I2C interface

Guide

Visibility and dissemination of the document:

Can be widely distributed

Language: **C**
Language version: **C99**
Endianness: **N/A**

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

MCU compatibility:
no limit

Tests: **N/A**

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

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

I2C interface

Version: 1.1.1 date: 26 Nov 2023

This I2C interface definitions for all the https://github.com/Emandhal drivers and developments.

Established by		Reviewed		Approved	
Name:	**FMA**	Name:	FMA	Name:	FMA
Date:	**26 Nov 2023**	Date:	26 Nov 2023	Date:	26 Nov 2023

Change history

Issue	Date	Nature / comment	Paragraph	Writer
1.1.1	26 Nov 2023	Add STM32cubeIDE	6.1.2	FMA
1.1.0	18 Nov 2023	Add Arduino	6.1.1	FMA
1.0.0	02 Oct 2021	**Initial release**	-	FMA

Content

Figure summary

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 to use the definitions to create I2C interfaces for https://github.com/Emandhal drivers and developments. All drivers that use I2C will use theses interfaces.

The interface:

- Can be use with any MCU (little or big endian)
- Only take care of the I2C interface entries
- All functions are identical which guarantees that all drivers will communicates with the I2C driver the same way

1.2. Documents, References, and abbreviations

1.2.1. Applicable documents

IDENTIFICATION	TITLE	DATE
-	-	-

1.2.2. Reference documents

IDENTIFICATION	TITLE	DATE
-	-	-

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 Memory Access
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 interface has been designed to:

- Use only one entry for both blocking and non-blocking (with interrupts or DMA) mode
- The driver can ask for an endianness transformation
- The driver can use the result of the interface to correct/change the transfer
- Create virtual I2C ports and link a driver to it

3. PRESENTATION

This interface is the entry point of an I2C driver. The driver will point to the I2C Initialization's function of the I2C driver to configure it, and the driver will point to the I2C Transfer's function of the I2C driver to transfer data.

Example with driver which communicate directly through a hardware I2C peripheral:

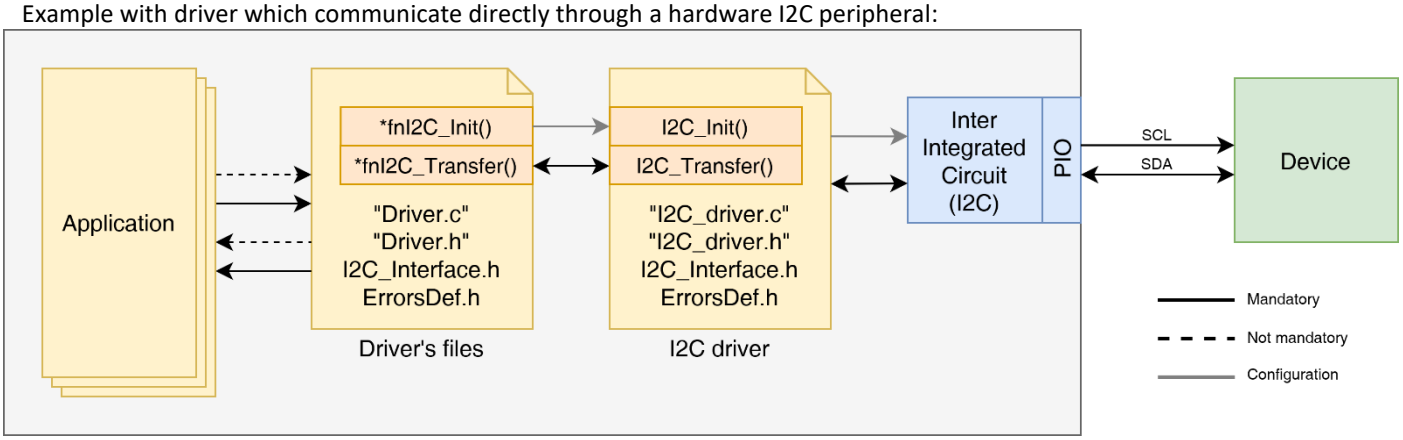


Figure 1 – Interface use with a hardware I2C peripheral

Example with driver which communicate through a software I2C interface:

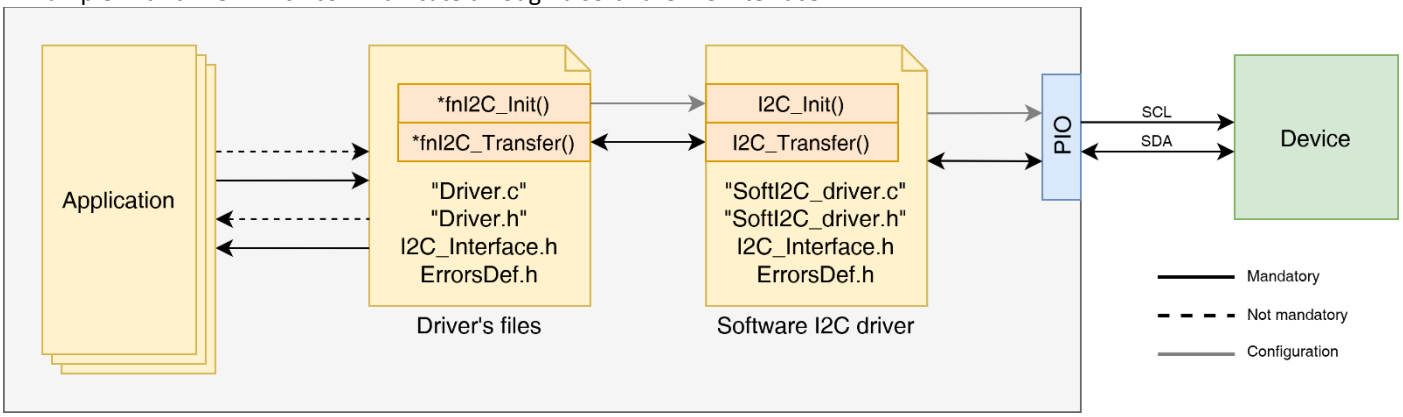


Figure 2 – Interface use with a software I2C interface

Example with a driver which communicate through a SPI to I2C device:

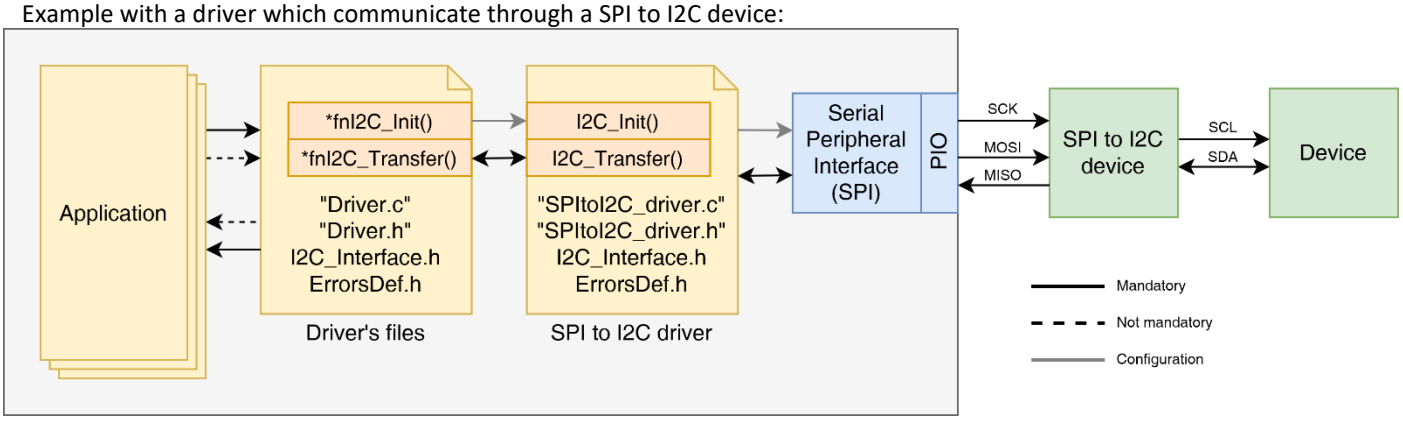


Figure 3 – Interface use with a SPI to I2C device

4. INTERFACE FILE CONFIGURATION

There is no Interface File specific configuration.

5. HOW TO DESIGN A DRIVER THAT USE THIS INTERFACE AS INPUT

Depending on device or peripheral that will be in use, the implementation will differ.

The I2CInterface_Packet.Config indicates to the driver what it can do with the packet to help the driver but may not be used. If not used, the I2C driver will be compatible with all of device's drivers. This configuration is design to be discarded without any problems.

On some device drivers some transformation can be done directly by the I2C driver to reduce the amount of CPU used by the driver, like asking the I2C driver to take care of the endianness transformation or the use of DMA because the driver work with non-blocking mode.

5.1. Basic use of the I2C_Interface structure

In case of a basic use of a I2C peripheral, the I2CInterface_Packet.Config is not used, and it will be compatible with all of drivers. This configuration is design to be discarded without any problems. The driver design should look like this:

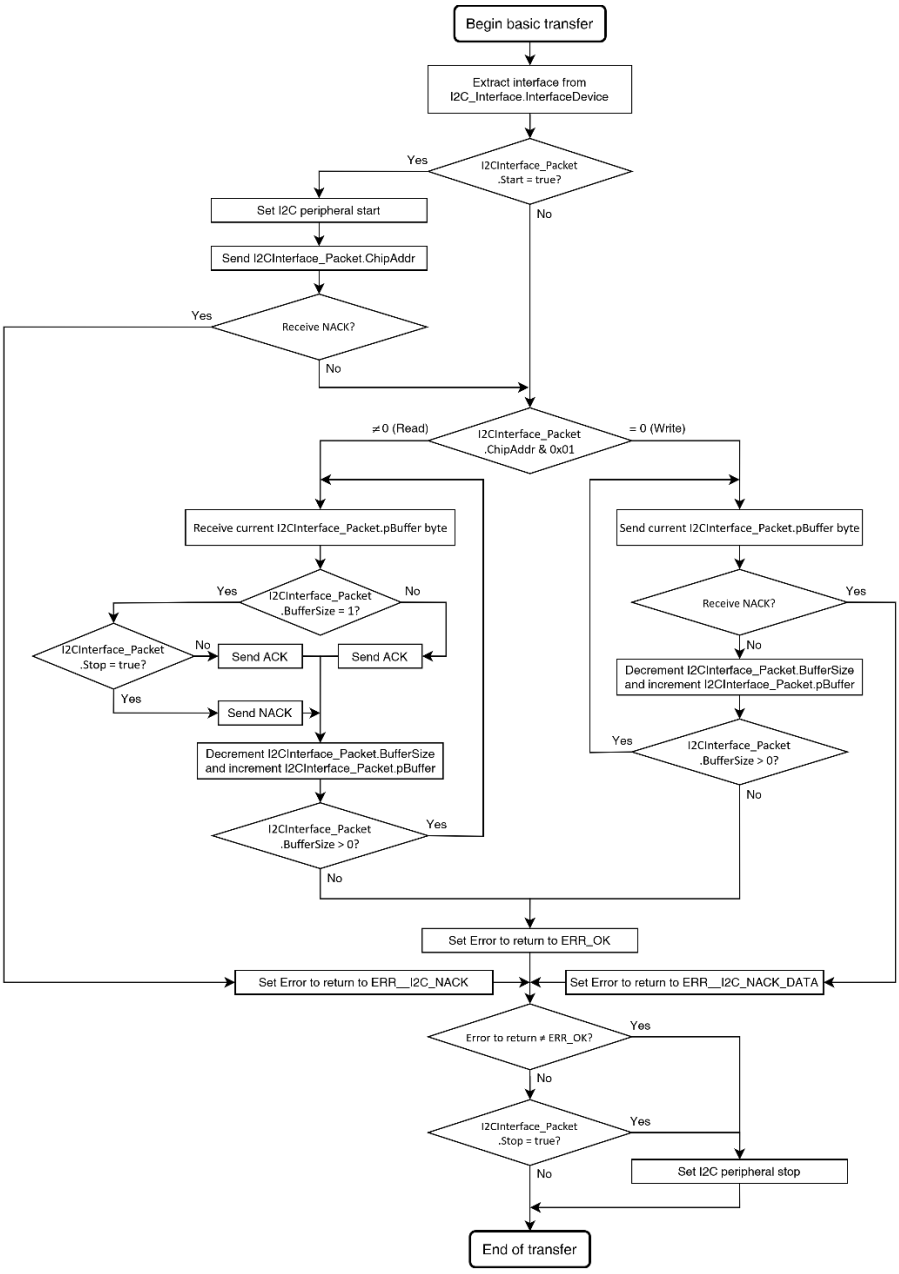


Figure 4 - Basic transfer diagram

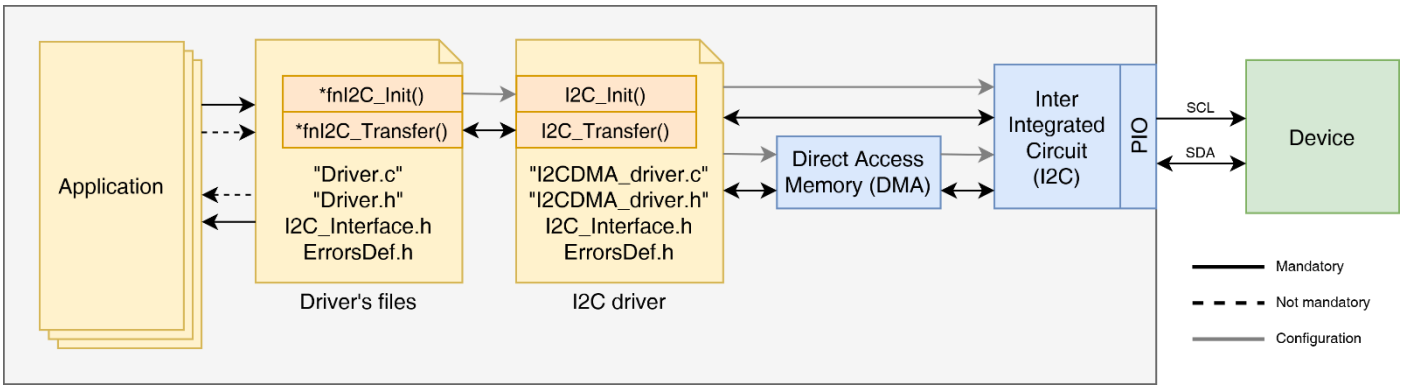
5.1.1. Example

Example of a driver .c file:

```
static bool I2Cconfigured = false;
static bool I2CstopSent   = true;



//===========================================================================
// Soft I2C driver interface configuration
//===========================================================================
eERRORRESULT SoftI2C_InterfaceInit(I2C_Interface *pIntDev, const uint32_t sclFreq)
{
  if (pIntDev == NULL) return ERR__I2C_PARAMETER_ERROR;
  if (I2Cconfigured) return ERR_OK;

  //--- Pin configuration of the soft I2C ---
  SDA_SOFT_PIO_En;
  SDA_SOFT_High;   // SDA = 1
  SDA_SOFT_In;     // SDA in

  SCL_SOFT_PIO_En;
  SCL_SOFT_High;   // SCL = 1
  SCL_SOFT_Out;    // SCL out

  //--- Bus initialization/recovery ---
  size_t z = 9;                // Clock up to 9 cycles. Here we force I2C SCL to clock until a device stuck in communication respond
  while (SDA_SOFT_Status == 0) // Look for SDA high in each cycle while SCL is high and then break
  {
    delay_us(1);
    SCL_SOFT_Low;  // SCL = 0
    delay_us(1);
    SCL_SOFT_High; // SCL = 1
    if (--z == 0) break;
  }

  I2Cconfigured = true;
  I2CstopSent   = true;
  return ERR_OK;
}



//===========================================================================
// Software I2C - Transfer data through an I2C communication
//===========================================================================
eERRORRESULT SoftI2C_Transfer(I2C_Interface *pIntDev, I2CInterface_Packet* const pPacketDesc)
{
  eERRORRESULT Error = ERR_OK;
  bool DeviceReady = true;
  bool ForceStop   = false;
  bool DeviceWrite = ((pPacketDesc->ChipAddr & 0x01) == 0);

  //--- Transfer data ---
  if (pPacketDesc->Start)
  {
    Error = SoftI2C_Start(pIntDev, pPacketDesc->ChipAddr); // Send a start if asked
    if (Error == ERR__I2C_NACK) DeviceReady = false;       // If the device receive a NAK, then the device is not ready
    if (Error != ERR_OK) ForceStop = true;                 // If there is an error while starting the transfer then
  }
  if (ForceStop == false)
  {
    if (DeviceWrite) // Device write
    {
      while (pPacketDesc->BufferSize > 0)
      {
        Error = SoftI2C_TxByte(pIntDev, *pPacketDesc->pBuffer); // Transmit byte
        if (Error == ERR__I2C_NACK) Error = ERR__I2C_NACK_DATA; // If we receive a NACK here then it is a NACK on data transfer
        if (Error != ERR_OK) { ForceStop = true; break; }       // If there is an error while receiving data from I2C then stop the
transfer
        pPacketDesc->pBuffer++;
        pPacketDesc->BufferSize--;
      }
    }
    else // Device read
    {
      while (pPacketDesc->BufferSize > 0)
      {                                                 // Receive byte and send ACK if not last byte to receive and stop else send NACK
        Error = SoftI2C_RxByte(pIntDev, pPacketDesc->pBuffer, !(pPacketDesc->Stop && (pPacketDesc->BufferSize == 1)));
        if (Error != ERR_OK) { ForceStop = true; break; } // If there is an error while receiving data from I2C then stop the transfer
        pPacketDesc->pBuffer++;
        pPacketDesc->BufferSize--;
      }
    }
  }
```

```
  if (pPacketDesc->Stop || ForceStop)
  {
    //--- Stop transfer ---
    eERRORRESULT ErrorStop = SoftI2C_Stop(pIntDev); // Stop I2C
    if (DeviceReady == false) return ERR__I2C_NACK;
    Error = (Error != ERR_OK ? Error : ErrorStop);
  }

  return Error;
}
```

5.2. DMA use of the I2C_Interface structure



To know if the driver that ask for a transfer needs to use a DMA for this transfer, the user have to check if `I2CInterface_Packet.Config.Bits.IsNonBlocking` is set to '1'. The driver design should look like this:

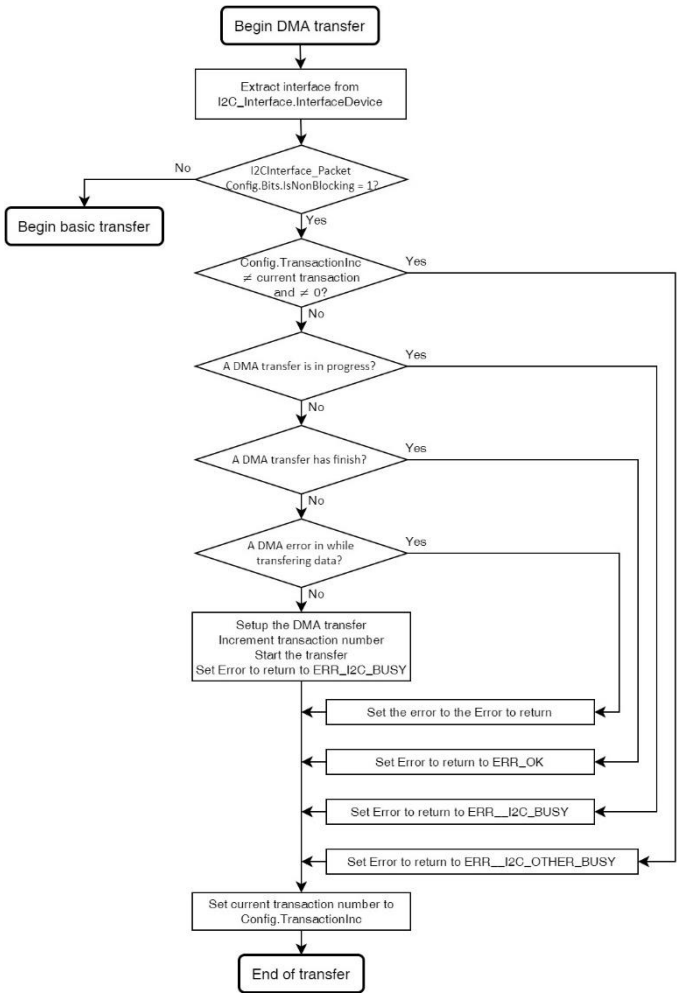


Figure 5 - DMA transfer diagram

5.3. Specific use of the I2C_Interface structure

There are others configurations to that some drivers can ask to adapt for specific use or to reduce CPU consumption.

5.3.1. Switch endianness

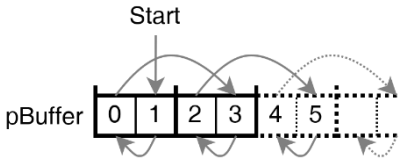
Some devices communicate with data with big-endian, on CPU that use little-endianness, the user or driver need to perform the switch of the endianness which consumes CPU time on big amount of data.

The I2CInterface_Packet.Config.Bits.EndianTransform indicate which transformation need to be performed. This transformation can be quickly applied on basic transfers with almost no CPU change. On DMA transfers, the transformation can be done only on complex DMA with data striding. At the end of transfer, set the I2CInterface_Packet.Config.Bits.EndianResult with the same value as I2CInterface_Packet.Config.Bits.EndianTransform to indicate to the driver that asks for this transfer that the endian transformation have been performed. If the transformation have not been performed, leave the I2CInterface_Packet.Config.Bits.EndianResult to 0 (I2C_NO_ENDIAN_CHANGE).

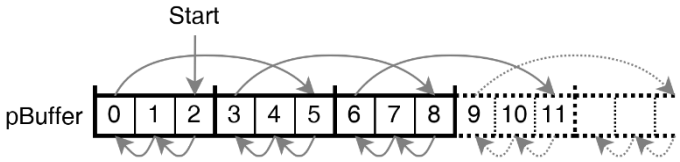
5.3.1.1. Switch endianness on basic transfers

First of all, the user shall verify that the size of buffer is a multiple of the byte size of the endian transformation. The byte size of the endian transformation can be extracted from the value of I2CInterface_Packet.Config.Bits.EndianTransform which is the block size of the transfer.

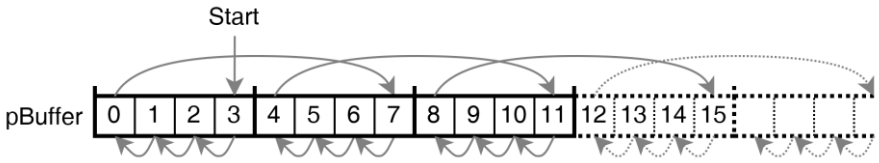
For I2C_SWITCH_ENDIAN_16BITS, this is how the address for the pBuffer should walk to perform the endianness transformation:



For I2C_SWITCH_ENDIAN_24BITS, this is how the address for the pBuffer should walk to perform the endianness transformation:



For I2C_SWITCH_ENDIAN_32BITS, this is how the address for the pBuffer should walk to perform the endianness transformation:



5.3.1.2. Switch endianness on DMA transfers

In this case, refer to the DMA controller of the device MCU/CPU datasheet. There is no specific algorithm to do that, all depends on the DMA controller and the DMA transfer configuration.

5.3.2. Transfer type

The transfer type is for some specific devices like SC18IS600 series from Texas Instrument. To perform write then read or write then write transfers, these devices need to know which part of the transfer they have to perform. That's what I2CInterface_Packet.Config.Bits.TransferType is used for. On basic transfer on standard devices, this is not used.

6. CONFIGURATION STRUCTURES

6.1. Interface object structure

The I2C_Interface object structure contains all information that is mandatory to communicate with a I2C peripheral.

6.1.1. Arduino I2C interface container structure

Source code:

```
typedef struct I2C_Interface I2C_Interface;

struct I2C_Interface
{
  TwoWire &_I2Cclass;
  I2CInit_Func fnI2C_Init;
  I2CTransferPacket_Func fnI2C_Transfer;
};
```

6.1.1.1. Data fields

TwoWire **&_I2Cclass**

Arduino I2C class.

I2CInit_Func **fnI2C_Init**

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

Type

typedef eERRORRESULT (*I2CInit_Func)(I2C_Interface *, const uint32_t)

Initial value / default

This function must point to a function else a ERR__PARAMETER_ERROR is returned by the driver's functions that need to use it.

I2CTransferPacket_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 (*I2CTransferPacket_Func)(I2C_Interface *, I2CInterface_Packet* const)

Initial value / default

This function must point to a function else an ERR__PARAMETER_ERROR is returned by the driver's functions that need to use it.

6.1.2. STM32cubeIDE I2C interface container structure

Source code:

```
typedef struct I2C_Interface I2C_Interface;

struct I2C_Interface
{
  I2C_HandleTypeDef *pHI2C;
  I2CInit_Func fnI2C_Init;
  I2CTransferPacket_Func fnI2C_Transfer;
  Uint32_t I2Ctimeout;
};
```

6.1.2.1. Data fields

I2C_HandleTypeDef ***pHI2C**

Pointer to I2C handle Structure definition.

I2CInit_Func **fnI2C_Init**

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

Type

typedef eERRORRESULT (*I2CInit_Func)(I2C_Interface *, const uint32_t)

Initial value / default

This function must point to a function else a ERR__PARAMETER_ERROR is returned by the driver's functions that need to use it.

I2CTransferPacket_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 (*I2CTransferPacket_Func)(I2C_Interface *, I2CInterface_Packet* const)

Initial value / default

This function must point to a function else an ERR__PARAMETER_ERROR is returned by the driver's functions that need to use it.

uint32_t **I2Ctimeout**

I2C timeout.

6.1.3. Generic I2C interface container structure

Source code:

```
typedef struct I2C_Interface I2C_Interface;

struct I2C_Interface
{
  void *InterfaceDevice;
  uint32_t UniqueID;
  I2CInit_Func fnI2C_Init;
  I2CTransferPacket_Func fnI2C_Transfer;
  uint8_t Channel;
};
```

6.1.3.1. Data fields

void ***InterfaceDevice**

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

uint32_t **UniqueID**

This is a protection for the #InterfaceDevice pointer. This value will be check when using the struct I2C_Interface in the driver which use the generic I2C interface.

I2CInit_Func **fnI2C_Init**

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

Type

typedef eERRORRESULT (*I2CInit_Func)(I2C_Interface *, const uint32_t)

Initial value / default

This function must point to a function else a ERR__PARAMETER_ERROR is returned by the driver's functions that need to use it.

I2CTransferPacket_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 (*I2CTransferPacket_Func)(I2C_Interface *, I2CInterface_Packet* const)

Initial value / default

This function must point to a function else an ERR__PARAMETER_ERROR is returned by the driver's functions that need to use it.

uint8_t **Channel**

I2C channel of the interface device.

6.1.4. Driver interface handle functions

```
eERRORRESULT (*I2CInit_Func)(
        I2C_Interface *pIntDev,
        const uint32_t sclFreq)
```

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

Parameters

Input	*pIntDev	Is the I2C interface container structure used for 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 in the I2C driver. It is recommended, during the implement of the pointer interface function, to return only errors listed in §6.3, and when all when fine, return `ERR_OK`.

```
eERRORRESULT (*I2CTransferPacket_Func)(
        I2C_Interface *pIntDev,
        I2CInterface_Packet* const pPacketDesc)
```

Interface packet function for I2C peripheral transfer. This function will be called when the driver needs to transfer data over the I2C communication with the device. The packet gives the possibility to set the address/command where the data will be stored/recall.

Parameters

Input	*pIntDev	Is the I2C interface container structure used for the communication
Input	pPacketDesc	Is the packet description to transfer through I2C

Return

Returns an `eERRORRESULT` value enumerator dependent of how the return error is implemented by the user in the I2C driver. It is recommended, during the implement of the pointer interface function, to return only errors listed in §6.3, and when all when fine, return `ERR_OK`.

6.2. I2C packet object structure

This is the descriptor of the I2C packet to transfer.

Source code:

```
typedef struct
{
  I2C_Conf Config;
  bool Start;
  uint8_t ChipAddr;
  uint8_t* pBuffer;
  size_t BufferSize;
  bool Stop;
} I2CInterface_Packet;
```

6.2.1. Data fields

`I2C_Conf` **Config**

Configuration of the I2C transfer.

Type

union `I2C_Conf`

`bool` **Start**

Indicate if the transfer needs a start (in case of a new transfer) or restart (if the previous transfer has not been stopped) in case of 'true' else this transfer does not need a start.

`uint8_t` **ChipAddr**

I2C slave chip address to communicate with.

*uint8_t** **pBuffer**

In case of write, this is the bytes to send to slave chip, in case of read, this is where data read will be stored.

size_t **BufferSize**

Buffer size in bytes.

bool **Stop**

Indicate if the transfer needs a stop after the last byte sent by this function call.

6.2.2. Structures and unions

This is the I2C configuration to apply to the packet to transfer. Source code:

```
typedef union I2C_Conf
{
  uint16_t Value;
  struct
  {
    uint16_t TransferType   :  3;
    uint16_t IsNonBlocking  :  1;
    uint16_t EndianResult   :  3;
    uint16_t EndianTransform:  3;
    uint16_t TransactionInc :  6;
    uint32_t                : 15;
    uint32_t Addr10bits     :  1;
  } Bits;
} I2C_Conf;
```

6.2.3. Data fields

uint16_t **Value**

This is the value of I2C_Conf.

eI2C_TransferType **Bits.TransferType**:3

This is the transfer type of the packet.

bool **Bits.IsNonBlocking**:1

Non-blocking use for the I2C: '1' = The driver ask for a non-blocking transfer (with DMA or interrupt transfer) ; '0' = The driver ask for a blocking transfer.

eI2C_EndianTransform **Bits.EndianResult**:3

If the transfer changes the endianness, the peripheral that do the transfer will say it here.

eI2C_EndianTransform **Bits.EndianTransform**:3

The driver that asks for the transfer needs an endian change from little to big-endian or big to little-endian.

uint8_t **Bits.TransactionInc**:6

Current transaction number (managed by the I2C+DMA driver). When a new DMA transaction is initiate, set this value to '0', the I2C+DMA driver will return an incremental number. This is for knowing that the transaction has been accepted or the bus is busy with another transaction.

uint32_t **Bits.Addr10bits**:1

Chip address length: '1' = 10-bits address ; '0' = 8-bits address.

6.2.4. Enumerators

enum **eI2C_TransferType**

Transfer type of the packet.

Enumerator

I2C_SIMPLE_TRANSFER	**0b000**	This packet is a simple transfer
I2C_WRITE_THEN_READ_FIRST_PART	**0b001**	This packet is the first part of a dual transfer (with restart). Transfer will be a write then read
I2C_WRITE_THEN_READ_SECOND_PART	**0b010**	This packet is the second part of a dual transfer (with restart). Transfer will be a write then read
I2C_WRITE_THEN_WRITE_FIRST_PART	**0b101**	This packet is the first part of a dual transfer (with restart). Transfer will be a write then write
I2C_WRITE_THEN_WRITE_SECOND_PART	**0b110**	This packet is the second part of a dual transfer (with restart). Transfer will be a write then write

enum **eI2C_EndianTransform**

Transfer type of the packet.

Enumerator

I2C_NO_ENDIAN_CHANGE	**0x0**	Do not change endianness therefore read/write byte at the same order as received/sent
I2C_SWITCH_ENDIAN_16BITS	**0x2**	Switch endianness per read/write 16-bits data received/sent
I2C_SWITCH_ENDIAN_24BITS	**0x3**	Switch endianness per read/write 24-bits data received/sent
I2C_SWITCH_ENDIAN_32BITS	**0x4**	Switch endianness per read/write 32-bits data received/sent

6.3. 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__I2C_NACK	**210**	Received a I2C not acknowledge
ERR__I2C_NACK_ADDR	**211**	Received a I2C not acknowledge while transferring address
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