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or data is NULL, or Interface functions are NULL

```
eERRORRESULT MLX90640_ReadDataWithDMA(
        MLX90640 *pComp,
        const uint16_t address,
        uint16_t* data,
        size_t size,
        I2C_Conf *configResult)
```

This function reads data using DMA from the MLX90640 device. This function will ask the I2C driver convert the big-endian data (the device communicates in big endian) to the endianness of the CPU.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address where data will be read in the MLX90640 (address will be incremented automatically)
Output	*data	Is where the data will be stored
Input	size	Is the size of the data array to read (count of 16-bits data)
Output	*configResult	Is the result of the configuration of the transfer. This will contain the I2C_ENDIAN_RESULT of the transfer (usually if the I2C DMA transfer have done the endian transformation)

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or data is NULL, or Interface functions are NULL

```
eERRORRESULT MLX90640_ReadRegister(
        MLX90640 *pComp,
        const uint16_t address,
        uint16_t* data)
```

This function reads a register from the MLX90640 device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the register address where data will be read in the MLX90640
Output	*data	Is where the data will be stored

Return

Returns an *eERRORRESULT* value enumerator. See MLX90640_ReadData()'s returns value for more information.

```
eERRORRESULT MLX90640_DumpEEPROM(
        MLX90640 *pComp,
        MLX90640_EEPROM *eepromDump)
```

This function reads the entire address range of the internal EEPROM of the MLX90640 device. The function will take care of the max EEPROM I2C speed operations (see note 5 of the table 5 of the datasheet). Took around 40ms to dump the EEPROM from the MLX90640 device with SCL@400kHz.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*eepromDump	Is where the EEPROM dump will be stored

Return

Returns an *eERRORRESULT* value enumerator. See MLX90640_ReadData()'s returns value for more information.

```
eERRORRESULT MLX90640_GetFrameData(
        MLX90640 *pComp,
        MLX90640_FrameData *frameData)
```

Get the last frame data on the MLX90640 device. This function reads the entire address range of the internal RAM of the MLX90640 device. Took around 40ms to dump the EEPROM from the MLX90640 device with SCL@400kHz.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*frameData	Is where the frame data will be stored

Return

Returns an *eERRORRESULT* value enumerator. See MLX90640_ReadData()'s returns value for more information.

11.2.1. Structures

11.2.1.1. MLX90640_EEPROM

The EEPROM structure contains all the parameters information but in a raw format. This structure is only useful for the MLX90640_ExtractDeviceParameters(), and MLX90640_CalculateTo() functions. To get the EEPROM structure, use the MLX90640_DumpEEPROM(), or MLX90640_ExtractDeviceParameters().

11.2.1.2. MLX90640_Parameters

The Parameters structure contains all the device parameters information from the MLX90640_EEPROM but pre-calculated. This structure is used by MLX90640_CalculateTo(), and MLX90640_CorrectBadPixels() functions. To fill the parameters structure, use the MLX90640_ExtractDeviceParameters() function.

11.2.1.3. MLX90640_FrameData

The MLX90640_FrameData structure contains all the frame information but in a raw format. This structure is only useful for the MLX90640_CalculateTo() function. To get the EEPROM structure, use the MLX90640_GetFrameData() functions.

11.2.1.4. MLX90640_FramePolling

The MLX90640_FramePolling structure contains the MLX90640_FrameData for both subframes and the MLX90640_FrameTo, plus additional data that makes the polling works with minimum manipulation in the application software. This structure is only useful for the MLX90640_PollFrameTo() function.

11.3. Write to registers

```
eERRORRESULT MLX90640_WriteData(
      MLX90640 *pComp,
      const uint16_t address,
      const uint16_t* data,
      size_t size)
```

Write data to the MLX90640. This function will convert data the from endianness of the CPU to big endian (the device communicates in big endian).

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address where data will be written in the MLX90640 (address will be incremented automatically)
Input	*data	Is the data array to write
Input	size	Is the size of the data array to store (count of 16-bits data)

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or data is NULL, or Interface functions are NULL, or Address is too high
- ERR__OUT_OF_RANGE when you want to send a not multiple of 4 data to RAM

```
eERRORRESULT MLX90640_WriteRegister(
      MLX90640 *pComp,
      const uint16_t address,
      const uint16_t* data)
```

This function writes data to a register of the MLX90640 device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address where data will be written in the MLX90640
Input	*data	Is the data array to store

Return

Returns an *eERRORRESULT* value enumerator. See MLX90640_WriteData()'s returns value for more information.

11.4. Device configuration

```
eERRORRESULT MLX90640_ConfigureDeviceI2C(
        MLX90640 *pComp,
        bool i2cFMpEnable,
        bool setThresholdTo1V8,
        bool setSDAdriverCurrentLimit)
```

Configure the I2C on the MLX90640 device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	i2cFMpEnable	Indicate if the FM+ mode of the device should be activated
Input	setThresholdTo1V8	Set the I2C threshold of the device. 'true' to set to 1.8V, 'false' to set to Vdd
Input	setSDAdriverCurrentLimit	Indicate if the device should limit its current on SDA pin

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL

```
eERRORRESULT MLX90640_ConfigureDevice(
        MLX90640 *pComp,
        eMLX90640_SubpageMode subpageMode,
        eMLX90640_RefreshRate refreshRate,
        eMLX90640_ReadingPattern readingPattern,
        eMLX90640_ADCresolution adcResolution)
```

Configure the MLX90640 device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	subpageMode	Indicate the subpage mode of the device
Input	refreshRate	Indicate the refresh rate of the device
Input	readingPattern	Indicate the reading pattern of the device
Input	adcResolution	Indicate the ADC resolution of the device

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL

```
eERRORRESULT MLX90640_ChangeI2Caddress(
        MLX90640 *pComp,
        uint8_t newAddress)
```

Change the I2C address of the MLX90640 device. After using this function, reset the device to use it at its new address.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	newAddress	Is the new I2C address to set. The value shall be > 0x00 and < 0x7F

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL or pComp->fnGetCurrentms is NULL
- ERR__I2C_INVALID_ADDRESS when the new address is < 0x01 or > 0x7F
- ERR__DATA_NOT_INITIALIZED when the MLX90640's I2C address register doesn't have an expected value. If this error happens, the device may no longer respond and can be considered as defective

11.5. Device's parameters

```
eERRORRESULT MLX90640_ExtractDeviceParameters(
      MLX90640 *pComp,
      MLX90640_EEPROM *eepromDump,
      bool dumpEEPROM)
```

Extract device's parameters from an EEPROM dump of a MLX90640 device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input/Output	*eepromDump	Is where the EEPROM dump will be stored if extracted or the EEPROM dump to use for parameters extraction
Input	dumpEEPROM	Indicate if the EEPROM shall be extracted and stored in *eepromDump

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL or pComp->Params is NULL
- ERR__NULL_BUFFER when eepromDump is NULL
- ERR__UNKNOWN_ELEMENT when the EEPROM dump seems not good
- ERR__TOO_MANY_BAD when there are too many bad pixels (broken and/or outlier) for the device
- ERR__TWO_BAD_SIDE_BY_SIDE when there are 2 neighbor pixels that are defective in the X axis or Y axis

```
void MLX90640_BigEndianToLittleEndian(
      uint16_t *data,
      size_t size)
```

Invert the endianness of the data array from big endian to little endian. In processor with little endianness, this function is useful because the device is communicating data in big endian.

Parameters

Input/Output	*data	Is the data array where to invert endianness
Input	size	Count of uint16_t data in the array

Return

Nothing

11.6. Calculus and processing

```
eERRORRESULT MLX90640_CalculateTo(
        MLX90640 *pComp,
        MLX90640_FrameData *frameData,
        float emissivity,
        float tr,
        MLX90640_FrameTo *result)
```

This function calculates the subframe T_o (Object Temperature) of the frame data previously extracted. It will only update the associated subframe's pixels and let others pixels untouched.

In order to compensate correctly for the emissivity and achieve best accuracy we need to know the surrounding temperature which is responsible for the second component of the IR signal namely the reflected part -Tr. In case this temperature is not available and cannot be provided it might be replaced by $T_r \approx T_a - 8$.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	*frameData	Is the extracted frame data from the device
Input	emissivity	Is the IR signal emitted by the object, if unknown set to 1.0f
Input	tr	Is the IR signal reflected from the object (the source of this signal is surrounding environment of the sensor), if unknown set to $T_a - 8$
Input	*result	Is where the result will be stored

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp, or frameData, or result is NULL, or pComp->Params is NULL.

```
eERRORRESULT MLX90640_CorrectBadPixels(
        MLX90640 *pComp,
        MLX90640_FrameTo *result)
```

This function corrects the defective pixel value by replacing its value by an interpolation of its neighboring pixels (See §9 of datasheet), here it is a mean of the neighboring pixels (X-1, X+1, Y-1, and Y+1, when available).

This function changes only defective pixels, others will not be touched.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input/output	*result	Is where the result will be stored

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or result is NULL, or pComp->Params is NULL.

11.7. Polling (asynchronous)

```
eERRORRESULT MLX90640_PollFrameTo(
        MLX90640 *pComp,
        MLX90640_FramePolling *result)
```

This function, when called regularly, process a subframe while the other is retrieved by DMA. In case of no DMA, the function retrieves the subframe and process it directly.

Warning

This function has a working buffer of 6,5kiB (MLX90640_FramePolling)

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input/output	*result	Contains either the working frame data buffers and the result Frame To

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or result is NULL, or pComp->Params is NULL.
- ERR__BUSY when no new data available at the moment
- ERR_OK when the MLX90640_FramePolling.Result (MLX90640_FrameTo) is refresh with new data

11.7.1. FrameTo structure

The frame T_o result is where the user can exploit the data, like draw an image for example. This structure is used by the MLX90640_CalculateTo() function.

Source code:

```
typedef struct MLX90640_FrameTo
{
  // Frame data parameters
  union
  {
    float PixelYX[MLX90640_ROW_COUNT][MLX90640_COL_COUNT];
    float Pixel[MLX90640_TOTAL_PIXELS_COUNT];
  };
  // Auxiliary data
  float Vdd;
  float Ta;
  // Min, Max value
  float MinToSubpage[2];
  float MaxToSubpage[2];
  // Moving Average Filter
#if (MLX90640_MOVING_AVERAGE_FILTER_VALUES_COUNT > 1)
  float PixGainCPSP0_Filter[MLX90640_MOVING_AVERAGE_FILTER_VALUES_COUNT];
  size_t Filter0_Index;
  float PixGainCPSP1_Filter[MLX90640_MOVING_AVERAGE_FILTER_VALUES_COUNT];
  size_t Filter1_Index;
#endif
} MLX90640_FrameTo;
```

11.7.1.1. Data fields

float **PixelYX**[MLX90640_ROW_COUNT][MLX90640_COL_COUNT]
float **Pixel**[MLX90640_TOTAL_PIXELS_COUNT]

The T_o value of each pixel in degrees.

float **Vdd**

V_{dd} of the last subframe.

float **Ta**

Ambient Temperature of the last subframe.

float **MinToSubpage**[2]

Minimum T_o value of each subframes. The minimum of the two is the minimum temperature of the frame.

float **MaxToSubpage**[2]

Maximum T_o value of each subframes. The maximum of the two is the maximum temperature of the frame.

PixGainCPSP0_Filter + Filter0_Index, and PixGainCPSP1_Filter + Filter1_Index are only available if the driver uses a moving average filter (with the define MLX90640_MOVING_AVERAGE_FILTER_VALUES_COUNT) and SHOULD NOT BE MODIFIED BY THE USER!

12. EXAMPLE OF “CONF_MLX90640.H” FILE

Source:

```
#ifndef CONF_MLX90640_H
#define CONF_MLX90640_H
//============================================================================



// If in debug mode, check NULL parameters that are mandatory in each function of the driver
#ifdef DEBUG
#  define CHECK_NULL_PARAM
#endif


// This define specify to the driver to pre-calculate Offset, Sensitivity, Kta, and Kv of each pixel and save them in
the MLX90640_Parameters
// If the following define is set, the driver will take 768x2+768x4x3 = 10704 bytes of RAM more to store theses
values
// If unset then the driver will take less ram but more time to calculate To of each pixels
#define MLX90640_PRECALCULATE_PIXELS_COEFFS


// In order to limit the noise in the final To calculation it is advisable to filter the CP readings at this point of
calculation
// This filter is only useful with device with thermal gradient compensation
// A good practice would be to apply a Moving Average Filter with length of 16 or higher
// If set to < 2, the filter will be disabled
#define MLX90640_MOVING_AVERAGE_FILTER_VALUES_COUNT  ( 16 )



//----------------------------------------------------------------------------
#endif /* CONF_MLX90640_H */
```

13. EXAMPLE OF DRIVER INTERFACE HANDLE FUNCTIONS (WITHOUT DMA)

This is an extract from the project for the SAMV71 Xplained Ultra board.

Example of driver interface handle functions in a .c file:

```
//=============================================================================
// Get millisecond
//=============================================================================
uint32_t GetCurrentms_V71(void)
{
  return msCount;
}


//*****************************************************************************************************************
//=============================================================================
// Hardware I2C driver interface configuration for the ATSAMV71
//=============================================================================
eERRORRESULT HardI2C_InterfaceInit_V71(void *pIntDev, const uint32_t sclFreq)
{
  if (pIntDev == NULL) return ERR__I2C_PARAMETER_ERROR;
  Twihs *I2C = (Twihs *)pIntDev;      // MCU specific: #define TWIHS0  ((Twihs*)0x40018000U) // (TWIHS0) Base Address

  //--- Reset ---
  if (SDA0_SOFT_Status == 0) // A device is stuck in communication and wait for a missing clocks
  {
    SDA0_SOFT_PIO_En;
    SDA0_SOFT_High;   // SDA = 1
    SDA0_SOFT_In;     // SDA in
    SCL0_SOFT_PIO_En;
    SCL0_SOFT_High;   // SCL = 1
    SCL0_SOFT_Out;    // SCL out
    size_t z = 9;                // Clock up to 9 cycles. Here we force I2C SCL to clock until a device stuck in
                                 // communication respond
    while (SDA0_SOFT_Status == 0) // Look for SDA high in each cycle while SCL is high and then break
    {
      delay_us(1);
      SCL0_SOFT_Low;  // SCL = 0
      delay_us(1);
      SCL0_SOFT_High; // SCL = 1
      if (--z == 0) break;
    }
    ioport_set_pin_mode(TWIHS0_DATA_GPIO, TWIHS0_DATA_FLAGS); // Restore SDA pin function
    ioport_disable_pin(TWIHS0_DATA_GPIO);                     // Restore SDA pin function
    ioport_set_pin_mode(TWIHS0_CLK_GPIO, TWIHS0_CLK_FLAGS);   // Restore SCL pin function
    ioport_disable_pin(TWIHS0_CLK_GPIO);                      // Restore SCL pin function
  }

  //--- Configuration of the TWI interface ---
  twihs_options_t opt;
  opt.speed = sclFreq;
  if (twihs_master_setup(BOARD_AT24MAC_TWIHS, &opt) != TWIHS_SUCCESS) return ERR__I2C_CONFIG_ERROR;
  I2C->TWIHS_CR |= TWIHS_CR_STOP;

  //--- Clear receive buffer ---
  uint8_t Data = BOARD_AT24MAC_TWIHS_INSTANCE->TWIHS_RHR;
  (void)Data; // Unused data
  //--- clear registers ---
  Data = BOARD_AT24MAC_TWIHS_INSTANCE->TWIHS_SR;
  (void)Data; // Unused data

  I2Cconfigured = true;
  return ERR_OK;
}
```

```
//===========================================================================
// Hardware I2C - Transfer data through an I2C communication for the ATSAMV71
//===========================================================================
eERRORRESULT HardI2C_Tranfert_V71(void *pIntDev, const uint8_t deviceAddress, uint8_t *data, size_t byteCount, bool
start, bool stop)
{
  if (pIntDev == NULL) return ERR__SPI_PARAMETER_ERROR;
  if (data == NULL) return ERR__SPI_PARAMETER_ERROR;
  Twihs *I2C = (Twihs *)pIntDev;      // MCU specific: #define TWIHS0  ((Twihs*)0x40018000U) // (TWIHS0) Base Address
  size_t RemainingBytes = byteCount;
  uint32_t Status;
  uint32_t Timeout = TWIHS_TIMEOUT;
  bool DeviceWrite  = ((deviceAddress & 0x01) == 0);
  I2C->TWIHS_MMR  = 0;
  I2C->TWIHS_MMR  = TWIHS_MMR_DADR(deviceAddress >> 1) | (DeviceWrite ? 0 : TWIHS_MMR_MREAD);
  I2C->TWIHS_IADR = 0; // Not used
  I2C->TWIHS_CR   = TWIHS_CR_MSEN | TWIHS_CR_SVDIS;

  //--- Device polling ? ---
  if ((data == NULL) || (byteCount <= 0))                       // Device polling only
  { // Little hack because TWI of V71 does not support device polling without using SMBus
    I2C->TWIHS_MMR &= ~TWIHS_MMR_MREAD;                         // The SMBus of this device does not support quick
                                                                // read command (no Stop will be sent)
    I2C->TWIHS_CR |= TWIHS_CR_SMBEN + TWIHS_CR_PECDIS;          // Enable SMBus
    I2C->TWIHS_CR |= TWIHS_CR_STOP;                             // Send a stop
    I2C->TWIHS_CR |= TWIHS_CR_QUICK;                            // Start the polling with a quick command

    Timeout = TWIHS_TIMEOUT;
    while (true)                                                // Wait the polling to finish
    {
      Status = I2C->TWIHS_SR;
      if ((Status & TWIHS_SR_NACK) > 0) return ERR__I2C_NACK;
      if (!Timeout--) return ERR__I2C_TIMEOUT;                  // Timeout ? return an error
      if ((Status & TWIHS_SR_TXCOMP) > 0) break;
    }
    return ERR_OK;
  }

  //--- Transfer data ---
  if (start) I2C->TWIHS_CR |= TWIHS_CR_START;                   // Send a start if asked
  if (DeviceWrite) // Device write
  {
    while (true)
    {
      Status = I2C->TWIHS_SR;
      if ((Status & TWIHS_SR_NACK) > 0) return ERR__I2C_NACK_DATA;
      if (!Timeout--) return ERR__I2C_TIMEOUT;                  // Timeout ? return an error
      if ((Status & TWIHS_SR_TXRDY) == 0) continue;
      Timeout = TWIHS_TIMEOUT;

      if (RemainingBytes == 0) break;                           // No data remaining to send, then break the loop
      I2C->TWIHS_THR = *data;                                   // Send next data byte
      data++;
      RemainingBytes--;
    }
    if (stop) I2C->TWIHS_CR |= TWIHS_CR_STOP;                   // Send a stop if asked
  }
  else // Device read
  {
    while (RemainingBytes > 0)
    {
      if ((RemainingBytes == 1) && stop)
        I2C->TWIHS_CR |= TWIHS_CR_STOP;                         // Last byte ? Send a stop if asked

      Timeout = TWIHS_TIMEOUT;
      while (true)                                              // Wait the polling to finish
      {
        Status = I2C->TWIHS_SR;
        if ((Status & TWIHS_SR_NACK) > 0)
        {
          if (RemainingBytes == byteCount)
               return ERR__I2C_NACK;
          else return ERR__I2C_NACK_DATA;
        }
        if (!Timeout--) return ERR__I2C_TIMEOUT;                // Timeout ? return an error
```

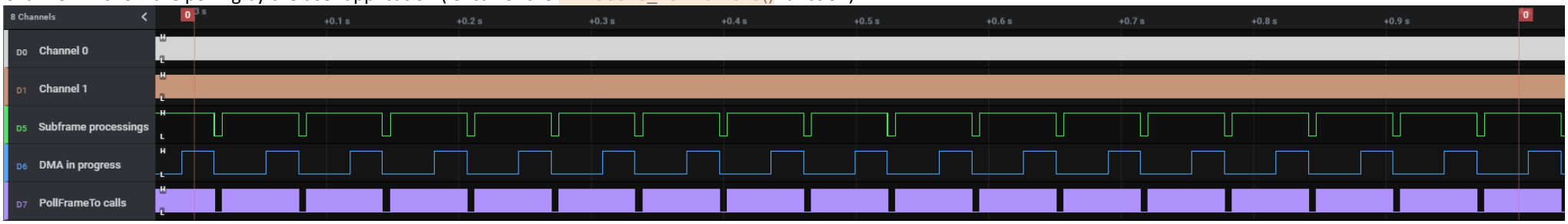
```
        if ((Status & TWIHS_SR_RXRDY) > 0) break;
      }

      *data = I2C->TWIHS_RHR;                                  // Get next data byte
      data++;
      RemainingBytes--;
    }
  }
  if (stop) while ((I2C->TWIHS_SR & TWIHS_SR_TXCOMP) == 0);    // Wait until both holding register and internal
                                                               // shifter are empty and STOP condition has been sent
  return ERR_OK;
}
```

Example of driver interface handle functions in a .h file:

```
/*! @brief Get millisecond
 *
 * This function will be called when the driver needs to get current millisecond
 */
uint32_t GetCurrentms_V71(void);


//*****************************************************************************************************************

/*! @brief Hardware I2C driver interface configuration for the ATSAMV71
 *
 * This function will be called at driver initialization to configure the interface driver soft I2C
 * @param[in] *pIntDev Is the EERAM47x16.InterfaceDevice of the device that call this function
 * @param[in] sclFreq Is the SCL frequency in Hz to set at the interface initialization
 * @return Returns an #eERRORRESULT value enum
 */
eERRORRESULT HardI2C_InterfaceInit_V71(void *pIntDev, const uint32_t sclFreq);


/*! @brief Hardware I2C - Transfer data through an I2C communication for the ATSAMV71
 *
 * This function will be called when the driver needs to transfer data over the I2C communication with the device
 * Can be read data of transmit data. It also indicate if it needs a start and/or a stop
 * @param[in] *pIntDev Is the Interface Device pointer of the device that call the I2C transfer
 * @param[in] deviceAddress Is the device address on the bus (8-bits only). The LSB bit indicate if it is a I2C Read
(bit at '1') or a I2C Write (bit at '0')
 * @param[in,out] *data Is a pointer to memory data to write in case of I2C Write, or where the data received will be
stored in case of I2C Read (can be NULL if no data transfer other than chip address)
 * @param[in] byteCount Is the byte count to write over the I2C bus or the count of byte to read over the bus
 * @param[in] start Indicate if the transfer needs a start (in case of a new transfer) or restart (if the previous
transfer have not been stopped)
 * @param[in] stop Indicate if the transfer needs a stop after the last byte sent
 * @return Returns an #eERRORRESULT value enum
 */
eERRORRESULT HardI2C_Tranfert_V71(void *pIntDev, const uint8_t deviceAddress, uint8_t *data, size_t byteCount, bool
start, bool stop);
```

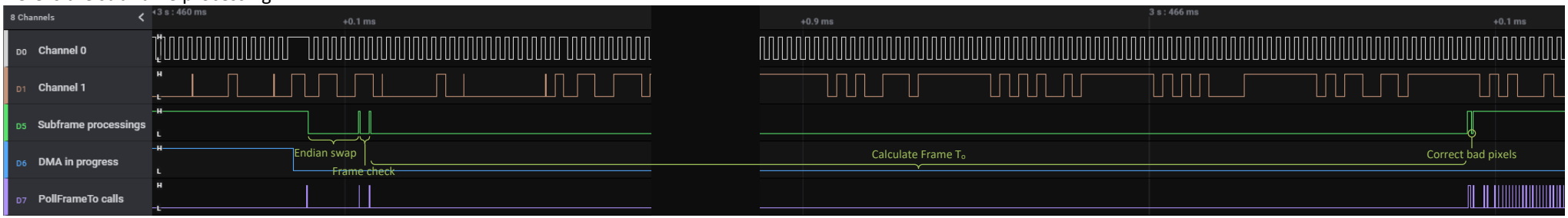
14. CHRONOGRAM OF DMA TRANSFER (ASYNCHRONOUS)

Here after use the configuration indicated in §8. Between the 2 red markers, there are 16 subframe processing. Active state is low for channels D5, D6 and D7.
Channel D5 show subframe processing like `MLX90640_BigEndianToLittleEndian()`, check of the received data, `MLX90640_CalculateTo()`, and `MLX90640_CorrectBadPixels()` functions.
Channel D6 show the DMA use. When at low level, that means the DMA is working.
Channel D7 show the polling by the user application (ie. call of the `MLX90640_PollFrameTo()` function).



As shown, the CPU process subframe while DMA is in progress.

Here is the subframe processing:



The DMA takes less that 40ms to retrieve a frame with a I2C clock at 400kHz: