



## STM32F10xx, STM32F2xx, STM32F4xx and STM32L1xx I2C Communication peripheral application library (CPAL)

### Introduction

The purpose of this document is to explain the architecture and the implementation of the I2C CPAL (Communication Peripheral Application Library).

CPAL is a library providing high layer API for I2C peripherals. It provides CPAL drivers , a project template allowing users to build a new project from scratch, and some examples showing how to use and customize the CPAL drivers.

An associated firmware package is provided with this document and it is available for download from the STMicroelectronics website: [www.st.com](http://www.st.com).

*Table 1* lists the microcontroller products concerned by this user manual.

**Table 1. Applicable products**

Type	Part numbers and product series
Microcontrollers	STM32F10xx STM32F2 Hi-performance STM32L151xx, STM32L152xx and STM32L162xx STM32F405xx, STM32F407xx, STM32F415xx and STM32F417xx STM32F427xx and STM32F437xx

# Contents

- 1 CPAL overview ..... 6**
- 2 CPAL architecture description ..... 7**
  - 2.1 CPAL application hierarchy ..... 7
  - 2.2 Communication layer ..... 10
    - 2.2.1 CPAL main structures (cpal.h) ..... 11
    - 2.2.2 CPAL communication functions (cpal\_i2c.c/h) ..... 19
  - 2.3 User application interface ..... 20
    - 2.3.1 Configuration interface (cpal\_conf.h) ..... 21
    - 2.3.2 User callback interface (cpal\_usercallback.c) ..... 25
  - 2.4 Low layer interface (hardware abstraction layer HAL) ..... 27
- 3 CPAL functional description ..... 31**
  - 3.1 Configuration ..... 31
    - 3.1.1 CPAL\_I2C\_Init() functional description ..... 31
    - 3.1.2 CPAL\_I2C\_DeInit() functional description ..... 32
    - 3.1.3 CPAL\_I2C\_StructInit() functional description ..... 34
  - 3.2 Communication ..... 34
    - 3.2.1 CPAL\_I2C\_Read() functional description ..... 36
    - 3.2.2 CPAL\_I2C\_Write() functional description ..... 37
    - 3.2.3 CPAL\_I2C\_Listen () functional description: ..... 37
    - 3.2.4 CPAL\_I2C\_IsDeviceReady() functional description ..... 39
    - 3.2.5 CPAL interrupts and DMA management ..... 39
  - 3.3 Event and error management (user callbacks) ..... 41
    - 3.3.1 Timeout management ..... 43
- 4 How to use and customize the CPAL library (step by step) ..... 46**
  - 4.1 Basic configuration ..... 46
    - 4.1.1 Select peripherals to be used ..... 46
    - 4.1.2 Configure transfer options ..... 47
    - 4.1.3 Select and configure user and error callbacks ..... 47
    - 4.1.4 Configure timeout management ..... 47
    - 4.1.5 Set Events, Errors and DMA interrupt priorities ..... 48
    - 4.1.6 Configure the Log Macro ..... 48

---

4.2	Structure initialization .....	49
4.3	Communication .....	49
4.4	Error management .....	50
4.5	Advanced configuration .....	51
4.5.1	Select peripheral I/O pins .....	51
4.5.2	Select TX and RX DMA channels .....	51
4.5.3	Set event, error and DMA interrupt priorities .....	51
<b>5</b>	<b>CPAL implementation example (step by step) .....</b>	<b>53</b>
5.1	Starting point .....	53
5.2	cpal_conf.h .....	53
5.3	cpal_usercallback.c .....	55
5.4	main.c .....	56
<b>6</b>	<b>CPAL Examples .....</b>	<b>59</b>
6.1	Basic EEPROM example .....	60
6.2	Two-board example .....	60
6.3	Advanced example .....	61
6.4	Two-board Listen mode example .....	62
<b>7</b>	<b>Memory footprint of CPAL components .....</b>	<b>64</b>
<b>8</b>	<b>FAQ .....</b>	<b>66</b>
<b>9</b>	<b>Revision history .....</b>	<b>69</b>

# List of tables

Table 1.	Applicable products	1
Table 2.	CPAL file descriptions	10
Table 3.	CPAL_InitTypeDef structure	12
Table 4.	CPAL_Dev field values	13
Table 5.	CPAL_Direction field values	13
Table 6.	CPAL_Mode field values	14
Table 7.	CPAL_ProgModel field values	14
Table 8.	CPA_TransferTypeDef structure fields	14
Table 9.	CPAL_State field values	15
Table 10.	wCPAL_DevError field values	15
Table 11.	CPAL option fields	17
Table 12.	wCPAL_Options field values	18
Table 13.	Architecture of CPAL Communication Layer functions	19
Table 14.	CPAL Communication Layer function list	19
Table 15.	CPAL configuration sections	21
Table 16.	CPAL configuration sections	26
Table 17.	HAL configuration sections	28
Table 18.	CPAL low layer interface function description	29
Table 19.	CPAL_I2C_Struct_Init() default values	34
Table 20.	I2C interrupt management order	40
Table 21.	DMA interrupt management order	40
Table 22.	CPAL I2C user callback list	41
Table 23.	Hardware resources used in CPAL examples	59
Table 24.	Memory footprint of CPAL components	64
Table 25.	FAQ	66
Table 26.	Document revision history	69

## List of figures

Figure 1.	CPAL library architecture . . . . .	7
Figure 2.	CPAL library hierarchy . . . . .	9
Figure 3.	CPAL_I2C_Init() function flowchart. . . . .	32
Figure 4.	CPAL_I2C_DelInit() function flowchart . . . . .	33
Figure 5.	CPAL_I2C_Read() function flowchart. . . . .	36
Figure 6.	CPAL_I2C_Write() function flowchart . . . . .	37
Figure 7.	CPAL_I2C_Listen () function flowchart. . . . .	38
Figure 8.	CPAL I2C timeout manager flowchart . . . . .	45
Figure 9.	Basic EEPROM example architecture . . . . .	60
Figure 10.	Two-board example architecture . . . . .	61
Figure 11.	Advanced example architecture . . . . .	62

# 1 CPAL overview

The main purpose of CPAL is to:

- Provide an intuitive, easy to use and practical API (Init, Deinit, Read, Write):
  - All configurations needed for the I2C communication peripheral (I/O pins, clocks, interrupt vectors, DMA channels ...) are internally managed by the CPAL low layer drivers.
  - The communication operations are also managed internally by the CPAL drivers (communication headers, address sending, interrupt and DMA control, error management ...).
  - All operations are controlled and monitored through a single configuration structure (one instance of the structure for each peripheral) holding all necessary configuration parameters (peripheral configuration, buffer addresses and sizes...) as well as the current communication status and error codes.
- Provide efficient and complete management of peripheral and communication errors. Peripheral events and errors are managed by the CPAL low layer drivers and allow you to integrate your specific code easily for each event and error. Communication errors are also managed by a timeout mechanism that can be customized by the user application.
- Provide high level customization and integration features:
  - Several static configurations allow code size reduction when some options are not used (for example, control of the number of used peripherals, disabling the management of some modes: 10-bit addressing, General call ...).
  - Dynamic and easy configuration through a unique control structure allowing peripheral and communication options to be dynamically enabled/disabled.
  - Several user callbacks: functions declared and called by the CPAL drivers and implemented by the user application when needed. These functions allow the user application to perform specific actions related to specific communication events/errors.
- Provide a device abstraction layer: CPAL library supports all STM32 device families.
- Provide efficient and simple debug features through the CPAL\_DEBUG option: debug messages are sent through a customizable macro. Log messages are sent at each step of the CPAL driver (this macro can be customized to send messages through a serial interface, debug IDE interface, LCD screen...).
- In order to optimize CPAL driver performance, all operations, except device initialization, are performed through direct register access.

## 2 CPAL architecture description

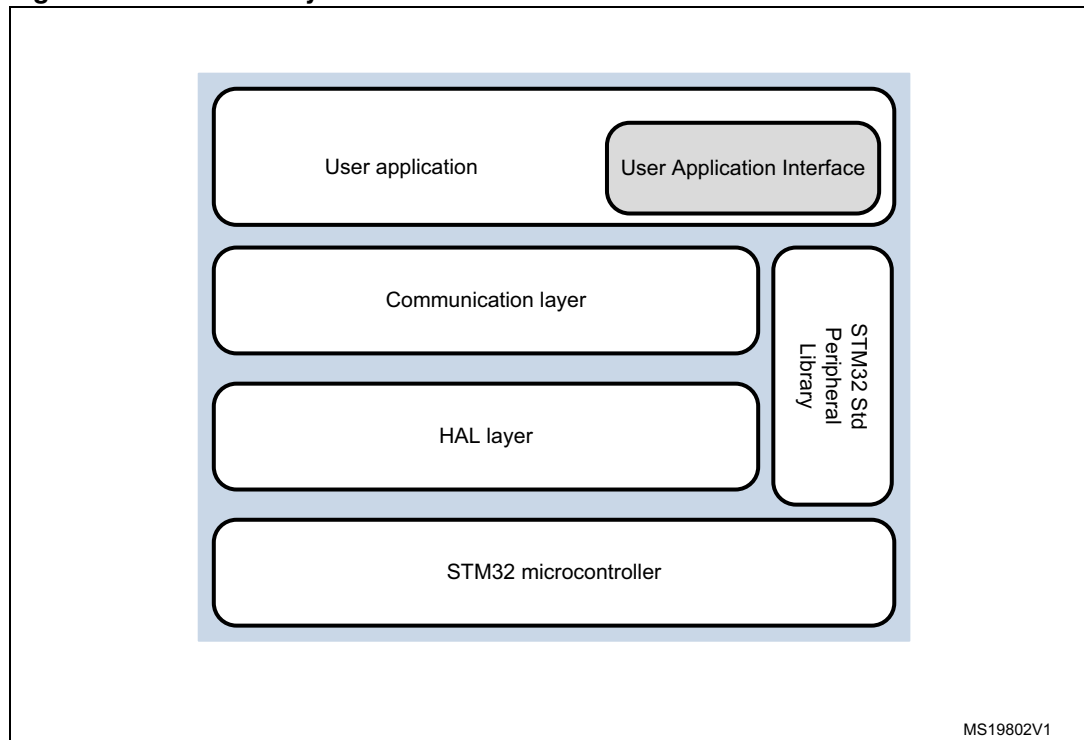
### 2.1 CPAL application hierarchy

The CPAL library consists of three layers:

- User application interface: allows the customization of CPAL library and implementation of user callbacks. This layer consists of files which you can modify according to your application requirements.
- Communication layer: contains the communication API for each peripheral (I2C).
- HAL layer: is a Hardware Abstraction Layer (HAL) that allows the different device registers to be controlled independently of the device family.

Each layer is described in more detail in the following sections.

**Figure 1. CPAL library architecture**



**Note:** *The CPAL library provides template files for implementing the User Application interface. You can modify these files or not according to your application requirements.*

The CPAL library package consists of three main folders:

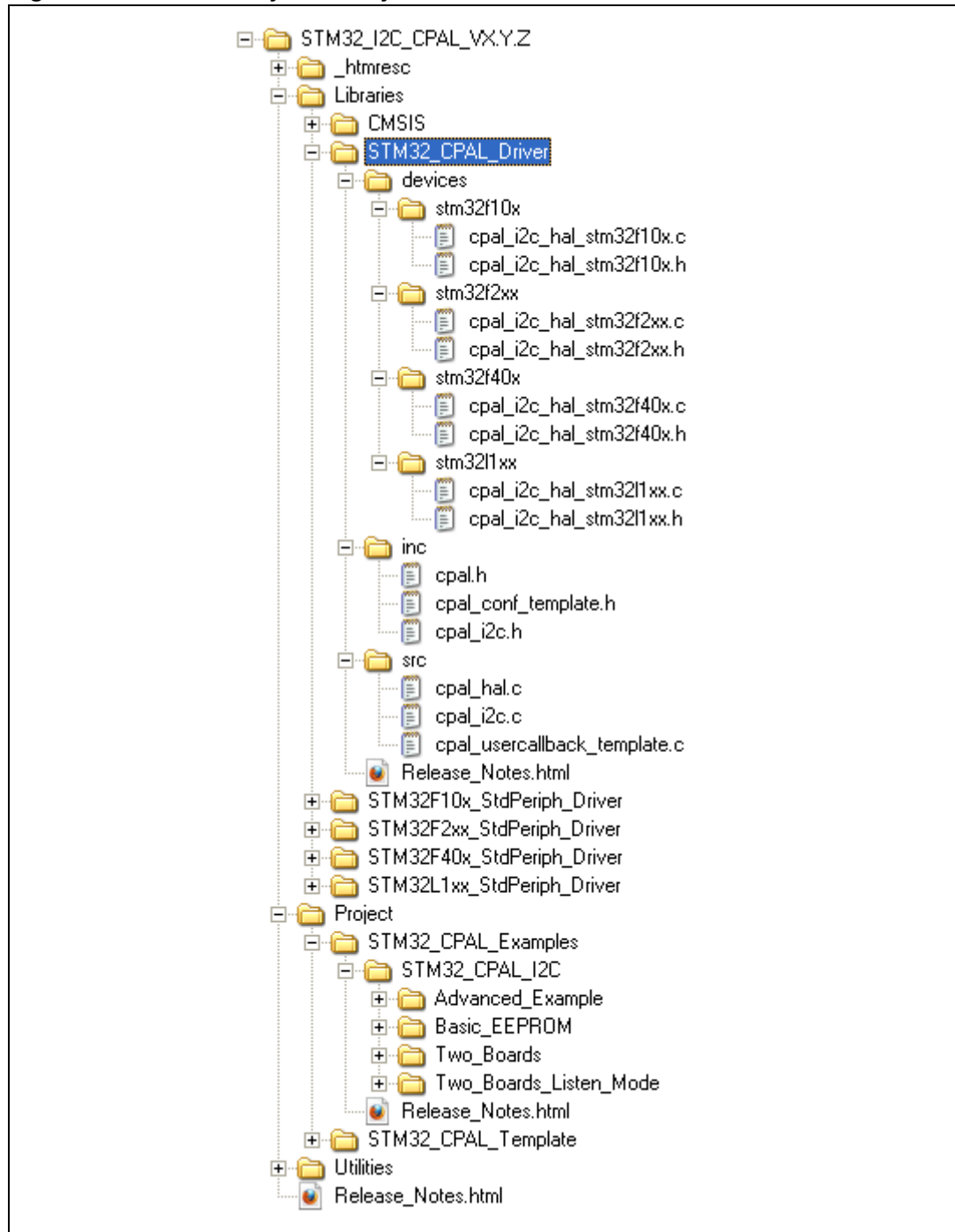
- STM32\_CPAL\_Driver: located in the Libraries folder at the same level as CMSIS and the STM32 standard peripheral drivers. This folder contains all the CPAL

drivers and header files as well as templates for user files (cpal\_conf\_template.h and cpal\_usercallback.c).

- STM32\_CPAL\_Examples: located in the Project folder. This folder contains the examples provided by STMicroelectronics for I2C peripheral showing how to use the CPAL library in a final user application.
- STM32\_CPAL\_Template: located in the Project folder. This folder contains a template (empty) project that allows you to start a project from scratch using the CPAL library drivers.



Figure 2. CPAL library hierarchy



The library files are listed in the following table. They are described in detail in the next sections.

**Table 2. CPAL file descriptions**

Layer	File name	Description
User Application Interface	cpal_conf.h	User file which can be modified to customize and/or configure the CPAL library drivers. A template of this file is provided in the Project/STM32_CPAL_Template project and its implementation is illustrated in each CPAL example.
	cpal_usercallback.c	User file that contains the User Callback implementations when needed. If no callback implementation is needed, this file may be not used (comment out the unused functions).
Communication layer	cpal.h	Contains the main structure definitions and the global CPAL structure declaration (as extern). It also contains all option definitions.
	cpal_i2c.c	This file contains the main operation functions for the peripheral (Init, Delnit, Read, Write...). It also contains all the peripheral-related interrupt handlers (peripheral and DMA interrupts) as well as the error management functions.
	cpal_i2c_.h	This is the header file for the cpal_i2c.c driver. It also contains the peripheral-specific error definitions.
Hardware Abstraction Layer	cpal_hal.c	Common low layer driver for all the peripherals.
	cpal_i2c_hal_stm32xxx x.c	Where xxxx is the device family identifier (for example stm32f10x). This file provides low layer functions specific to the device family for controlling the I2C peripheral.
	cpal_i2c_hal_stm32xxx x.h	Where xxxx is the device family identifier (for example. stm32f10x). This file provides low layer configuration options (IO selection, DMA channel selection, Interrupt configuration ...) as well as low layer macros used for peripheral control. This file may be modified by the user when needed, to use different I/Os, DMA channels ...

## 2.2 Communication layer

The CPAL communication layer interface contains all the needed functions (APIs) that can be called by the user application.

It consists of the following files:

- cpal.h
- cpal\_i2c.c
- cpal\_i2c.h

**Important notes:**

- All I2C interrupt handlers and all the DMA-related interrupt handlers are declared and managed exclusively by the CPAL library. The user application does not need and should not declare these interrupt handlers. If these handlers are needed for other

purposes (for example a DMA interrupt shared by several peripherals...) the user application may use the related callbacks.

- Interrupt priority groups and preemption orders are also managed by the CPAL driver. To configure these parameters, modify the `cpal_conf.h` file.

## 2.2.1 CPAL main structures (cpal.h)

The CPAL library provides a unique structure grouping all parameters needed to:

- Configure a peripheral
- Use it for communication
- Monitor the state of the CPAL driver and the peripheral.

This structure is declared in the `cpal.h` file and is used as the unique argument for all driver functions.

Each peripheral instance has a separate structure holding its configuration parameters and all the related transfer arguments and status. These structures are declared in the driver file and are also declared as extern in the `cpal.h` file (so no need for you to declare them in your application files).

### Example:

```
#ifndef CPAL_USE_I2C1
extern CPAL_InitTypeDef I2C1_DevStructure;
#endif /* CPAL_USE_I2C1 */

#ifndef CPAL_USE_I2C2
extern CPAL_InitTypeDef I2C2_DevStructure;
#endif /* CPAL_USE_I2C2 */
...
```

The CPAL configuration structure is defined as follows:

```
typedef struct
{
CPAL_DevelopmentDef          CPAL_Dev;
CPAL_DirectionTypeDef       CPAL_Direction;
CPAL_ModeTypeDef            CPAL_Mode;
CPAL_ProgramModelTypeDef    CPAL_ProgModel;
CPAL_TransferTypeDef*       pCPAL_TransferTx;
CPAL_TransferTypeDef*       pCPAL_TransferRx;
__IO CPAL_StateTypeDef      CPAL_State;
__IO uint32_t                wCPAL_DevError;
uint32_t                     wCPAL_Options;
__IO uint32_t                wCPAL_Timeout;
I2C_InitTypeDef*            pCPAL_I2C_Struct;
}CPAL_InitTypeDef;
```

The structure fields are detailed in [Table 3](#).

**Table 3. CPAL\_InitTypeDef structure**

Field type	Field name	Description
CPAL_DevTypeDef	CPAL_Dev	This field specifies the peripheral to be configured and controlled by this structure.
CPAL_DirectionTypeDef	CPAL_Direction	This field specifies the transfer directions that are to be supported for the related peripheral (transmission-only, reception-only or both directions). This parameter does not indicate the direction of the current transfer but only the directions supported by the peripheral. Any value listed in <a href="#">Table 5</a> can be assigned to this field.
CPAL_ModeTypeDef	CPAL_Mode	Select the operating mode for the peripheral: Master mode or Slave mode. This mode determines if the peripheral initiates the transfer or waits till it receives transfer data from another master. Any value listed in <a href="#">Table 6</a> can be assigned to this field.
CPAL_ProgModelTypeDef	CPAL_ProgModel	Select the programming model for the next transfers: Interrupt (peripheral transfer interrupts will manage all the transactions and peripheral and driver states) or DMA (all data transfers are managed by DMA channels. CPU is then free to perform other user tasks). Any value listed in <a href="#">Table 7</a> can be assigned to this field. <i>Note: For I2C, when DMA mode is enabled, the addressing phase cannot be managed by DMA but only by interrupts.</i>
CPAL_TransferTypeDef*	pCPAL_TransferTx	This field points to a structure holding all the transmission transfer parameters (buffer addresses and number of data bytes). A value can be assigned to this field as described in <a href="#">Table 8</a> .
CPAL_TransferTypeDef*	pCPAL_TransferRx	This field points to a structure holding all the reception transfer parameters (buffer addresses and number of data bytes). A value can be assigned to this field as described in <a href="#">Table 8</a> .
__IO CPAL_StateTypeDef	CPAL_State	The state field holds the current state of the CPAL driver for the related peripheral instantiated by the CPAL_Dev field. These values are described in more detail in <a href="#">Section 3.3</a> . One of the values listed in <a href="#">Table 9</a> can be assigned to this field. <i>Note: After managing a peripheral error (by clearing the error flag and returning to the transfer correct status), update this field in order to continue normal operations.</i>

**Table 3. CPAL\_InitTypeDef structure (continued)**

Field type	Field name	Description
__IO uint32_t	wCPAL_DevError	This field holds the peripheral-related error code instantiated by the field CPAL_Dev. One of the values listed in <a href="#">Table 10</a> can be assigned to this field. <i>Note: After managing the peripheral error (by clearing the error flag and returning to the transfer correct status), update this field and set it to CPAL_I2C_ERR_NONE in order to continue normal operations.</i>
uint32_t	wCPAL_Options	This field allows you to configure additional options for the transfer configuration. These options are described in more detail in <a href="#">Table 12</a> . Any combination of the specified values can be used for this field (with respect to the conditions related to each option).
__IO uint32_t	wCPAL_Timeout	This field is used for timeout detection. It holds the current value of the timeout counter for the peripheral controlled by this structure.
I2C_InitTypeDef*	pCPAL_I2C_Struct	This field points to a peripheral configuration structure as defined in the standard peripheral library. Only one value can be assigned to this field.

The tables below describe the possible values in detail for each structure field.

**Table 4. CPAL\_Dev field values**

Field value	Description
CPAL_I2Cx	x = peripheral instance (1 or 2 or 3 ...) The possible values for this field depend on the peripherals available in the microcontroller and the enabled defines (CPAL_USE_I2Cx) in the cpal_conf.h file.

**Table 5. CPAL\_Direction field values**

Field value	Description
CPAL_DIRECTION_TX	This value allows only transmission transfers for the selected peripheral.
CPAL_DIRECTION_RX	This value allows only reception transfers for the selected peripheral.
CPAL_DIRECTION_TXRX	This value allows both transmission and reception transfers for the selected peripheral.

**Table 6. CPAL\_Mode field values**

Field value	Description
CPAL_MODE_MASTER	When this value is selected for the peripheral, then it is configured in Master mode and it initiates the transfers (for example, generate the communication clock, the slave address...)
CPAL_MODE_SLAVE	When this value is selected for the peripheral, then it is configured in Slave mode and it waits till a master initiates the transfer.

**Table 7. CPAL\_ProgModel field values**

Field value	Description
CPAL_PROGMODEL_INTERRUPT	When this value is selected for the peripheral, then all data transfer is managed by the peripheral interrupt IRQ handlers (implemented in the CPAL library). The user application then only has to monitor the status of the transfer through the status fields.
CPAL_PROGMODEL_DMA	When this value is selected for the peripheral, then all data transfer is managed by the peripheral-related DMA channel. This is the most optimized transfer mode which allows high transfer rates and frees the CPU for other user application tasks. In this case, the user application has to monitor DMA channel flags or interrupts (CPAL provides appropriate callbacks for managing DMA events). <i>Note: For I2C peripherals, DMA cannot handle 1-byte buffer transfers. So when DMA mode is configured and the buffer size is equal to 1, then DMA mode is disabled and interrupt mode is enabled for this transfer. At the end of the 1-byte transfer, DMA mode is re-enabled and interrupt mode is disabled.</i>

**Table 8. CPA\_TransferTypeDef structure fields**

Field type	Field name	Description
uint8_t*	pbBuffer	This field should contain the pointer to the buffer to be written to or read from. Even when the transfer buffer format is not Byte format, this pointer should be casted to Byte format. The user has to set the value of this field at the start of each transfer. Then the CPAL drivers update it according to the current transfer status. In Interrupt mode: this field is updated (incremented) at each data transmission or reception. In DMA mode: this field is updated only at the end of the current transfer.
uint32_t	wNumData	This field should contain the number of data to be transferred (regardless of their format: Byte or Half-Word or Word). The number of data to be transferred is related only to the peripheral format configuration (8-bit or 16-bit or 32-bit). The user has to set the value of this field at the start of each new transfer. Then the CPAL drivers update it according to the current operations. You can check this field to know how many data have been transferred. In Interrupt mode: this field is updated (decremented) by the CPAL drivers at each data transmission or reception. In DMA mode: this field is updated by the CPAL drivers only at the end of the transfer.

**Table 8. CPA\_TransferTypeDef structure fields (continued)**

Field type	Field name	Description
uint32_t*	wAddr1	In Master mode: this field specifies the address of the slave to communicate with. In Slave mode: This field is not used.
uint32_t*	wAddr2	In Master mode: this field specifies the physical/register address to be written to or read from into the slave (for example for memory devices ...). In Slave mode: this field is not used.

**Table 9. CPAL\_State field values**

Field value	Description
CPAL_STATE_DISABLED	This state is the default state of the CPAL driver. It is set when the related peripheral is disabled (not initialized) and all related resources are free.
CPAL_STATE_READY	This state is set when the related peripheral is initialized and all its resources are assigned.
CPAL_STATE_READY_TX	This state is set when the related peripheral has triggered the communication procedure for transmission.
CPAL_STATE_READY_RX	This state is set when the related peripheral has triggered the communication procedure for reception.
CPAL_STATE_BUSY	This state is set when a write or read operation has been started (but effective transfer on the peripheral has not started yet).
CPAL_STATE_BUSY_TX	This state is set when a transmission transfer is ongoing for the related peripheral.
CPAL_STATE_BUSY_RX	This state is set when a reception transfer is ongoing for the related peripheral.
CPAL_STATE_ERROR	This state is set when an error occurs on the related peripheral or at CPAL driver level. When this state is set, the user application can check the wCPAL_DevError field to determine which error occurred.

**Table 10. wCPAL\_DevError field values**

Field value	Description
CPAL_I2C_ERR_NONE	This is the default state of the error field. It indicates that no peripheral error occurred.
CPAL_I2C_ERR_TIMEOUT	This state indicates that a timeout occurred during the communication or configuration phase. Thus a specific time has elapsed without correct response/event from the peripheral or the slave (in Master mode).

Table 10. wCPAL\_DevError field values (continued)

Field value	Description
CPAL_I2C_ERR_BERR	<p>Bus error: This error occurs when the I2C peripheral detects an external Stop or Start condition during address or data transfer. In this case, the BERR bit is set and an interrupt is generated if the ITERREN bit is set.</p> <p><b>In Slave mode:</b>                      Data are discarded and the lines are released by hardware:</p> <ul style="list-style-type: none"> <li>- In case of a misplaced Start, the Slave considers that it is a restart and it waits for an address, or a Stop condition.</li> <li>- In case of a misplaced Stop, the Slave behaves like for a Stop condition and the lines are released by hardware.</li> </ul> <p><b>In Master mode:</b>                      The lines are not released and the state of the current transmission is not affected. It is up to the software to abort or not the current transmission.</p> <p>Software clearing sequence for the BERR bit 8: write '0' to this bit.</p>
CPAL_I2C_ERR_ARLO	<p>Arbitration lost error: this error occurs when the I2C interface detects an arbitration lost condition. In this case:</p> <ul style="list-style-type: none"> <li>- The ARLO bit is set by hardware (and an interrupt is generated if the ITERREN bit is set). The I2C interface goes automatically back to Slave mode (the M/SL bit is cleared). When the I2C loses the arbitration, it is not able to acknowledge its slave address in the same transfer, but it can acknowledge it after a repeated Start from the “winning” master. Lines are released by hardware.</li> </ul> <p>Software clearing sequence for the ARLO bit 9: write '0' to this bit.</p>
CPAL_I2C_ERR_AF	<p>Acknowledge failure: This error occurs when the interface detects a non-acknowledge bit. In this case:</p> <ul style="list-style-type: none"> <li>- The AF bit is set and an interrupt is generated if the ITERREN bit is set. A transmitter which receives a NACK must reset the communication:</li> <li>- In Slave mode: lines are released by hardware.</li> <li>- In Master mode: a Stop or repeated Start condition must be generated by software.</li> </ul> <p>Software clearing sequence for the ARLO bit 10: write '0' to this bit.</p>
CPAL_I2C_ERR_OVR	<p>Overrun/Underrun error: An overrun error can occur in Slave mode when clock stretching is disabled and the I2C interface is receiving data. The interface has received a byte (RxNE=1) and the data in DR has not been read, before the next byte is received by the interface. In this case: the last received byte is lost.</p> <ul style="list-style-type: none"> <li>- In case of Overrun error, software should clear the RxNE bit and the transmitter should re-transmit the last received byte.</li> </ul> <p>Underrun error can occur in Slave mode when clock stretching is disabled and the I2C interface is transmitting data. The interface has not updated the DR with the next byte (TxE=1), before the clock of the next byte is generated. In this case:</p> <ul style="list-style-type: none"> <li>- The same byte in the DR register is sent again.</li> <li>- The user should make sure that data received on the receiver during an underrun error are discarded and that the next bytes are written within the clock low-time specified in the I2C bus standard.</li> </ul> <p>For the first byte to be transmitted, the DR must be written after ADDR is cleared and before the first SCL rising edge. If not possible, the receiver must discard the first data.</p> <p>Software clearing sequence for the ARLO bit 11: write '0' to this bit.</p>



The wCPAL\_Options field in the CPAL device structure can be used to manage additional configuration options for peripheral initialization and communications.

The options are bit-field values (each option is coded on 1 bit into the 32-bit word-field wCPAL\_Options). Multiple options may be assigned to the wCPAL\_Options field at the same time.

**Table 11. CPAL option fields**

0	CPAL_OPT_I2C_DUALADDR
1 .. 7	7-bit address
8	Reserved
9	CPAL_OPT_DMATX_HTIT
10	Reserved
11	Reserved
12	CPAL_OPT_DMARX_HTIT
13	Reserved
14	CPAL_OPT_DMATX_CIRCULAR
15	CPAL_OPT_DMARX_CIRCULAR
16	CPAL_OPT_NO_MEM_ADDR
17	CPAL_OPT_16BIT_REG
18	Reserved
19	Reserved
20	CPAL_OPT_I2C_GENCALL
21	CPAL_DMA_1BYTE_CASE
22	CPAL_OPT_I2C_ERRIT_DISABLE
23	CPAL_OPT_I2C_NOSTOP
24	CPAL_OPT_I2C_NOSTOP_MODE
25	Reserved
26	Reserved
27	Reserved
28	Reserved
29	Reserved
30	CPAL_OPT_I2C_NACK_ADD
31	Reserved

The table below describes in detail the meaning of each bit-field.

- When a bit is set to 1, the related option is enabled.
- If it is reset, the related option is disabled (except for the bits [1..7] which hold the 7-bit address).

**Table 12. wCPAL\_Options field values**

Field value	Description
CPAL_OPT_I2C_DUALADDR	Enable the I2C Dual Addressing mode for the relative device <sup>(1)</sup>
7 Bit Address	This is the value of the Own Address 2. This value will be configured and used only if the CPAL_OPT_I2C_OA2 option is enabled.
CPAL_OPT_DMATX_HTIT	Enable the DMA Tx Half Transfer Complete interrupt for the relative device.
CPAL_OPT_DMARX_HTIT	Enable the DMA Rx Half Transfer Complete interrupt for the relative device.
CPAL_OPT_DMATX_CIRCULAR	Enable the Circular mode for the Tx DMA transfers on the relative device.
CPAL_OPT_DMARX_CIRCULAR	Enable the Circular mode for the Rx DMA transfers on the relative device.
CPAL_OPT_NO_MEM_ADDR	Enable No Memory addressing mode for the I2C relative device. This means that master sends only the device slave address (no physical addresses into slave). <sup>(2)</sup>
CPAL_OPT_16BIT_REG	Enable 16-Bit register addressing mode. Thus the register/physical address (sent after the device slave address) will be considered as 2-Bytes address. <sup>(3)</sup>
CPAL_OPT_I2C_GENCALL	Enable the I2C General Call mode for the relative device.
CPAL_DMA_1BYTE_CASE	This option is Read-only (thus it is enabled/disabled only by internal CPAL driver). It can be read by the user application to monitor the status of the communication. This option is set when DMA mode is selected and the size of the buffer to be received is equal to 1. In this case, the DMA is not enabled and the transfer is performed through interrupt mode. After the end of transfer, DMA mode is enabled again.
CPAL_OPT_I2C_ERRIT_DISABLE	Disable the I2C Errors interrupt (Bus Error, Arbitration Loss, Acknowledge Failure and Overrun/Underrun Errors). By default the errors interrupt are enabled.
CPAL_OPT_I2C_NOSTOP	Disable the sending of Stop condition at the end of the current buffer transfer for the relative I2C device. This option may be enabled when multiple packets have to be sent consecutively without Stop condition. This option can be used only if software end mode is selected. <sup>(4)</sup>
CPAL_OPT_I2C_NOSTOP_MODE	Start communication in No STOP generation mode and close communication by Generating stop.
CPAL_OPT_I2C_NACK_ADD	Enable the Initialization of the I2C Slave device without enabling the acknowledgement of its own address. This option must not be used with No Stop generation mode.
Reserved	All reserved bits values are ignored. Their use is reserved for future needs.

1. To enable this option, proceed as follows to affect The Own Address 2 and enable other options:  
wCPAL\_Options = Own\_Address\_2\_Value | CPAL\_OPT\_I2C\_DUALADDR | Other Options .

2. This option is available only for I2C devices in Master mode. Physical address is an address into the slave device into/from which the write/read operation will be performed (ie. memory address / physical register address).

- 3. This option is available only when `CPAL_OPT_NO_MEM_ADDR` is disabled. Otherwise, when `CPAL_OPT_NO_MEM_ADDR` is enabled, this option is ignored.
- 4. This option can be used only with Master transmitter and Slave receiver modes.

## 2.2.2 CPAL communication functions (cpal\_i2c.c/h)

All Communication Layer interface functions are built following the scheme detailed in the following table.

**Table 13. Architecture of CPAL Communication Layer functions**

(Type) Returned value	Name	(Type) Arguments
uint32_t  Result of the operation (CPAL_PASS if operation is successful and a different value if operation failed).	CPAL_I2C_Xxxx  Where Xxxx is the operation name (i.e. Init, Read, Write...).	CPAL_InitTypeDef* pDevInitStruct  All functions accept one single argument: the pointer to the CPAL peripheral configuration structure.

The CPAL Communication Layer functions are described in the following table.

**Table 14. CPAL Communication Layer function list**

Function name	Description
CPAL_I2C_Init()	This function initializes the related peripheral and all needed resources (GPIOs, clocks, DMA and interrupts ...) depending on the parameters configured in the configuration structure pointed by pDevInitStruct.
CPAL_I2C_DeInit()	This function frees the resources used by the related peripheral (GPIOs, clocks, DMA, interrupts ...) and disables then deinitializes the peripheral itself. Thus every used resource is configured to its default state. If a resource has not been used by the peripheral, then it is not deinitialized. Thus, if a peripheral is configured in DMA mode then configured again in Interrupt mode, when this function is called, it only deinitializes interrupt-related resources (DMA resources will remain configured). <i>Note: When calling this function, make sure that any resource shared between multiple peripherals are correctly configured after deinitialization (i.e. if a DMA channel TC interrupt is used by I2C and one other peripheral, then make sure to re-enable this DMA channel and its interrupt after calling CPAL_I2C_DeInit function).</i>
CPAL_I2C_StructInit()	This function initializes the related peripheral structure (pointed by pDevInitStruct) by filling all fields with their default values. <b>Caution:</b> Pointer fields are filled with CPAL local variable pointers. To avoid any risks, it is recommended to declare application local/global variables and fill these fields with their pointers.
CPAL_I2C_Read()	This function reads/receives a data buffer through the related peripheral. All information on the read transfer parameters and current status are extracted from the pCPAL_TransferRx field described in <a href="#">Table 8</a> . In each step of communication, the CPAL_State field of the structure pointed by pDevInitStruct is continuously updated to report the current state and the potential errors.

Table 14. CPAL Communication Layer function list (continued)

Function name	Description
CPAL_I2C_Write()	This function writes/sends a data buffer through the related peripheral. All information on the write transfer parameters and current status are extracted from pCPAL_TransferTx field described in <a href="#">Table 8</a> . In each step of communication, the CPAL_State field of the structure pointed by pDevInitStruct is continuously updated to report the current state and the potential errors. <sup>(1)</sup>
CPAL_I2C_Listen()	This function allows a slave to start a communication without knowing in advance the type of operation (Read or Write). The slave enters in an idle state and waits until it is addressed. Depending on the requested operation, User Callbacks specific to this mode are called. All information required for transfer (read and write parameters), DMA and Interrupts configuration must be implemented in these Callbacks by the user.
CPAL_I2C_IsDeviceReady()	<p>This function can be used to:</p> <ul style="list-style-type: none"> <li>– Wait until the target peripheral is ready for communication (i.e. for memories after write operations).</li> <li>– Verify that the external slave peripheral is connected to the bus (using its address).</li> </ul> <p>This function sends the peripheral slave address on the bus then waits till the peripheral responds to this address (meaning that the previous operation was successfully completed or/and the peripheral is connected to the bus). If no response is received after a timeout period the function exits and returns the CPAL_FAIL result. If the peripheral responds correctly, then the function exits and returns the CPAL_PASS result.</p> <p>This function can be called once to verify that the Slave peripheral is connected, or is in a loop to wait till the peripheral responds correctly.</p>

1. When CPAL\_I2C\_Read() or CPAL\_I2C\_Write() function is called, the user application may perform other tasks while the transfer is ongoing. To know when the transfer is complete, the application may monitor the state of the CPAL\_State field or use the DMA Tx/Rx Transfer Complete interrupts when DMA mode is enabled.

*Note:* It is possible to configure and use more than one peripheral simultaneously since each peripheral has its own state control.

## 2.3 User application interface

The user application interface consists of two files (cpal\_conf.h and cpal\_usercallback.c) which are described in the following sections.

These files may be modified by the user for each application need. The CPAL library only provides templates for these files, then the user should copy these templates into his project and optionally modify them according to the application needs.

### Important notes:

- All I2C interrupt Handlers and all the DMA-related interrupt Handlers are exclusively declared and managed by the CPAL library. The user application does not need and should not declare these interrupt handlers. If these handlers are needed for other purposes (i.e. DMA interrupt shared between several peripherals...), the user application may use the related callbacks.
- Interrupt priority groups and preemption orders are also managed by the CPAL driver. To configure these parameters, modify the cpal\_conf.h file.

### 2.3.1 Configuration interface (cpal\_conf.h)

The configuration interface allows you to customize the library for your application needs. This is not mandatory to modify this file: the default configuration may be used without any modification. Only some parameters can be modified.

To configure this single file (cpal\_conf.h), you should enable, disable or modify some options or group of options by un-commenting, commenting or modifying values of the related defines in the cpal\_conf.h file.

The CPAL configuration steps are grouped in sections and detailed in the following table:

- [Section 1: Peripheral selection](#)
- [Section 2: Transfer option configuration](#)
- [Section 3: User callback configuration](#)
- [Section 4: Timeout configuration](#)
- [Section 5: Interrupt priority selection](#)
- [Section 6: CPAL debug configuration](#)

**Table 15. CPAL configuration sections**

Section	Options	Description
Section 1: Peripheral selection	CPAL_USE_I2CX	Where X is the peripheral instance (1, 2 or 3 ...) Uncomment a define to enable the related peripheral. When commented, the peripheral cannot be used and all related resources are not declared by the CPAL library. Thus, less memory space is used).
Section 2: Transfer option configuration	The following options are static configurations allowing you to reduce the code size when some features are not used by the application.	
	CPAL_I2C_MASTER_MODE	Uncomment this define to enable Master mode use for I2C peripherals. When this define is commented, none of Master mode features and operations can be called.
	CPAL_I2C_SLAVE_MODE	Uncomment this define to enable Slave mode use for I2C peripherals. When this define is commented, none of slave mode features and operations can be called.
	CPAL_I2C_LISTEN_MODE	Uncomment this define to enable Listen mode for use by I2C slave peripherals by calling the CPAL_I2C_Listen() function. When this define is uncommented, CPAL_I2C_Read() and CPAL_I2C_Write() functions can be used only with master mode.

Table 15. CPAL configuration sections (continued)

Section	Options	Description
Section 2: Transfer option configuration (continued)	CPAL_I2C_DMA_PROGMODEL	Uncomment this define to enable the use of DMA for data transfers. When this define is commented, DMA programming model cannot be used.
	CPAL_I2C_DMA_1BYTE_CASE	Uncomment this define to enable managing 1 byte transfer sizes when DMA mode is enabled. For some peripherals, DMA cannot be used for one-byte transfer on I2C. Thus CPAL library handles this case separately. If the user application does not need to handle one-byte transfers (all application data transfers are larger than 1-byte size), the code size may be optimized by commenting this define. This define is ignored if the CPAL_I2C_DMA_PROGMODEL define is commented.
	CPAL_I2C_IT_PROGMODEL	Uncomment this define to enable the use of Interrupt mode for data turnovers. When this define is commented, all interrupt management code is disabled except for events and error management.
	CPAL_I2C_10BIT_ADDR_MODE	This option is used to allow the code to handle 10-bit addressing mode. When this option is enabled, it does not mean that all I2C peripherals communicate in 10-bit addressing mode: to select this mode for an I2C peripheral, the user has to enable the related option in the CPAL structure field.
	CPAL_16BIT_REG_OPTION	This option is valid only when CPAL_OPT_NO_MEM_ADDR option is disabled. It enables the code managing the 16-bit addressing mode for the register/physical address into slave memory. When this option is enabled, it does not mean that all devices will communicate in 16-bit register/physical addressing mode: to select this mode for a peripheral, the user has to enable the related option in the CPAL structure field.
	CPAL_I2C_CLOSECOM_METHOD1	These options enable the use of the methods for closing the I2C communication described in the reference manual. For more information, refer to the microcontroller reference manual for the description of each method.
	CPAL_I2C_CLOSECOM_METHOD2	
Section 3: User callback configuration	USE_CPAL_CRITICAL_CALLBACK <sup>2</sup>	Uncomment this define to enable the use of the critical section protection mechanism: when a group of instruction needs to be executed without being interrupted, the CPAL drivers call the critical section callbacks to protect this code section. Critical section callbacks are declared in the cpal_conf.h file and can be modified by the user: CPAL_EnterCriticalSection_UserCallback CPAL_ExitCriticalSection_UserCallback
		Generic description: this section contains all User Callbacks defined in the CPAL library. User Callbacks are functions that are called from CPAL library internal layers and may be implemented by the user in order to perform specific actions after specific events. Only their prototypes are declared in the CPAL library. To enable and use a callback, comment the related define in cpal_conf.h file, then implement the callback body into cpal_usercallback.c file (callback prototype is already declared in CPAL library). For more details about callbacks, refer to <a href="#">Section 2.3.2</a> <b>Caution:</b> Most of these functions (except error callbacks) are intended to perform short actions. Implementing functions with a too long execution time may cause communication errors.

**Table 15. CPAL configuration sections (continued)**

Section	Options	Description
Section 4: Timeout configuration	CPAL_TIMEOUT_INIT()	<p>This macro is used by the CPAL drivers to configure and enable a timeout countdown mechanism (i.e. using systick). It is called at each initialization of a CPAL peripheral (when calling CPAL_I2C_Init() function).</p> <p>The timeout counter functions as follows:</p> <ul style="list-style-type: none"> <li>– The counter generates fixed-period ticks and calls CPAL_I2C_TIMEOUT_Manager() callback at each tick.</li> <li>– The CPAL_I2C_TIMEOUT_Manager() checks the value of wCPAL_Timeout of all the available I2C peripheral structures:</li> <li>– If wCPAL_Timeout = CPAL_I2C_TIMEOUT_DEFAULT then no action is performed.</li> <li>– If wCPAL_Timeout = CPAL_I2C_TIMEOUT_MIN then the CPAL structure state is set to CPAL_STATE_ERROR and CPAL_TIMEOUT_UserCallback() is called to manage the error.</li> <li>– If wCPAL_Timeout has any other value, the function decrements its value by 1 and exit.</li> </ul> <p>The User may implement his own timeout mechanism (i.e. using SysTick timer or other timers).</p> <p>The counting unit should preferably be set to 1 millisecond (ms).</p> <p>This function should configure the counting unit and enable the counting start.</p> <p>Other timeout initialization procedures may be implemented depending on application needs.</p>
	CPAL_TIMEOUT_DEINIT()	<p>This macro is used to deinitialize the countdown mechanism. It is called whenever a peripheral is deinitialized (when calling CPAL_I2C_DeInit() function).</p> <p>Other timeout initialization procedures may be implemented depending on application needs. This function may be performed for each peripheral separately.</p>

**Table 15. CPAL configuration sections (continued)**

Section	Options	Description
Section 4: Timeout configuration (continued)	CPAL_I2C_TIMEOUT_Manager	<p>This define may be used when SysTick timer (or one other timer) is managed (in interrupt mode) for the timeout procedure. It routes the SysTick (or the timer) interrupt to the CPAL_I2C_TIMEOUT_UserCallback function handling timeout errors.</p> <p>In case of multiple peripheral types managed by the same interrupt handler, an intermediate function may be implemented and called into the interrupt handler.</p> <p>Example:                      In cpal_conf.h:  <pre>#define CPAL_I2C_TIMEOUT_Manager      UserFunction1 #define CPAL_I2C_TIMEOUT_Manager      UserFunction2</pre>                     In stm32fxxx_it.c file:  <pre>void SysTick_Handler(void) {     ...     UserFunction1();     UserFunction2();     ... }</pre>                     When another timeout mechanism (based on interrupt) is implemented, the user should route the interrupt IRQ handler to the same callback CPAL_I2C_TIMEOUT_Manager)</p>
	CPAL_I2C_TIMEOUT_MIN	<p>The minimum timeout value for the peripheral timeout counter when enabled (this value is applied to the device structure timeout field wCPAL_Timeout).</p>
	CPAL_I2C_TIMEOUT_DEFAULT	<p>The default value for the timeout counter. When the counter is set to this value, no decrement is performed on the field wCPAL_Timeout of the peripheral structure.</p>
	CPAL_I2C_TIMEOUT_WWW	<p>Where WWW can be replaced by the peripheral event (i.e. SB, ADDR ...)</p> <p>These defines determine the maximum timeout allowed for the specified event (this value is added to the CPAL_I2C_TIMEOUT_MIN to calculate the allowed timeout period).</p> <p>The user may specify different timeout periods for each event in order to meet the requirements and constraints of the application.</p>



**Table 15. CPAL configuration sections (continued)**

Section	Options	Description
Section 5: Interrupt priority selection	CPAL_NVIC_PRIORGROUP	Uncomment one of the available defines to set the level of preemption and sub-priority groups. This configuration is applied to all interrupt handlers. If the user application modifies the interrupt priority group configuration in other locations, then it impacts the CPAL functions.
	I2CX_IT_OFFSET_PREPRIO	Where X is the peripheral instance (1, 2 or 3 ...). Modify the related define value to set the level of interrupt preemption priority. All preemption priorities of the I2CX peripheral will be set in the HAL layer relatively to this offset value for example: #define I2C1_IT_EVT_PREPRIO I2C1_IT_OFFSET_PREPRIO + 0 #define I2C1_IT_ERR_PREPRIO I2C1_IT_OFFSET_PREPRIO + 2 #define I2C1_IT_DMATX_PREPRIO I2C1_IT_OFFSET_PREPRIO + 1 ...
	I2CX_IT_OFFSET_SUBPRIO	Where X is the peripheral instance (1, 2 or 3 ...). Modify the related define value to set the level of interrupt sub-priority offset. All sub-priorities of the I2CX peripheral are set in the HAL layer relatively to this offset value for example: #define I2C1_IT_EVT_SUBPRIO I2C1_IT_OFFSET_SUBPRIO + 0 #define I2C1_IT_ERR_SUBPRIO I2C1_IT_OFFSET_SUBPRIO + 0 #define I2C1_IT_DMATX_SUBPRIO I2C1_IT_OFFSET_SUBPRIO + 0 ...
Section 6: CPAL debug configuration	CPAL_DEBUG	Uncomment this define to enable an event log coded into the CPAL drivers. The event log can be re-directed through the CPAL_LOG(Str) macro. When this define is enabled, then an additional time is inserted in several places in the code, which may affect the performance of the library and even the correctness of the communication.
	CPAL_LOG(Str)	This macro is valid only when the CPAL_DEBUG define is enabled. It allows you to re-direct the logging function to the user-defined output stream (i.e. using printf()) and re-directing printf to the USART peripheral or IDE tool log window).

### 2.3.2 User callback interface (cpal\_usercallback.c)

The callback interface allows the implementation of user callbacks when needed. A template file is provided in the library (cpal\_usercallback\_template.c) with empty callback functions. This file contains all supported user callbacks.

It is not mandatory to implement Callbacks. Only callbacks that are needed by the user application may be implemented, the other ones may be kept commented (if a callback is implemented, then its related define in the cpal\_conf.h file should be commented).

All Callbacks (except when otherwise mentioned) accept a single argument: the pointer to the CPAL peripheral structure (CPAL\_InitTypeDef\*). Thus, it is possible to identify which peripheral called the function (using field CPAL\_Dev) and to determine the current state and error (using fields: CPAL\_State and wCPAL\_DevError). All Callbacks return a void value:

```
void CPAL_I2C_XXXX_UserCallabck(CPAL_InitTypeDef* pDevInitStruct);
```

**Caution:** Most of these functions (except error callbacks) are intended to perform rapid actions. Implementing functions with a too long execution time may cause communication errors.

**Table 16. CPAL configuration sections**

Callback	Description
<b>Transfer callbacks</b>	
CPAL_I2C_ZZ_UserCallback	– Where ZZ is the transfer direction: TX or RX. These functions are called before transmitting data (TX) and after receiving data (RX) on I2C peripheral.
CPAL_I2C_ZZTC_UserCallback	– Where ZZ is the transfer direction: TX or RX. These functions are called when the Transfer is completed in DMA or Interrupt programming model.
CPAL_I2C_DMAZZTC_UserCallback	– Where ZZ is the transfer direction: TX or RX. These functions are called when the Transfer Complete interrupt occurs for the related transfer direction DMA channel.
CPAL_I2C_DMAZZHT_UserCallback	– Where ZZ is the transfer direction: TX or RX. These functions are called when a Half Transfer interrupt occurs for the related transfer direction DMA channel.
CPAL_I2C_DMAZZTE_UserCallback	– Where ZZ is the transfer direction: TX or RX. These functions are called when a Transfer Error interrupt occurs for the related transfer direction DMA channel.
<b>Error Callbacks</b>	
For the error callbacks, there are two possible configurations depending on two exclusive defines (only one of these defines should be enabled, never both of them): USE_SINGLE_ERROR_CALLBACK USE_MULTIPLE_ERROR_CALLBACK Enable USE_SINGLE_ERROR_CALLBACK to use only one callback for all peripheral errors. The User must check which error caused a call of the error function by using the related error status fields in the CPAL structure. When this define is enabled, only CPAL_I2C_ERR_UserCallback can be activated. Enable USE_MULTIPLE_ERROR_CALLBACK to use a separate error callback for each peripheral error event. When this define is enabled, CPAL_I2C_ERR_UserCallback is not available.	
CPAL_I2C_ERR_UserCallback	This callback is valid only when USE_SINGLE_ERROR_CALLBACK option is enabled. CPAL accepts two arguments to this function: – pDevInstance: instance of the related peripheral (i.e. CPAL_I2C1) – DeviceError: error code (i.e. CPAL_I2C_ERR_BERR) This function is called when any error occurs on the I2C peripheral. All I2C peripherals share the same error callback. The user has to check which peripheral caused the entering in this callback.
CPAL_I2C_WWW_UserCallback	– Where WWW is the peripheral error (i.e. BERR, ARLO, OVR ...). These callbacks are valid only when USE_MULTIPLE_ERROR_CALLBACK option is enabled. These functions are called when the related error occurs on the I2C peripheral. All peripherals of the same type (i.e. all I2Cs peripherals) share the same error callback. The user has to check which peripheral caused the entry into this callback.
<b>Address mode callbacks</b>	
CPAL_I2C_GENCALL_UserCallback	This callback is valid only when CPAL_OPT_I2C_GENCALL is enabled (in Section 4) and when I2C is configured in Slave mode. This function is called when a General call event occurs on an I2C peripheral.

Table 16. CPAL configuration sections (continued)

Callback	Description
CPAL_I2C_DUALF_UserCallback	This callback is valid only when CPAL_OPT_I2C_DUALADDR is enabled (in Section 4). This function is called when a slave I2C peripheral is configured to support dual addressing mode and receives correctly its second address from the master.
<b>Listen Mode callbacks</b>	
CPAL_I2C_SLAVE_XX_XX_UserCallback	– Where XXXX is the requested operation: Read or Write. These functions are called when Slave is in Listen mode and receives its own address.
<b>Critical sections callbacks</b>	
CPAL_EnterCriticalSection_UserCallback	These callbacks are used to protect critical section of code (when a group of instructions needs to be executed without being interrupted).
CPAL_ExitCriticalSection_UserCallback	These callbacks accept 0 argument. These callbacks may be modified by the user.

## 2.4 Low layer interface (hardware abstraction layer HAL)

The low layer interface is a hardware abstraction layer allowing the CPAL library to be hardware independent and allowing the user to modify, update or configure hardware sections easily and efficiently.

It consists of the following files (where xxxx is the family identifier, for example stm32f10x):

- cpal\_hal.c
- cpal\_i2c\_hal\_stm32xxxx.c
- cpal\_i2c\_hal\_stm32xxxx.h

All hardware components (i.e. I/O pin names, clock enable defines, DMA channels...) are stored in different tables (one table for each parameter and one cell for each device). This allows hardware configuration to be easily updated regardless of hardware modifications and supported device numbers.

Some hardware configurations may be modified using the cpal\_i2c\_hal\_stm32xxxx.h configuration sections.

The supported configurations are listed in the following table:

**Table 17. HAL configuration sections**

Section	Options	Description
Section 1: Peripheral pin selection	CPAL_I2CX_YYY_GPIO_PIN	– Where X is the peripheral instance (1, 2 or 3...) and YYY is the pin function (i.e. SCL, SDA for I2C peripheral).
	CPAL_I2CX_YYY_GPIO_PORT	
	CPAL_I2C_YYY_GPIO_CLK	For each parameter, set the define value to use the related configuration, i.e: <pre>#define CPAL_I2C1_SCL_GPIO_PORT GPIOB #define CPAL_I2C1_SCL_GPIO_CLK RCC_APB2Periph_GPIOB #define CPAL_I2C1_SCL_GPIO_PIN GPIO_Pin_6</pre> For each I/O, one single configuration is allowed. A table in the cpal_i2c_stm32xxxx.h file shows the possible configurations for each I/O and each peripheral.
Section 2: DMA Channels selection	CPAL_I2CX_DMA_ZZ_Channel	– Where X is the peripheral instance (1, 2 or 3...) and ZZ is the transfer direction: TX or RX. Set the define values to configure the DMA channels for each peripheral and direction. Only one define should be used for each peripheral channel direction. A table in the cpal_i2c_stm32xxxx.h file shows the possible configurations for each DMA channel.
Preemption- and Sub-priorities are set depending on priority offsets: I2CX_IT_OFFSET_PREPRIO and I2CX_IT_OFFSET_SUBPRIO defined in the cpal_conf.h file. Generally, I2C Error interrupts should have the highest priority level, then the DMA Transfer complete interrupts and finally the I2C Event interrupts.		
Section 3: Peripheral and DMA interrupts priority selection	I2CX_IT_WWW_SUBPRIO	– Where X is the peripheral instance (1, 2 or 3...) and WWW is the peripheral interrupt or DMA channel interrupt (i.e. EVT, ERR, DMATX, DMARX...). Modify the related define value to set the level of interrupt sub-priority.
	I2CX_IT_WWW_PREPRIO	– Where X is the peripheral instance (1, 2 or 3...) and WWW is the peripheral interrupt or DMA channel interrupt (i.e. EVT, ERR, DMATX, DMARX...). Modify the related define value to set the level of interrupt sub-priority.

The HAL layer provides basic functions enabling the control and configuration of all components required for communication. These functions are detailed in the following table.

**Table 18. CPAL low layer interface function description**

Function name	Argument type	Argument name	Description
CPAL_I2C_HAL_CLKInit	CPAL_DevTypeDef	Device	This function configures and enables all I2C peripheral clocks.
CPAL_I2C_HAL_CLKDeInit	CPAL_DevTypeDef	Device	This function disables the I2C peripheral clock.
CPAL_I2C_HAL_GPIOInit	CPAL_DevTypeDef	Device	This function configures and enables all the I/O pins used by the I2C peripheral as well as the GPIO port clocks. <sup>(1)</sup>
CPAL_I2C_HAL_GPIODeInit	CPAL_DevTypeDef	Device	This function de-initializes all the I/O pins used by the I2C peripheral, configure them to their default values. The related GPIO port clocks are not disabled.
CPAL_I2C_HAL_DMAInit	CPAL_DevTypeDef	Device	This function initializes the DMA channels required for the buffer Tx/Rx transfers related to the I2C peripheral and specified by Direction and Option fields. This function also configures and enables the required DMA clocks.
	CPAL_DirectionTypeDef	Direction	
	uint32_t	Options	
CPAL_I2C_HAL_DMADeInit	CPAL_DevTypeDef	Device	This function de-initializes the DMA channels used by the I2C peripheral and configures them to their default values. The DMA clocks are not disabled by this function.
	CPAL_DirectionTypeDef	Direction	
CPAL_I2C_HAL_DMATXConfig CPAL_I2C_HAL_DMARXConfig	CPAL_DevTypeDef	Device	This function configures the DMA channels specific for Tx/Rx transfer by setting the buffer address and the number of data to be transferred through the I2C peripheral. This function checks the following options: CPAL_OPT_DMATX_CIRCULAR
	CPAL_TransferTypeDef*	TransParameter	
	uint32_t	Options	
CPAL_I2C_HAL_ITInit	CPAL_DevTypeDef	Device	This function configures and enables the NVIC interrupt channels used by the I2C peripheral according to the enabled options (Interrupt/DMA) This function checks the following options: CPAL_OPT_I2C_ERRIT_DISABLE CPAL_OPT_DMATX_HTIT CPAL_OPT_DMARX_HTIT
	uint32_t	Options	
	CPAL_DirectionTypeDef	Direction	
	CPAL_ProgModelTypeDef	CPAL_ProgModel	

**Table 18. CPAL low layer interface function description (continued)**

Function name	Argument type	Argument name	Description
CPAL_I2C_HAL_ITDeInit	CPAL_DevTypeDef	Device	This function disables the NVIC interrupt channels used by the I2C peripheral in the current configuration and according to the enabled options (Interrupt/DMA) This function checks the following options: CPAL_OPT_I2C_ERRIT_DISABLE CPAL_OPT_DMATX_HTIT CPAL_OPT_DMARX_HTIT
	uint32_t	Options	
	CPAL_DirectionTypeDef	Direction	
	CPAL_ProgModelTypeDef	CPAL_ProgModel	

1. GPIO pin selection can be customized using the section dedicated to the yhcpal\_conf.h file configuration.

## 3 CPAL functional description

### 3.1 Configuration

The whole CPAL configuration mechanism is based on a single structure (CPAL\_InitTypeDef) holding all needed configuration information for each peripheral (one structure for each peripheral) as well as the current state of the communication and of the peripheral.

A default structure is declared by the CPAL for each peripheral. And these default structures should be used by the customer application to configure and to monitor the peripheral.

Example: the following structures are declared in cpal.h file for the I2C peripherals:

```
extern CPAL_InitTypeDef I2C1_DevStructure;
extern CPAL_InitTypeDef I2C2_DevStructure;
...
```

The functions related to the configuration are:

- CPAL\_I2C\_Init()
- CPAL\_I2C\_DeInit()
- CPAL\_I2C\_StructInit()

*Note: It is possible to configure and use more than one peripheral simultaneously since each peripheral has its own state control.*

#### 3.1.1 CPAL\_I2C\_Init() functional description

CPAL\_I2C\_Init() function should be called at the startup of the application before performing any communication operations. It should be called after filling the related I2Cx peripheral structure fields (I2Cx\_DevStructure) with the required parameters.

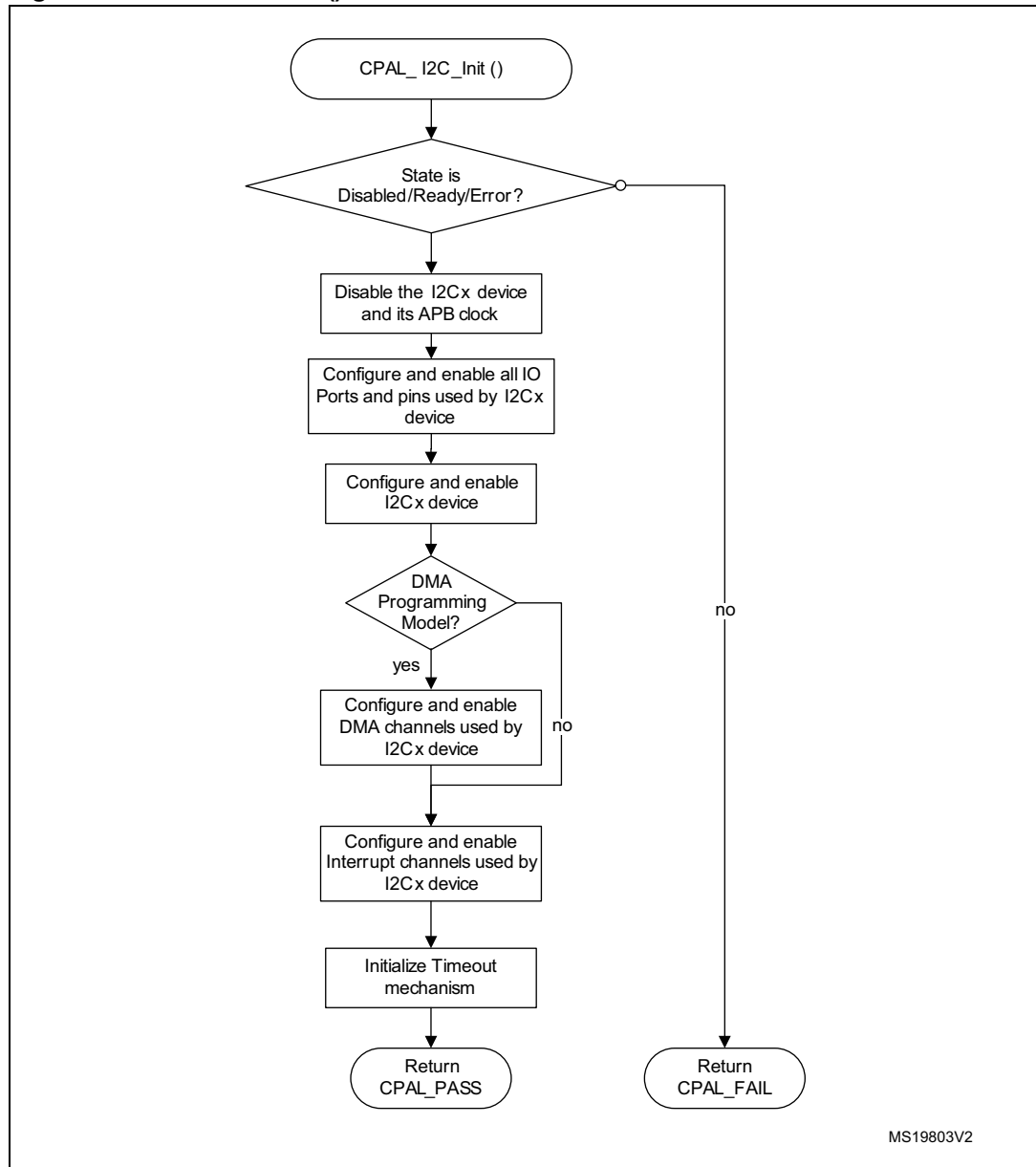
The CPAL\_I2C\_Init() function performs the following actions:

- Disable the I2Cx peripheral and reset its APB clock.
- Disable then enable and configure the GPIO ports and pins used for the I2Cx peripheral.
- Enable and initialize the I2Cx peripheral according to parameters in initialization structure pointed by the pCPAL\_I2C\_Struct field and the additional configuration set into the field wCPAL\_Options (General Call mode, Dual Address mode...).
- Enable the DMA and/or the interrupts and their related clocks and channels according to the values in the fields CPAL\_Direction, CPAL\_ProgModel and wCPAL\_Options.

Initialize the Timeout mechanism as described in [Section 3.3.1](#).

This function can be called as many times as required (i.e. when some configuration parameters are modified), but in all cases it must be called at least once before starting any communication operation.

Figure 3. CPAL\_I2C\_Init() function flowchart



### 3.1.2 CPAL\_I2C\_DeInit() functional description

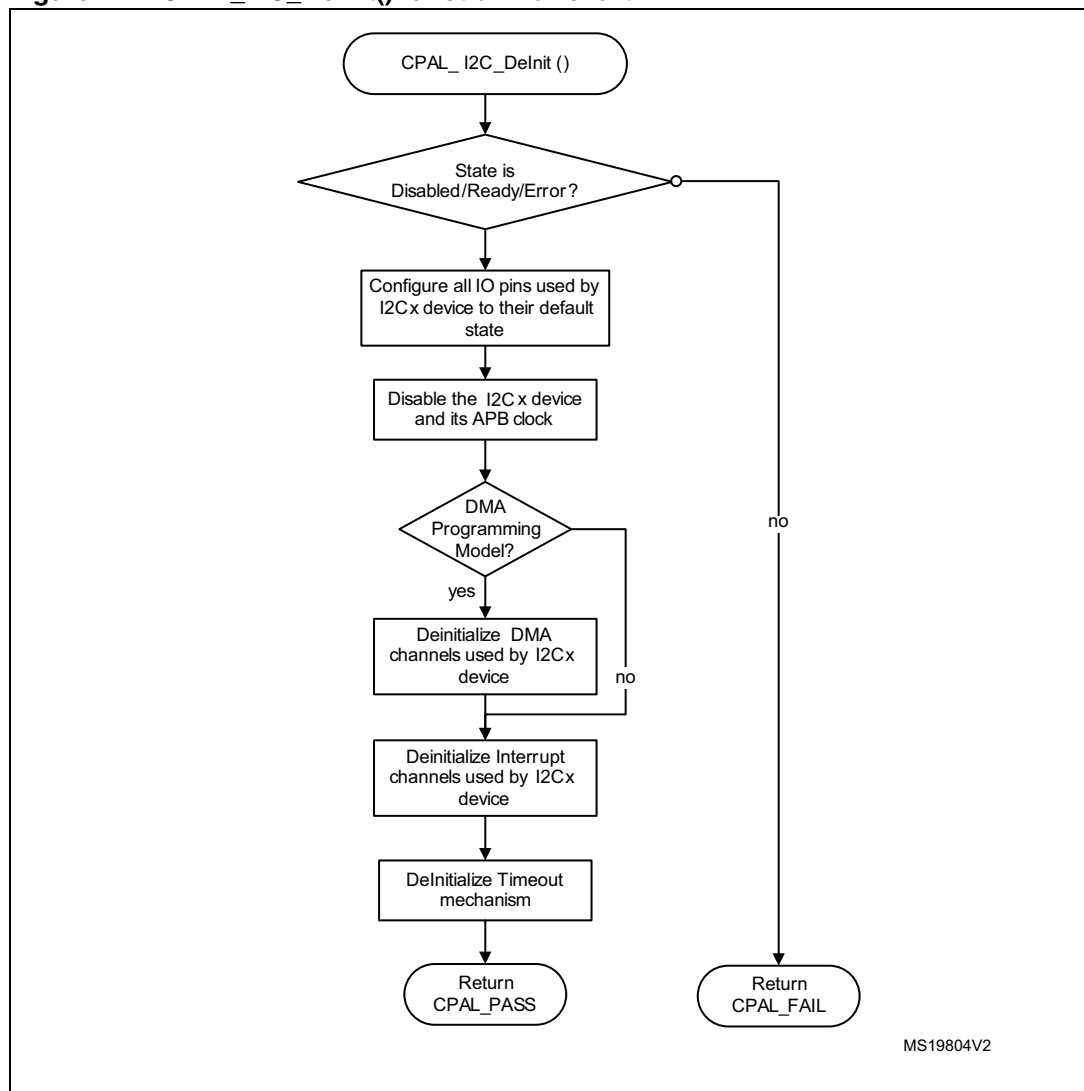
When the communication transfer is over, or when the application has to abort the peripheral operations, the CPAL\_I2C\_DeInit() function can be called to free all the resources used by the peripheral in the current configuration and return to default values.



The CPAL\_I2C\_DeInit() function performs the following actions:

- Check the state of the CPAL peripheral:
  - If the state is: CPAL\_STATE\_READY or CPAL\_STATE\_ERROR or CPAL\_STATE\_DISABLED the operation is performed and then the function exits and returns “CPAL\_PASS” value.
  - If the state is different from the states above, then the function exits and returns “CPAL\_FAIL” value.
- Disable the GPIO ports and pins used for the I2Cx peripheral (reset to default state).
- Disable the I2Cx peripheral and its APB clock.
- Disable the DMA and/or the interrupts and their related clocks and channels depending on the current values of fields CPAL\_Direction, CPAL\_ProgModel and wCPAL\_Options.

Figure 4. CPAL\_I2C\_DeInit() function flowchart



### 3.1.3 CPAL\_I2C\_StructInit() functional description

The default values could be used for the peripheral configuration by setting the I2Cx\_DevStructure structure fields to their default values using the function CPAL\_I2C\_StructInit().

This function sets the default values as detailed in the following table.

**Table 19. CPAL\_I2C\_Struct\_Init() default values**

Field		Default value
CPAL_Dev		CPAL_I2C1
CPAL_Direction		CPAL_DIRECTION_TXRX
CPAL_Mode		CPAL_MODE_MASTER
CPAL_ProgModel		CPAL_PROGMODEL_DMA
CPAL_State		CPAL_STATE_DISABLED
wCPAL_DevError		CPAL_I2C_ERR_NONE
wCPAL_Options		0x00000000 (all options disabled)
wCPAL_Timeout		CPAL_TIMEOUT_DEFAULT
pCPAL_TransferTx		pNull
pCPAL_TransferRx		pNull
pCPAL_I2C_Struct	I2C_ClockSpeed	100000 (100 KHz)
	I2C_Mode	I2C_Mode_I2C
	I2C_DutyCycle	I2C_DutyCycle_2
	I2C_OwnAddress1	0x00
	I2C_Ack	I2C_Ack_Enable
	I2C_AcknowledgedAddress	I2C_AcknowledgedAddress_7bit

## 3.2 Communication

Once the configuration step is performed successfully, the application is able to perform communication operations using the functions:

- CPAL\_I2C\_Read()
- CPAL\_I2C\_Write()
- CPAL\_I2C\_Listen()
- CPAL\_I2C\_IsDeviceReady()

The CPAL\_I2C\_Read() and CPAL\_I2C\_Write() functions require that the peripheral transfer structures should be already configured as described in [Table 8: CPA\\_TransferTypeDef structure fields](#).

Transfer structures which are used with CPAL\_I2C\_Listen() function may be configured before calling this function or after calling it in CPAL\_I2C\_SLAVE\_WRITE\_UserCallback or CPAL\_I2C\_SLAVE\_READ\_UserCallback.

It is advised that these fields point to local or global variables initialized by the application, in order to avoid risks due to non-initialized pointers and memory allocation errors.

Once the CPAL\_I2C\_Read(), CPAL\_I2C\_Write() and CPAL\_I2C\_Listen() function is called, the user application may:

- Wait till the end of transfer by monitoring:
  - the wCPAL\_State field value
  - or the number of data in the wNumData field of the pCPAL\_TransferRx or pCPAL\_TransferTx structure
  - or the DMA transfer complete callbacks
  - or the interrupt transfer callbacks.
- Perform other tasks while the transfer is ongoing (transfer is handled by interrupts or DMA channels) and check periodically the state of the transfer (as explained above).
- Move to other tasks and control the CPAL transfer only through DMA Transfer complete callbacks (described in [Section 2.3.2](#)). This method is preferred for continuous communication with DMA circular mode option enabled.

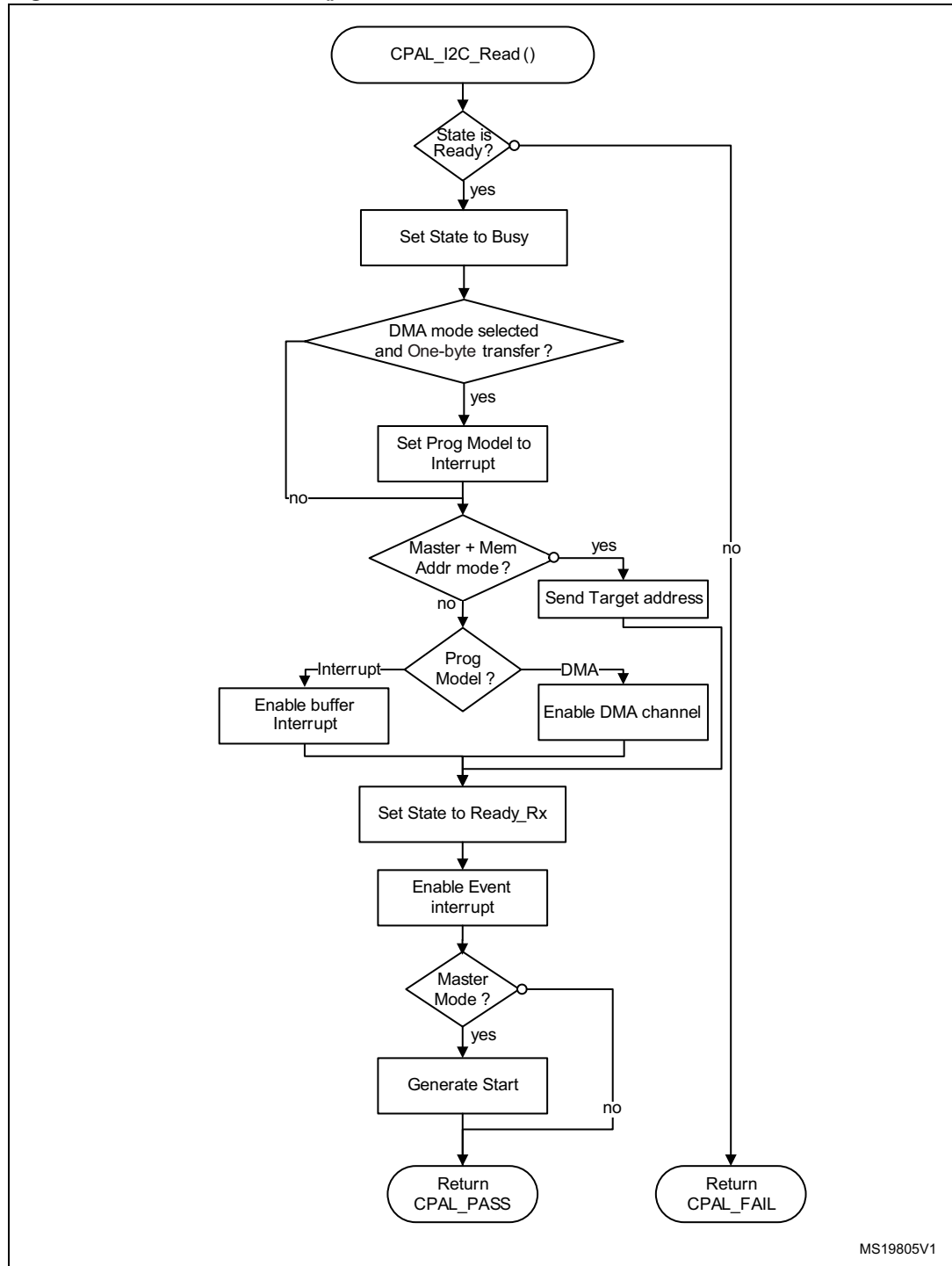
The CPAL\_I2C\_Read(), CPAL\_I2C\_Write() and CPAL\_I2C\_Listen() functions just prepare and configure the communication.

The effective transfer operation (transmission, reception, event management, error management ...) is handled by interrupts and DMA functions as described in [Section 3.2.5](#).

### 3.2.1 CPAL\_I2C\_Read() functional description

The CPAL\_I2C\_Read() function use the information configured in the peripheral structure and the information pointed by pCPAL\_TransferRx to perform the read of the received buffer through the selected I2Cx peripheral.

Figure 5. CPAL\_I2C\_Read() function flowchart

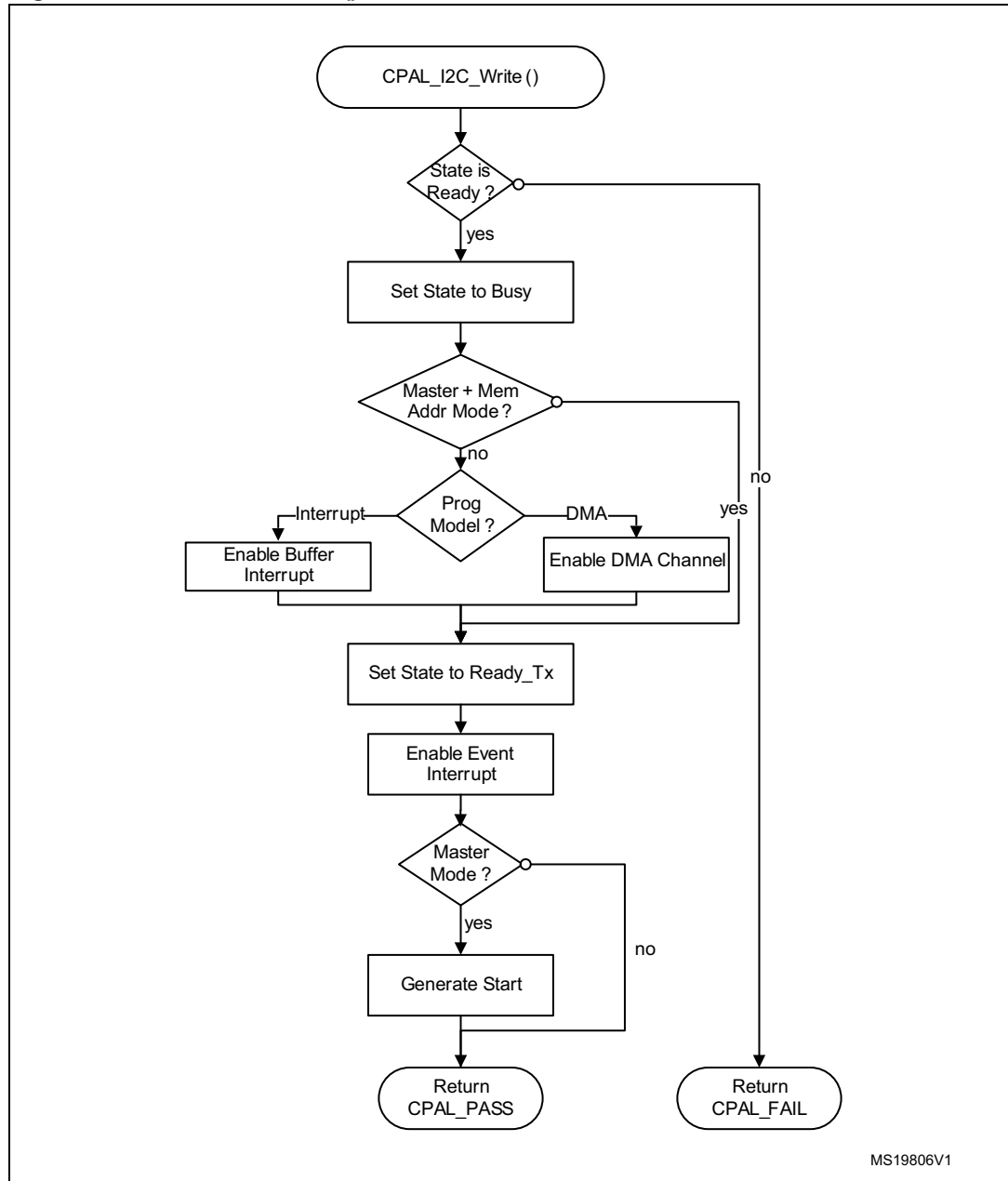


MS19805V1

### 3.2.2 CPAL\_I2C\_Write() functional description

The CPAL\_I2C\_Write() function uses the information configured in the peripheral structure and the information pointed by pCPAL\_TransferTx to perform the write of the selected buffer through the selected I2Cx peripheral.

Figure 6. CPAL\_I2C\_Write() function flowchart



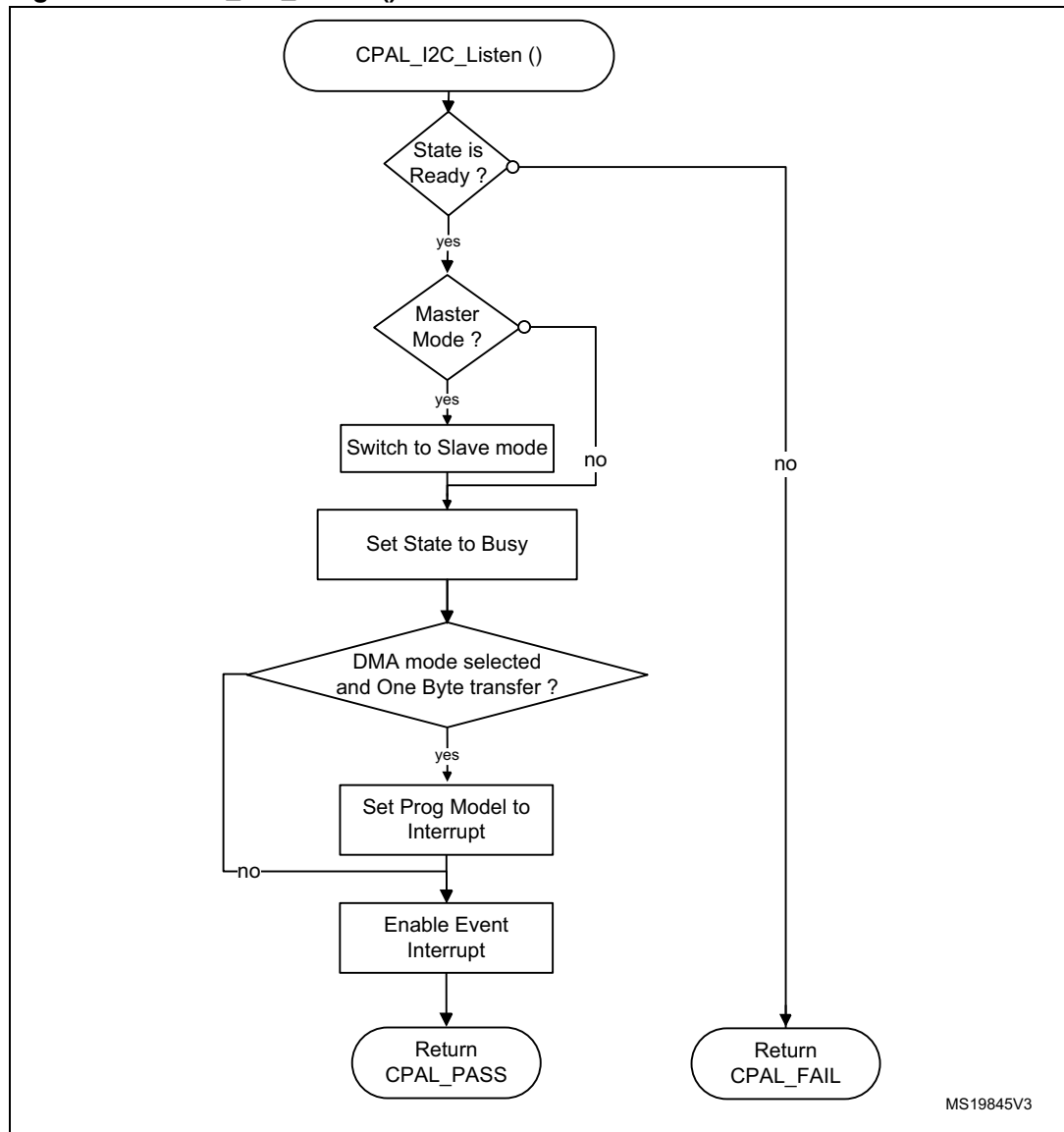
### 3.2.3 CPAL\_I2C\_Listen () functional description:

The CPAL\_I2C\_Listen () function is only used with slave device when Listen mode is activated. If this function is called in master mode the device is forced to slave mode.

CPAL\_I2C\_Listen() function replaces CPAL\_I2C\_Read() and CPAL\_I2C\_Write() functions. Code source overload of CPAL Library is reduced when Listen mode is used (when Master mode is disabled code source will decrease significantly).

When CPAL\_I2C\_Listen () function is called, slave enters in idle state and waits until it receive its own address. In ADDR routine of the slave mode, CPAL\_I2C\_SLAVE\_READ\_UserCallback or CPAL\_I2C\_SLAVE\_WRITE\_UserCallback is called, depending on type of received request. In these Callbacks user must configure transfer parameters and DMA or Interrupts in accordance with selected programming model.

Figure 7. CPAL\_I2C\_Listen () function flowchart



### 3.2.4 CPAL\_I2C\_IsDeviceReady() functional description

The CPAL\_I2C\_IsDeviceReady() function can be called by Master device to verify that a Slave device is connected to the bus or to check if/when the last operation has been successfully completed (i.e. for memory devices).

Thus, it could be called once or into a loop. It sends the communication headers (depending on the peripheral mode and the configuration) with the peripheral address and waits for the slave to respond.

If the peripheral responds correctly to this address, the function exits and returns "CPAL\_PASS".

If an incorrect answer is received or no answer is received during the timeout period, then the timeout mechanism is triggered and used to exit the function with "CPAL\_FAIL" value.

In case of success, the function closes the communication so that a new communication can start with the selected peripheral.

This function does not use or affect the transfer parameters of reception or transmission (structures pointed by pCPAL\_TransferTx and pCPAL\_TransferRx fields).

### 3.2.5 CPAL interrupts and DMA management

Once the CPAL\_I2C\_Read() and CPAL\_I2C\_Write() functions are called, the communication is configured and prepared (DMA or interrupt channels enabled, communication header sent in Master mode ...). Then the effective transmission/reception operations as well as the event and error management is performed by interrupts and DMA functions.

These operations are different according to the peripheral type (I2C), the peripheral mode (Master, Slave), the programming model (Interrupt, DMA) and the option configuration (No memory addressing mode, General call mode, ...).

For I2C peripherals, there are two types of interrupts:

- Event and error interrupts
- Buffer interrupts

Regardless of the selected programming model (CPAL\_PROGMODEL\_DMA or CPAL\_PROGMODEL\_INTERRUPT), the event and error interrupts are always enabled and used to control the communication flow. Buffer interrupt will be enabled only when CPAL\_PROGMODEL\_INTERRUPT mode is selected or when CPAL\_PROGMODEL\_DMA mode is selected for a buffer size equal to one.

The priority of events and error management corresponds to the order in which they are tested into the interrupt functions:

**Table 20. I2C interrupt management order**

Order	Interrupt	Details	Callback
1	SB	Available for Master mode only. Manages the event "Start Bit sent" which means that the transmission of the start bit has been performed correctly (slave address has to be sent).	NA
2	ADDR	Manages the event "Address phase done" which means: – In Master mode: that the Slave acknowledged its address. – In Slave mode: that the peripheral received the start bit followed by its own address and acknowledged it.	CPAL_I2C_GENCALL_UserCallback CPAL_I2C_DUALF_UserCallback
3	ADD10	Available in Master mode only. Manages the event "10-bit Address acknowledged" which means that the Slave received and acknowledged its 10-bit address.	NA
4	TXE	Manages the event "Transfer Buffer Empty" which means a new data shall be written in the I2C data register for the next transfer.	CPAL_I2C_TX_UserCallback CPAL_I2C_TXTC_UserCallback
5	RXNE	Manages the event "Receive Buffer Not Empty" which means a data has been received and should be read from the data register.	CPAL_I2C_RX_UserCallback CPAL_I2C_RXTC_UserCallback
6	STOPF	Available in Slave mode only. Manages the event "Stop bit received" which means that the Master has closed the communication.	Slave mode only: CPAL_I2C_RXTC_UserCallback

For I2C peripherals, the Error interrupt has a dedicated IRQ channel different from the Event interrupt. This means that errors can be managed asynchronously and independently of the communication events.

When DMA mode is selected with DMA interrupt options, the following DMA interrupts are handled: Transfer Complete interrupt, Half Transfer Complete interrupt and Transfer Error interrupt.

**Table 21. DMA interrupt management order**

Order	Interrupt	Details	Callback
1	TC	Manages the DMA event "Transfer Complete" which means that all data programmed in DMA controller have been transferred (transmitted/received).	CPAL_I2C_DMATXTC_UserCallback CPAL_I2C_DMARXTC_UserCallback
2	HT	Manages the DMA event "Half Transfer Complete" which means that half of the data programmed in DMA controller has been transferred (transmitted/received).	CPAL_I2C_DMATXHT_UserCallback CPAL_I2C_DMARXHT_UserCallback
3	TE	Manages the event "DMA Transfer Error" which means that an error occurred during the DMA transfer.	CPAL_I2C_DMATXTE_UserCallback CPAL_I2C_DMARXTE_UserCallback

All errors lead to a single operation: call CPAL\_I2C\_ErrorHandler().



The CPAL\_I2C\_ErrorHandler() function handles all peripheral errors and timeout errors (DMA errors are managed by CPAL\_I2C\_DMATXTE\_UserCallback() and CPAL\_I2C\_DMARXTE\_UserCallback() functions). This function performs the basic error recovery operations (clears the error flag and source if possible, resets the CPAL peripheral state ...) and then calls the user error callback.

### 3.3 Event and error management (user callbacks)

As mentioned in previous sections, the CPAL allows the user application to control the communication and to perform specific actions triggered by specific communication/errors events through the callback functions.

Into all CPAL communication layer drivers, in strategic places some functions are called. The prototypes of these functions are declared into the CPAL drivers but they are not implemented. The user application may implement and use them when needed (refer to [Section 3: User callback configuration](#) in [Table 15: CPAL configuration sections](#) for more details about callback configurations). To know at which level a callback function is called, refer to [Section 3.2.5](#).

All User Callbacks are optional: if a callback is not implemented (its define should be uncommented in the cpal\_conf.h file) then it will be defined as a void function and will not impact the code or the functionality of the driver.

All Callbacks accept a single argument: the pointer to the CPAL Peripheral structure (CPAL\_InitTypeDef\*). Thus, it is possible to identify which peripheral called the function (using field CPAL\_Dev) and to determine the current state and error (using fields: CPAL\_State and wCPAL\_DevError). All Callbacks return a void value:

```
void CPAL_I2C_XXXX_UserCallback(CPAL_InitTypeDef* pDevInitStruct);
```

**Caution:** Callbacks (except error callbacks) are used to perform short operations. If a callback function takes a too long execution time, it may lead to communication errors due to the inserted delay. This is not applicable for Error callbacks since in this case communication is already stopped.

The list of all available callbacks and their description is provided in the following table.

**Table 22. CPAL I2C user callback list**

Callback name	Description
<b>Communication User Callbacks</b>	
These functions are called when correct communication events occur. They are generally used to prepare data before transmitting or processing them after reception. Thus, they should be as short as possible in order to avoid affecting the communication process.	
CPAL_I2C_TX_UserCallback	This function is called when the TXE interrupt handler is entered and before writing data to the peripheral DR register. This callback shall be used to prepare the next data to be sent.
CPAL_I2C_RX_UserCallback	This function is called when the RXNE interrupt handler is entered and after reading the received data from the peripheral DR register. This callback shall be used to manage the last received data.
CPAL_I2C_TXTC_UserCallback	This function is called when data transmission is completed and communication is closed in the Interrupt and DMA Programming model.

**Table 22. CPAL I2C user callback list (continued)**

Callback name	Description
CPAL_I2C_RXTC_UserCallback	This function is called when data reception is completed and communication is closed in Interrupt and DMA Programming Model.
CPAL_I2C_DMATXTC_UserCallback	DMA TX callbacks are available if the DMA programming model is selected for at least one peripheral. These functions are called when a DMA interrupt is entered for the configured DMA channel and the related event has occurred for the transmitting direction DMA channel: TC (Transfer Complete), HT (Half Transfer Complete) or TE (DMA Transfer Error).
CPAL_I2C_DMATXHT_UserCallback	
CPAL_I2C_DMATXTE_UserCallback	
CPAL_I2C_DMARXTC_UserCallback	DMA RX callbacks are available if the DMA programming model is selected for at least one peripheral. These functions are called when a DMA interrupt is entered for the configured DMA channel and the related event has occurred for the receiving direction DMA channel: TC (Transfer Complete), HT (Half Transfer Complete) or TE (DMA Transfer Error).
CPAL_I2C_DMARXHT_UserCallback	
CPAL_I2C_DMARXTE_UserCallback	
CPAL_I2C_GENCALL_UserCallback	General Call event callback is available only when the option CPAL_OPT_I2C_GENCALL is enabled for the peripheral. This function is called when a General Call address is correctly received for a slave I2C peripheral.
CPAL_I2C_DUALF_UserCallback	Dual Address Flag callback is available only when the option CPAL_OPT_I2C_DUALADDR is enabled for the peripheral. This function is called when the peripheral (in Slave mode) receives correctly a header with its second address.
CPAL_I2C_SLAVE_WRITE_UserCallback	This function is called in slave listen mode when a master device requests a Write operation. This callback must be implemented to configure pCPAL_TransferTX with transfer parameters. DMA and interrupts must be also configured in this callback depending on the selected programming model. User should call "CPAL_I2C_Enable_DMA_IT(pDevInitStruct, CPAL_DIRECTION_TX)" followed by "__CPAL_I2C_HAL_ENABLE_EVTIT(pDevInitStruct->CPAL_Dev)" after configuring Transfer parameters.
CPAL_I2C_SLAVE_READ_UserCallback	This function is called in slave listen mode when a master device requests a Read operation. This callback must be implemented to configure pCPAL_TransferRX with transfer parameters. DMA and interrupts must also be configured in this callback depending on the selected programming model. User should call "CPAL_I2C_Enable_DMA_IT(pDevInitStruct, CPAL_DIRECTION_RX)" followed by "__CPAL_I2C_HAL_ENABLE_EVTIT(pDevInitStruct->CPAL_Dev)" after configuring Transfer parameters.
<b>Error User Callbacks</b>	
These functions are called when an error occurs during communication. The user application should implement these functions to recover from the error and restore communication. Basic recovery operations are already performed by the CPAL drivers before calling the error Callbacks (clear error flag and source when possible, reset the CPAL state fields and timeout mechanism). The user application should then try to restore lost buffers/data or reset the whole system when recovery is not possible.	

Table 22. CPAL I2C user callback list (continued)

Callback name	Description
CPAL_I2C_BERR_UserCallback	Multiple Error callbacks are available only when the define USE_MULTIPLE_ERROR_CALLBACK is enabled in the file cpal_conf.h. Each function is called when the peripheral error interrupt is entered and the error is identified to be one of the following: BERR (Bus Error), ARLO (Arbitration Loss), OVR (Overrun or Underrun) and AF (Acknowledge Failure).
CPAL_I2C_ARLO_UserCallback	
CPAL_I2C_OVR_UserCallback	
CPAL_I2C_AF_UserCallback	
CPAL_I2C_ERR_UserCallback	Single Error callback is available only when the define USE_SINGLE_ERROR_CALLBACK is enabled in the cpal_conf.h file. This function is called when the peripheral error interrupt is entered and before identifying the error source (BERR, ARLO, OVR or AF). The user application may check the error using the parameter passed to the callback (pointer to the peripheral structure).
<b>Timeout User Callbacks</b>	
Timeout functions are called by the CPAL drivers when detection of failures within a defined time-frame is required. When an operation takes more time than expected, the timeout user callback function is called. Basic recovery operations are already performed by the CPAL drivers before calling this function (clear error sources when possible, stop communication, reset the CPAL state fields ...). The user application should then try to restore lost buffers/data or reset the whole system when recovery is not possible.	
CPAL_TIMEOUT_INIT	This callback is used to configure and enable the timeout counter peripheral/function used to generate periodic ticks/interrupts (i.e. enable and configure SysTick timer and its related interrupt). This function is called into CPAL_I2C_Init() function.
CPAL_TIMEOUT_DEINIT	This callback is used to free the counter resource when the peripheral is de-initialized (i.e. disable SysTick timer and its interrupt). This function is called into CPAL_I2C_DeInit() function.
CPAL_TIMEOUT_UserCallback	This function is called when a timeout condition occurs for a peripheral. It is not called when timeout occurs simultaneously with a peripheral error (BERR, OVR...) because in this case only the error callback is called.

### 3.3.1 Timeout management

For a communication to start, in most cases, the application must wait until some events occur. These events may depend on external devices and may not occur in case of a device- or bus-failure. In this case the only way to detect the error is to limit the time during which the system can wait for the event to occur. CPAL drivers implement a Timeout mechanism used to achieve this control and prevent the application from being blocked because of any communication failure.

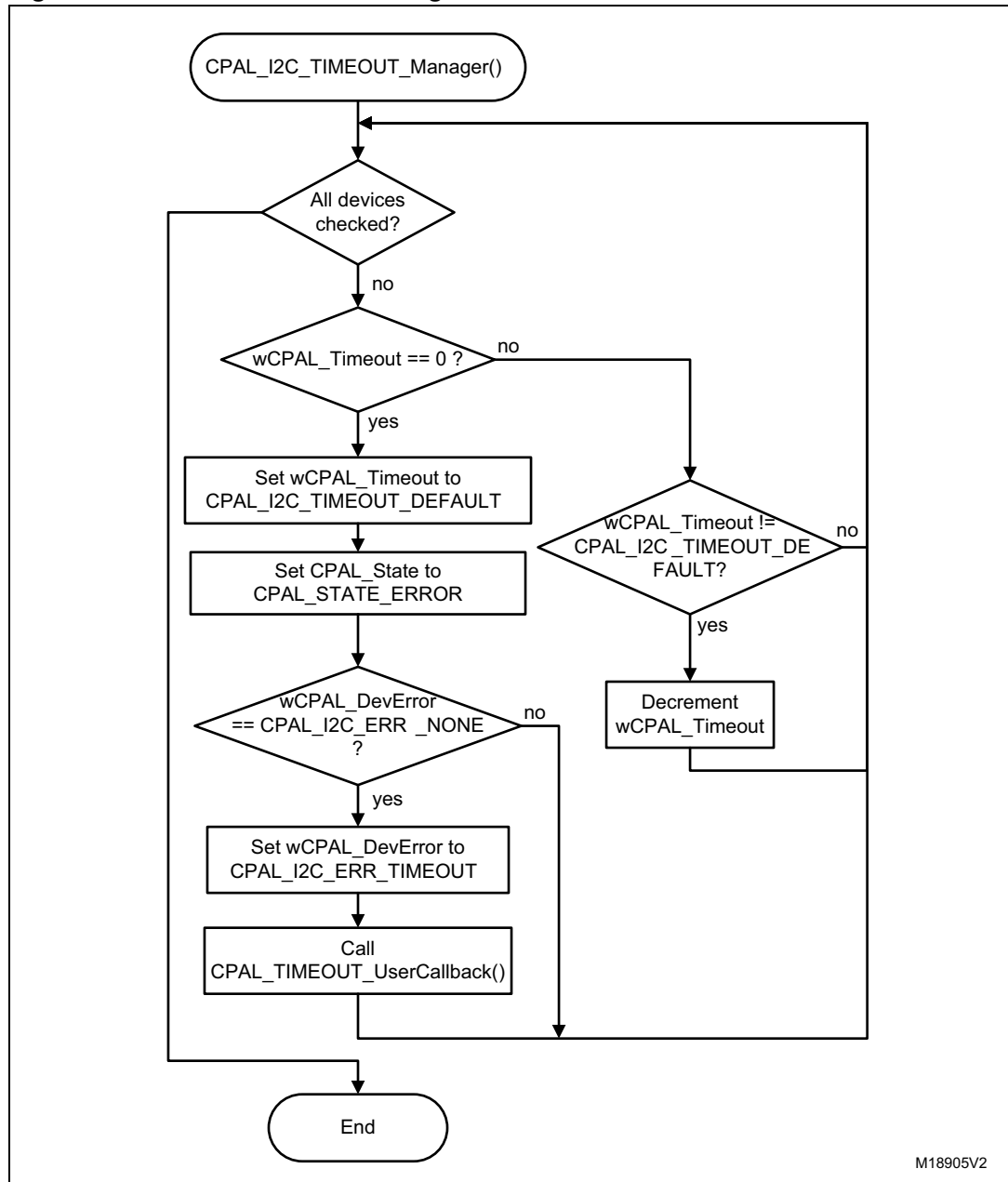
The timeout mechanism is based on three elements:

- **Timeout counter:** A peripheral/function should be used to count and generate periodic and equal ticks (i.e. SysTick or Timer). This peripheral/function may generate an interrupt (or be scheduled for a specified period of time) and call the CPAL peripheral timeout manager function: CPAL\_I2C\_TIMEOUT\_Manager() at each tick. The initialization of the counter mechanism is performed by the callback CPAL\_TIMEOUT\_INIT() which should be implemented by the user application. It triggers all initialization procedures required for the counter peripheral/function (i.e. configure and enable the SysTick timer and its interrupt). This function is called in

CPAL\_I2C\_Init() function. One other callback is available to free the counter resource: CPAL\_TIMEOUT\_DEINIT) and is called in the CPAL\_I2C\_Init() function.

- The Timeout Manager: This function is implemented in the CPAL driver. The Timeout Manager checks all the I2C peripheral structures and verify the value of wCPAL\_Timeout. If it is different from CPAL\_I2C\_TIMEOUT\_DEFAULT and other than 0, it decrements the wCPAL\_Timeout value by 1. If it reaches 0 then the specified delay has been elapsed and the CPAL\_TIMEOUT\_UserCallback() function is called. This function is already implemented in the CPAL library and could be called directly by the user application or mapped to an interrupt or a user function through the define in the cpal\_conf.h file (i.e. #define CPAL\_I2C\_TIMEOUT\_Manager SysTick\_Handler).
- The timeout user callback (CPAL\_TIMEOUT\_UserCallback()): this function is called when the timeout condition occurs for any peripheral. The user application may clear reset communication or peripheral or microcontroller depending on the situation.

Figure 8. CPAL I2C timeout manager flowchart



M18905V2

## 4 How to use and customize the CPAL library (step by step)

The CPAL library offers several configuration and customization levels. Some configurations are static (defines in `cpal_conf.h` file) and others are dynamic (peripheral configuration structure field assignment).

Most configuration and customization steps are optional (default configuration or a subset of default configuration may be used instead of setting all parameter values).

The following sections explain all the steps needed to configure, customize and implement the CPAL library into a user application.

### 4.1 Basic configuration

The first step (optional) is to set the configuration of the peripheral and the CPAL driver.

This step can be done through the modification of the `cpal_conf.h` (refer to [Section 2.2.1](#) for detailed description) file which contains all the configurable parameters of the peripheral and CPAL library.

#### Important notes:

- All I2C interrupt Handlers and all the related DMA interrupt Handlers are exclusively declared and managed by the CPAL library. The user application does not need and should not declare these interrupt handlers. If these handlers are needed for other purposes (i.e. DMA interrupt shared between several peripheral, etc.) the user application may use the related callbacks.
- Interrupt priority groups and preemption orders are also managed by the CPAL driver. To configure these parameters, modify the `cpal_conf.h` file.

`cpal_conf.h` file is divided into independent sections:

- Section 1: Select peripherals to be used
- Section 2: Configure transfer options
- Section 3: Select and configure user and error callbacks
- Section 4: Configure timeout management.
- Section 5: Global interrupt priority offsets
- Section 6: Configure the log macro

#### 4.1.1 Select peripherals to be used

The first section of `cpal_conf.h` allows a selection of the peripherals which will be used and enabled by the CPAL library. When a peripheral is not used, its related define should be commented in order to save memory space and execution time.

Example:

I2C1 used and I2C2 not used:

```
#define CPAL_USE_I2C1          /*<! Uncomment to use I2C1 peripheral */
//#define CPAL_USE_I2C2      /*<! Uncomment to use I2C2 peripheral */
```

### 4.1.2 Configure transfer options

This section allows a choice of some transfer options. Transfer options are different from the options set to the `wCPAL_Options` field of the CPAL peripheral structures. The transfer options are static defines which are used to remove the section of code handling the related communication option (and thus to reduce the code size). When a transfer option is disabled, all the related code in the CPAL driver is disabled, so the option is no more available for the `wCPAL_Options` field.

For example, if the `CPAL_16BIT_REG_OPTION` option is disabled (related define is commented in `cpal_conf.h` file):

```
//#define CPAL_16BIT_REG_OPTION
```

Then, setting the option `CPAL_OPT_16BIT_REG` to the `wCPAL_Options` has no effect.

### 4.1.3 Select and configure user and error callbacks

This section allows a selection of the callbacks which will be implemented by the user application. To implement a callback in your application: comment the related callback define in this section and then implement the body of the callback in your application (the prototype is already declared in the CPAL driver).

For more details about user callbacks refer to [Section 3.3](#).

### 4.1.4 Configure timeout management

This section is used to configure the timeout mechanism. Timeout mechanism may be not used: in this case, CPAL will not handle communication errors and will handle only peripheral errors which generate error interrupts. In this case, you have to define the timeout callbacks as void functions (i.e. `#define CPAL_TIMEOUT_UserCallback (void)`).

To use the Timeout mechanism (which offers a higher level of communication security), a counter peripheral/function should be used in order to call the `CPAL_I2C_TIMEOUT_Manager()` function at each tick (preferably each 1 ms, and through a high priority interrupt).

**Caution:** If the counter is implemented using an interrupt, then the associated interrupt channel must be set to a priority level strictly higher than all the CPAL interrupt channels priorities.

To configure the timeout mechanism correctly, three steps can be followed:

- Set the Initialization and De-Initialization functions: map the `CPAL_TIMEOUT_INIT()` and `CPAL_TIMEOUT_DEINIT()` functions to a counter initialization function (i.e. `SysTick_Config((SystemCoreClock / 1000))` and `SysTick->CTRL = 0`). These functions will be called respectively in `CPAL_I2C_Init()` and `CPAL_I2C_DeInit()` functions.
- Map the `CPAL_I2C_TIMEOUT_Manager` function to a user function (ideally, this function could be directly mapped to the counter interrupt in order to be called each time the specified period of time has elapsed: that is, `#define CPAL_I2C_TIMEOUT_Manager SysTick_Handler`).
- Then it is possible (optionally) to modify the maximum timeout value associated to each operation in order to meet the application constraints. To modify the maximum timeout value for an operation, set the required value in ms to the define related to this operation (for example: if the application should wait a maximum of 10 ms for the Start Bit flag: `#define CPAL_I2C_TIMEOUT_SB 10`).

**Example:**

Timeout mechanism implemented through SysTick interrupt configured to be generated each 1 ms:

```
#define CPAL_TIMEOUT_INIT()          SysTick_Config((SystemCoreClock / 1000))
#define CPAL_TIMEOUT_DEINIT()        SysTick->CTRL = 0

#define CPAL_I2C_TIMEOUT_Manager     SysTick_Handler

#define CPAL_I2C_TIMEOUT_SB          2
#define CPAL_I2C_TIMEOUT_ADDR        3
#define CPAL_I2C_TIMEOUT_ADD10       3
#define CPAL_I2C_TIMEOUT_TXE         2
#define CPAL_I2C_TIMEOUT_RXNE        2
#define CPAL_I2C_TIMEOUT_BTF         4
#define CPAL_I2C_TIMEOUT_BUSY        2
```

**4.1.5 Set Events, Errors and DMA interrupt priorities**

This section can be used to configure the global priority level offset for each I2CX peripheral. This offset sets the peripheral interrupt priority levels in the file `cpal_i2c_stm32xxxx.h` file.

If the Timeout mechanism is implemented with an interrupt channel, then make sure that its interrupt priority is higher than any other CPAL interrupt priority.

**4.1.6 Configure the Log Macro**

The CPAL library offers an internal debugging mechanism based on messages printed for most operations performed by CPAL driver. The printed messages may be mapped to an IDE log window, to an LCD screen, to a USART interface (RS232)...

To enable this feature, the define `CPAL_DEBUG` should be enabled and the log macros `CPAL_LOG()` should be mapped to a user printing function.

Be aware that using this feature may slow down the execution of the CPAL operations and may even affect in some cases the communication. The `CPAL_LOG` function should be optimized to perform fast print operation in order to minimize the impact of this feature on the communication.

**Example:**

Implementation of `CPAL_LOG` with `printf` function modified to send data through a USART interface to a "Hyperterminal" application:

```
#define CPAL_DEBUG

#ifdef CPAL_DEBUG
#define CPAL_LOG(Str)          printf(Str)
#else
#define CPAL_LOG(Str)          ((void)0)
#endif /* CPAL_DEBUG */
```

And in the user application, define the `printf` function:

```
#ifdef __GNUC__
/* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf
   set to 'Yes') calls __io_putchar() */
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */
```



## 4.2 Structure initialization

All CPAL functions use peripheral configuration structures (I2Cx\_DevStructure) to control and monitor all communication and configuration operations. Consequently, before using any function of the CPAL, the related peripheral structure has to be set.

For each peripheral a predefined structure is declared into the CPAL driver. This structure has to be used into the application for any configuration or monitoring purposes (no declaration is needed for these structures as they are already exported by CPAL drivers).

### Example:

```
CPAL_InitTypeDef I2C1_DevStructure;
CPAL_InitTypeDef I2C2_DevStructure;
...
```

There are three ways to set configuration structures:

- Use default configuration: to use the predefined default configuration, call the function `CPAL_I2C_StructInit()` which will set the default values detailed in [Table 19: CPAL\\_I2C\\_Struct\\_Init\(\) default values](#).
- Modify only some fields after calling `CPAL_I2C_StructInit()`. Example:
  - `CPAL_I2C_StructInit(&I2C1_DevStructure);`
  - `I2C1_DevStructure.CPAL_Direction = CPAL_DIRECTION_RX`
  - `I2C1_DevStructure.CPAL_Mode = CPAL_MODE_SLAVE`
  - `CPAL_I2C_Init(&I2C1_DevStructure);`
- Set all fields of the structure to required values.

After setting the configuration structure values, user application should call `CPAL_I2C_Init()` function in order to configure the I2C peripheral and all related peripherals (I/Os, interrupts, DMA, clocks ...) with the required settings.

**Caution:** The fields `pCPAL_TransferTx` and `pCPAL_TransferRx` are set by default to local structures with null pointers. In order to avoid issues due to memory overflow or addressing errors, these two fields should be set to point to valid structures declared in the user application.

When the device has to be stopped, the `CPAL_I2C_DeInit()` function can be called in order to free all resources used by this peripheral (I/Os, interrupts, DMA ...). In this case, the configuration structure keeps the last used values.

## 4.3 Communication

After the configuration phase, peripherals are ready to be used for communication.

Before starting to communicate with an external device, the application may check its availability on the bus using the function `CPAL_I2C_IsDeviceReady()`. If this function returns the `CPAL_PASS` value, then the external device is ready to communicate. Otherwise, the external device is not ready or the bus is not free (device error may be set in this case and the related callback may be used to manage the error).

Then to send or receive data, follow the steps below:

- (Re-)configure the structures pointed by `pCPAL_TransferTx/ pCPAL_TransferRx` with the valid values of: buffer pointer, number of data and optional addresses. (refer to [Table 8: CPA\\_TransferTypeDef structure fields](#) for more details). If the structure is

already prepared or when DMA circular mode option is enabled, there is no need to perform this operation.

- Check the state of the peripheral (wCPAL\_State field of the I2Cx\_DevStructure). If this state is different from CPAL\_STATE\_READY, then either the I2C peripheral is communicating or an error occurred. In both cases, it is not possible to use the I2C peripheral in the current state. The application may call CPAL\_I2C\_DeInit() in this case to return to its default state and restart communication. It is also possible to check the state by directly calling CPAL\_I2C\_Read()/CPAL\_I2C\_Write() function: if this function returns a value different from CPAL\_PASS, then the current state does not allow communication or an error occurred, as explained above.
- Call CPAL\_I2C\_Read()/CPAL\_I2C\_Write() function to start read/write operation. After calling this function, the transfer starts through the related peripheral using the interrupts or DMA (depending on the programming model set). In addition, the application may perform other parallel tasks while the CPAL driver is handling transfer through DMA or interrupts.
- Monitor the end of transfer: this can be performed using two basic methods:
  - Directly monitor the state of the peripheral through the wCPAL\_State field. The communication is completed when the state returns to CPAL\_STATE\_READY.
  - Use the CPAL\_I2C\_TXTC\_UserCallback() and/or CPAL\_I2C\_RXTC\_UserCallback () callback functions which are called when transfer is completed in both DMA and interrupt modes
- Two other advanced possibilities exist (provide more control on data handling):
  - If DMA mode is selected, the Transfer complete interrupt callbacks CPAL\_I2C\_DMATXTC\_UserCallback() and CPAL\_I2C\_DMARXTC\_UserCallback() may be used. They are called when the DMA has completed the transfer operation (but transfer is still not completed on the I2C bus).
  - If the Interrupt mode is selected, the CPAL\_I2C\_TX\_UserCallback() and CPAL\_I2C\_RX\_UserCallback() callbacks may be used to monitor the number of remaining data.
- At the end of the transfer, a new transfer may be started, or the peripheral may be de-initialized (and free all the used resources) using the function CPAL\_I2C\_DeInit().

## 4.4 Error management

CPAL drivers aim at managing all possible types of errors in order to offer the possibility for the application to handle them and for communication recovery when possible.

There are three types of error management:

- Peripheral errors: errors managed by the peripheral (an interrupt is generated when the error occurs). These errors are monitored by the CPAL driver and the application may use Error callbacks in order to perform specific actions for each error (refer to [Section 3.3](#) for more details about the error callbacks).
- Communication errors: they cannot be detected by the peripheral (no interrupt/flag generated when the error occurs). Example: external device disconnected in the middle of a communication session, external device blocked by the last operation... These errors are detected by the CPAL driver through the timeout mechanism (refer to [Section 3.3.1](#) for more details). When a timeout is detected, the CPAL\_TIMEOUT\_UserCallback() function is called and then application may perform

through this function the necessary operations used to recover from an error and re-start communication correctly.

## 4.5 Advanced configuration

In addition to the basic configuration, some other parameters may be modified to customize the CPAL library. These parameters are related to each device family so they are located in the `cpal_i2c_hal_stm32xxxx.h` file (which is specific for each device family).

Note that a modification in this file applies to all programs using CPAL drivers.

`cpal_i2c_hal_stm32xxxx.h` file configuration is divided into independent sections:

- Select Peripheral I/O pins.
- Select TX and RX DMA Channels.
- Set Events, Errors and DMA Interrupts Preemption and Sub-Priorities.

### 4.5.1 Select peripheral I/O pins

This section allows a selection of the I/O pins which will be used for each peripheral. For each pin, set the define value to use the related configuration. Only one configuration may be used for each I/O pin.

Example:

PB6 and PB7 used for I2C SCL and SDA pins:

```
#define CPAL_I2C1_SCL_GPIO_PORT      GPIOB
#define CPAL_I2C1_SCL_GPIO_CLK      RCC_APB2Periph_GPIOB
#define CPAL_I2C1_SCL_GPIO_PIN      GPIO_Pin_6

#define CPAL_I2C1_SDA_GPIO_PORT      GPIOB
#define CPAL_I2C1_SDA_GPIO_CLK      RCC_APB2Periph_GPIOB
#define CPAL_I2C1_SDA_GPIO_PIN      GPIO_Pin_7
```

### 4.5.2 Select TX and RX DMA channels

This section is used to select which DMA channels will be used for each peripheral direction. Only one define should be used for each peripheral direction.

Example:

DMA1 Channel6 and Channel7 used for I2C1 peripheral:

```
/* I2C1 peripheral */
#define CPAL_I2C1_DMA_TX_Channel      DMA1_Channel6
#define CPAL_I2C1_DMA_RX_Channel      DMA1_Channel7
```

### 4.5.3 Set event, error and DMA interrupt priorities

This section is used to set individual interrupt channel priorities for all used interrupts. Interrupt priorities are configured relatively to an offset defined in `cpal_conf.h` file (`I2CX_IT_OFFSET_SUBPRIO` and `I2CX_IT_OFFSET_PREPRIO`).

Generally the following interrupt priority order should be applied:

For I2C peripherals: error interrupts should have the highest priority level, then DMA interrupts (allowing the application to close communication) and finally the I2C event interrupts.

If the Timeout mechanism is implemented with an interrupt channel, then make sure that its interrupt priority is higher than any other CPAL interrupt priority.

## 5 CPAL implementation example (step by step)

This section describes all steps for using and customizing CPAL library to build a project from scratch. It uses a real example: an application with the requirements described below.

- Use two I2Cs (I2C1 and I2C2) to control an EEPROM memory and a temperature sensor (each on separate I2C bus).
- EEPROM memory is used for read/write at fixed addresses.
- Temperature sensor has a unique register (temperature value).
- Both interfaces are used simultaneously.
- EEPROM interface uses DMA mode.
- Temperature Sensor interface uses Interrupt mode.

*Note: The CPAL package already provides ready-to-use EEPROM and Temperature Sensor drivers with more advanced features, as well as examples showing how to use them. This section just provides illustrating implementation example from scratch.*

### 5.1 Starting point

The typical starting point is the template project provided within the CPAL package (Project\STM32\_CPAL\_Template\). This folder contains all needed template files as well as the project files for different IDEs.

In this folder, three files should be modified:

- cpal\_conf.h: this file is updated according to the needs of the application (in order to reduce code size and meet the required features)
- cpal\_usercallback.c: this file is updated to implement the functions needed by the user application and that will be called by CPAL drivers.
- main.c: this file contains the main program of the application.

### 5.2 cpal\_conf.h

In order to optimize the code size of the CPAL library, this file is used to disable the unused features.

Section 1:

The application needs 2 I2Cs. Consequently, CPAL\_USE\_I2C1 and CPAL\_USE\_I2C2 defines are kept enabled.

Section 2:

Slave mode is not needed. Consequently, CPAL\_I2C\_SLAVE\_MODE can be commented.

Both DMA and Interrupt modes are needed. But the application does not need to read less than one byte from the EEPROM memory. So CPAL\_I2C\_DMA\_PROGMODEL and CPAL\_I2C\_IT\_PROGMODEL must be kept enabled but CPAL\_I2C\_DMA\_1BYTE\_CASE can be commented.

Both I2Cs use simple 7-bit addressing. Consequently, CPAL\_I2C\_10BIT\_ADDR\_MODE can be commented. In the same way, if EEPROM locations and the Temperature Sensor

register are addressed through 8 bits only (memory size < 0xFF)n there is no need for CPAL\_16BIT\_REG\_OPTION. It can then be commented.

#### Section 3:

If no specific error management is required, the application may just reset the system whatever the error type. Consequently, USE\_MULTIPLE\_ERROR\_CALLBACK can be commented. Then CPAL\_I2C\_ERR\_UserCallback define should be uncommented and CPAL\_I2C\_ERR\_UserCallback define should be commented.

For EEPROM, DMA mode is used for both directions (read and write) and for Temperature Sensor, Interrupt mode is used only in reception direction. So, it is easier to use the callbacks common to DMA and Interrupt: CPAL\_I2C\_TXTC\_UserCallback and CPAL\_I2C\_RXTC\_UserCallback defines should then be commented.

#### Section 4:

No need to modify this section for this application. Note that it is strongly recommended to use CPAL\_TIMEOUT\_UserCallback function. For this application, it could just reset the system.

In other cases, if the application is implemented with RTOS structure or if the SysTick timer is already used for other purposes, then this section can be modified as follows:

Define new macro for the timeout initialization (\_CPAL\_TIMEOUT\_INIT) and the timeout de-initialization (\_CPAL\_TIMEOUT\_DEINIT). Example: use the TIM3 timer to generate regular ticks interrupts managing timeout mechanism:

```
#define _CPAL_TIMEOUT_INIT()      APP_InitTimer() /* No initialization needed */
#define _CPAL_TIMEOUT_DEINIT()   APP_DeInitTimer() /* No deinitialization needed */
#define CPAL_I2C_TIMEOUT_Manager TIM3_IRQHandler /* Use the IRQ handler of TIM3 */
void APP_InitTimer(void);        /* User function declaration */
void APP_DeInitTimer(void);      /* User function declaration */
```

In this case, two functions should be implemented in the user application: APP\_InitTimer() and APP\_DeInitTimer(). They can be implemented using standard peripheral library drivers (refer to the Timer TimeBase example for more details).

Finally, adjust the values of CPAL\_TIMEOUT\_WWW (where WWW is the I2C event i.e. SB, ADDR ...) according to the timebase value.

#### Section 5:

No need to modify this section for this application.

In other cases, when multiple interrupts are managed by the application, the interrupt group and priority configuration may be adjusted in this section.

#### Section 6:

No need to modify this section for this application.

In other cases, where debug tools are not available, the CPAL debug feature may be enabled using the USART or LCD interface. For the USART and LCD, you need to enable the define CPAL\_DEBUG and then, in the user application, the printf function must be retargeted to the USART or LCD.

Example for USART interface:

```
#ifndef __GNUC__
/* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf
   set to 'Yes') calls __io_putchar() */
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */

/**
 * @brief Retargets the C library printf function to the USART.
 * @param None
 * @retval None
 */
PUTCHAR_PROTOTYPE
{
    /* Place your implementation of fputc here */
    /* e.g. write a character to the USART */
    USART_SendData(EVAL_COM1, (uint8_t) ch);

    /* Loop until the end of transmission */
    while (USART_GetFlagStatus(EVAL_COM1, USART_FLAG_TC) == RESET)
    {}

    return ch;
}
```

*Note: The debug feature uses a large amount of Flash memory space due to the debug message definitions. Also, if the USART/LCD interface is too slow, it may significantly impact the behavior of the CPAL driver (in some cases, communication may be corrupted). So make sure that the debug interface (USART/LCD/..) is fast enough to avoid impacting the behavior of the CPAL driver.*

### 5.3 cpal\_usercallback.c

In this file, all used callbacks should be implemented. In this application, 4 callbacks are used: CPAL\_I2C\_ERR\_UserCallback, CPAL\_I2C\_TXTC\_UserCallback, CPAL\_I2C\_RXTC\_UserCallback and CPAL\_TIMEOUT\_UserCallback.

As detailed above, the error callbacks just have to reset the system in case of errors:

```
/**
 * @brief User callback that manages the Timeout error.
 * @param pDevInitStruct .
 * @retval None.
 */
uint32_t CPAL_TIMEOUT_UserCallback(CPAL_InitTypeDef* pDevInitStruct)
{
    NVIC_SystemReset(); /* Generate a system reset */
    return CPAL_PASS; /* This statement will not be reached */
}

/**
 * @brief User callback that manages the I2C peripheral errors.
 * @note Make sure that the define USE_SINGLE_ERROR_CALLBACK is uncommented in
 *       the cpal_conf.h file, otherwise this callback will not be functional.
 * @param pDevInitStruct.
 * @param DeviceError.
 * @retval None
 */
```

```

    */
uint32_t CPAL_I2C_ERR_UserCallback (CPAL_InitTypeDef* pDevInitStruct)
{
    NVIC_SystemReset(); /* Generate a system reset */
    return CPAL_PASS; /* This statement will not be reached */
}

```

The Rx and Tx transfer complete callbacks may be used to inform the user application layer that the transfer is completed in order to start a new transfer:

```

/**
 * @brief Manages the End of Rx transfer event.
 * @param pDevInitStruct .
 * @retval None.
 */
void CPAL_I2C_RXTC_UserCallback (CPAL_InitTypeDef* pDevInitStruct)
{
    APPTransferComplete = 1; /* assuming that APPTransferComplete is global variable
                               used by the application */
    APP_ToggleLED(); /* User application function that informs user of the end of
                       an operation by toggling LEDs */
    return;
}

/**
 * @brief Manages the End of Tx transfer event.
 * @param pDevInitStruct .
 * @retval None.
 */
void CPAL_I2C_TXTC_UserCallback (CPAL_InitTypeDef* pDevInitStruct)
{
    APPTransferComplete = 1; /* assuming that APPTransferComplete is global variable
                               used by application
    APP_ToggleLED(); /* User application function that informs user of the end of
                       an operation by toggling LEDs */
    return;
}

```

All other non-used callbacks should be kept commented.

## 5.4 main.c

In this file, the application uses the CPAL driver functions to configure the I2C interface and then control it to communicate with the EEPROM and Temperature Sensor.

Variables and structures:

In order to send and receive data, the application needs local transfer structures:

```

/* CPAL local transfer structures: 2 structures for EEPROM and 1 Rx structure for
Temperature Sensor */
CPAL_TransferTypeDef sEERxStructure, sEETxStructure, sTSRxStructure;
uint8_t tEERxBuffer[255], tEETxBuffer[255], TSData;

```



**Configuration:**

First, the transfer and configuration structures should be filled:

```

/* Initialize local Reception structures for EEPROM */
sEERxStructure.pbBuffer = tEERxBuffer;    /* EEPROM Receive buffer */
sEERxStructure.wAddr1 = EE_ADDRESS;        /* EEPROM Address */

/* Initialize local Transmission structures for EEPROM */
sEETxStructure.pbBuffer = tEETxBuffer;    /* EEPROM Tx buffer */
sEETxStructure.wAddr1 = EE_ADDRESS;        /* EEPROM Address */

/* Initialize local Transmission structures for Temperature Sensor */
sTSRxStructure.pbBuffer = TSData          /* Temperature Sensor Receive buffer:
one byte needed */
sTSRxStructure.wAddr1 = TS_ADDRESS;        /* Temperature Sensor Address */

```

Then, the CPAL I2C structures should be initialized (note that for each I2C peripheral, a structure is already declared by CPAL drivers and exported as extern to application layer):

```

/* Configure the peripheral structure */
CPAL_I2C_StructInit(&I2C1_DevStructure); /* Set all fields to default values */
I2C1_DevStructure.CPAL_Mode = CPAL_MODE_MASTER;
I2C1_DevStructure.wCPAL_Options = CPAL_OPT_NO_MEM_ADDR;
I2C1_DevStructure.CPAL_ProgModel = CPAL_PROGMODEL_DMA;
I2C1_DevStructure.pCPAL_I2C_Struct->I2C_ClockSpeed = 100000; /* 100 KHz */
I2C1_DevStructure.pCPAL_TransferRx = &sEERxStructure;
I2C1_DevStructure.pCPAL_TransferTx = &sEETxStructure;
/* Initialize CPAL peripheral with the selected parameters */
CPAL_I2C_Init(&I2C1_DevStructure);

/* Configure the peripheral structure */
CPAL_I2C_StructInit(&I2C2_DevStructure); /* Set all fields to default values */
I2C2_DevStructure.CPAL_Mode = CPAL_MODE_MASTER;
I2C2_DevStructure.CPAL_ProgModel = CPAL_PROGMODEL_INTERRUPT;
I2C2_DevStructure.pCPAL_I2C_Struct->I2C_ClockSpeed = 50000; /* 50 KHz */
I2C2_DevStructure.pCPAL_TransferRx = &sEERxStructure;
I2C2_DevStructure.pCPAL_TransferTx = pNULL; /* Not needed */
/* Initialize CPAL peripheral with the selected parameters */
CPAL_I2C_Init(&I2C2_DevStructure);

```

## Communication:

In the example below, each peripheral communication will be managed in separate code sections. Then each section could be implemented in a single infinite loop, in separate interrupt handlers, in tasks ... This only depends on the application architecture.

```

/* Write 100 data to EEPROM at address 0x30 */
sEETxStructure.wNumData = 100; /* Number of data to be written */
sEETxStructure.wAddr2 = 0x30; /* Address into EEPROM */
if (CPAL_I2C_Write(&I2C1_DevStructure) != CPAL_PASS)
{
/* I2C bus or peripheral is not able to start communication: Error management */
}

/* Wait the end of transfer */
while(I2C1_DevStructure-> CPAL_State != CPAL_STATE_READY)
{
/* Read temperature value */
sTSRxStructure.wNumData = 1;
if (CPAL_I2C_Read(&I2C2_DevStructure) != CPAL_PASS)
{
/* I2C bus or peripheral is not able to start communication: Error management */
}

/* Wait for the end of transfer */
while(I2C2_DevStructure-> CPAL_State != CPAL_STATE_READY)
{
}

/* Check the temperature value range */
if (sTSRxStructure.pbBuffer[0] == CRITICAL_VALUE)
{
/* Stop communication or switch to low power mode */
}
}

/* Read back the 100 data from EEPROM at address 0x30 */
sEERxStructure.wNumData = 100; /* Number of data to be read */
sEERxStructure.wAddr2 = 0x30; /* Address into EEPROM */
if (CPAL_I2C_Read(&I2C1_DevStructure) != CPAL_PASS)
{
/* I2C bus or peripheral is not able to start communication: Error management */
}

/* Wait the end of transfer */
while(I2C1_DevStructure-> CPAL_State != CPAL_STATE_READY)
{
/* Application may perform other tasks while CPAL read operation is ongoing */
}

/* At this point, data has been received, they can be used by the application
(compare, process...) */

```

## 6 CPAL Examples

In addition to the template project, the CPAL package provides a set of examples for each peripheral, aiming at providing different levels of implementation complexity.

This release comes with three examples running on STM3210C-EVAL RevB (STM32F10x Connectivity line), STM32100E-EVAL RevB (STM32F10x High-Density Value line), STM32L152-EVAL RevB (STM32L1xx Ultra Low Power Medium-Density), STM322xG-EVAL RevB (STM32F2xx series) and STM324xG-EVAL RevA (STM32F40x series) evaluation boards and can be easily tailored to any other supported device and development board."

The following table shows the hardware resources used in these four examples.

**Table 23. Hardware resources used in CPAL examples**

Example	Board	Used Resources					
		Touch Screen <sup>(1)</sup>	Buttons	EEPROM <sup>(2)</sup>	Temp Sense <sup>(3)</sup>	LCD	LED
Advanced Example	STM3210C_EVAL	X	X	X	X	X	X
	STM32100E_EVAL						
	STM322xG_EVAL						
	STM324xG_EVAL						
Basic EEPROM	STM3210C_EVAL			X			X
	STM32100E_EVAL						
	STM32L152_EVAL <sup>(4)</sup>						
	STM322xG_EVAL						
	STM324xG_EVAL						
	STM32L152D_EVAL						
Two Boards	STM3210C_EVAL		X			X	X
	STM32100E_EVAL						
	STM32L152_EVAL						
	STM322xG_EVAL						
	STM324xG_EVAL						
	STM32L152D_EVAL						
Two Boards listen mode	STM3210C_EVAL		X			X	X
	STM32100E_EVAL						
	STM32L152_EVAL						
	STM322xG_EVAL						
	STM324xG_EVAL						
	STM32L152D_EVAL						

1. TouchScreen is connected to STM32 via IOExpander.
2. M24C64 EEPROM is mounted in STM3210C\_EVAL, STM32100E\_EVAL, STM322xG\_EVAL and STM324xG\_EVAL. M24LR64 EEPROM is mounted in STM32L152D\_EVAL.
3. IOExpander Tempsense is used with STM3210C\_EVAL, STM322xG\_EVAL and STM324xG\_EVAL.
4. With STM32L152\_EVAL an external EEPROM must be connected to I2C1.

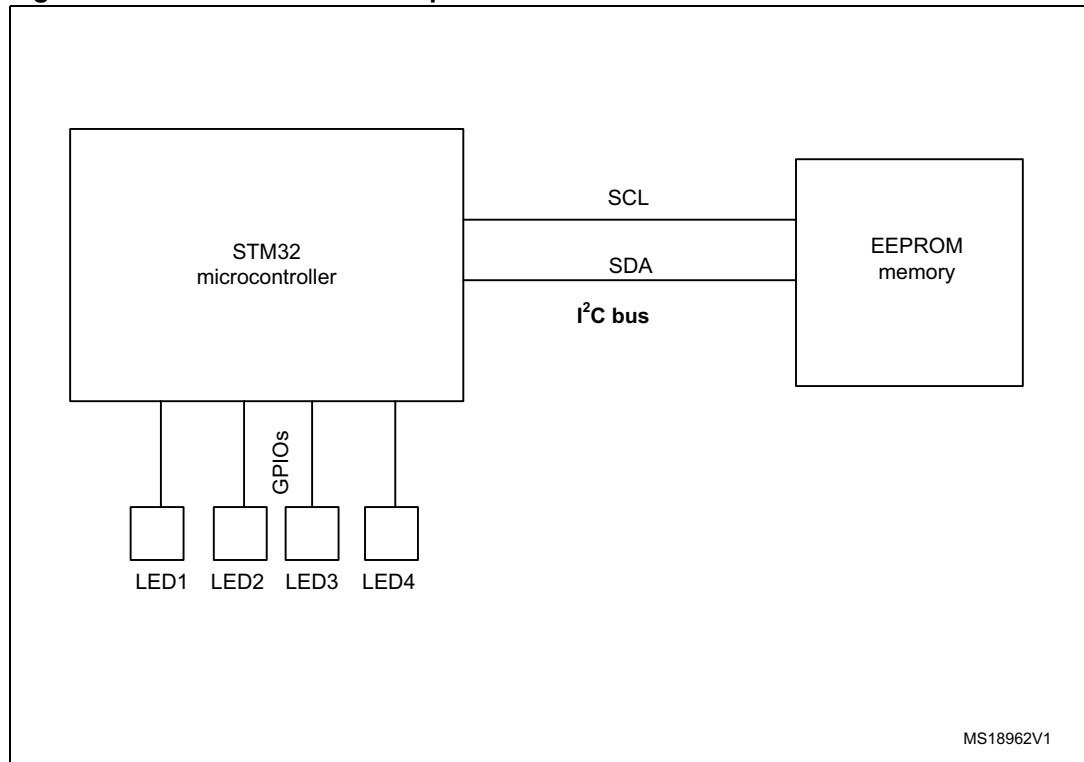
## 6.1 Basic EEPROM example

To communicate with EEPROM memories, the user application may directly use the CPAL functions. Since the memory management requires some software overhead and in order to simplify and ease the use of CPAL as communication interface for EEPROM memories, a specific EEPROM memory driver is provided into the CPAL package:

Utilities\STM32\_EVAL\_CPAL\Common\stm32\_eval\_i2c\_ee\_cpal.c/.h

Then the use of this driver is illustrated through the EEPROM example. In this example, the following actions are performed:

**Figure 9. Basic EEPROM example architecture**



The EEPROM memory is written with a constant buffer stored in the internal Flash memory. When the write operation is completed, the content of the EEPROM is read back (into a buffer located in the internal SRAM) and verified (it should match the written values). Then a second write operation is performed in a different location of the EEPROM. It is then read back and verified with written one. Meanwhile, a timer is configured to generate periodic interrupts in which a LED blinks.

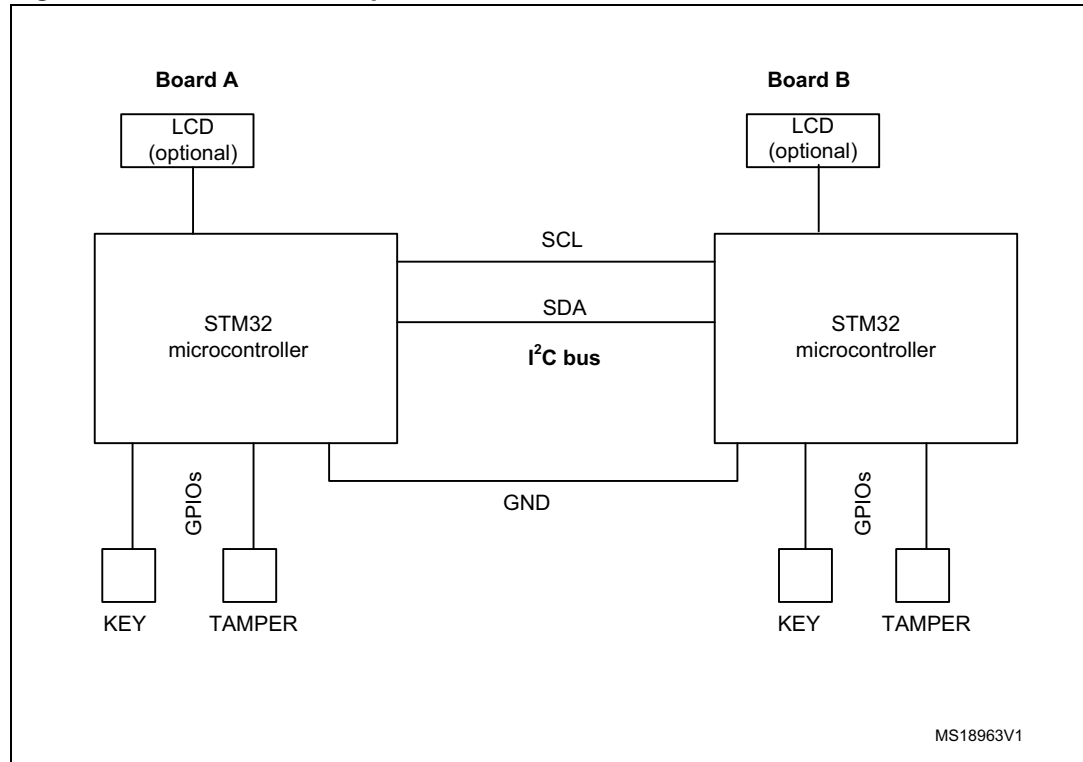
For more details about hardware requirements and the supported devices and boards, refer to the readme file located in the example directory.

## 6.2 Two-board example

This example shows how two microcontrollers may communicate on the same I2C bus using the CPAL drivers.

This example shows how the I2C bus arbitration between multiple masters can be managed with the CPAL library and how the errors can be managed and recovered using the CPAL error callbacks and timeout mechanism.

**Figure 10. Two-board example architecture**



The two STM32 devices are configured as Slaves at the example startup. When the user pushes Key or Tamper push buttons, the activated STM32 device switches to Master mode and remains in this mode till the next reset. It periodically sends status messages to the other Slave. The period of these messages depends on the last pushed button. If a device receives a message while it is not configured yet in Master mode, it remains in Slave mode till the next reset.

When a button is pushed, the activated STM32 device sends a message specific to each button and different from the status message.

The message reception and transmission status as well as the error detection and recovery status are displayed on the LCD screen.

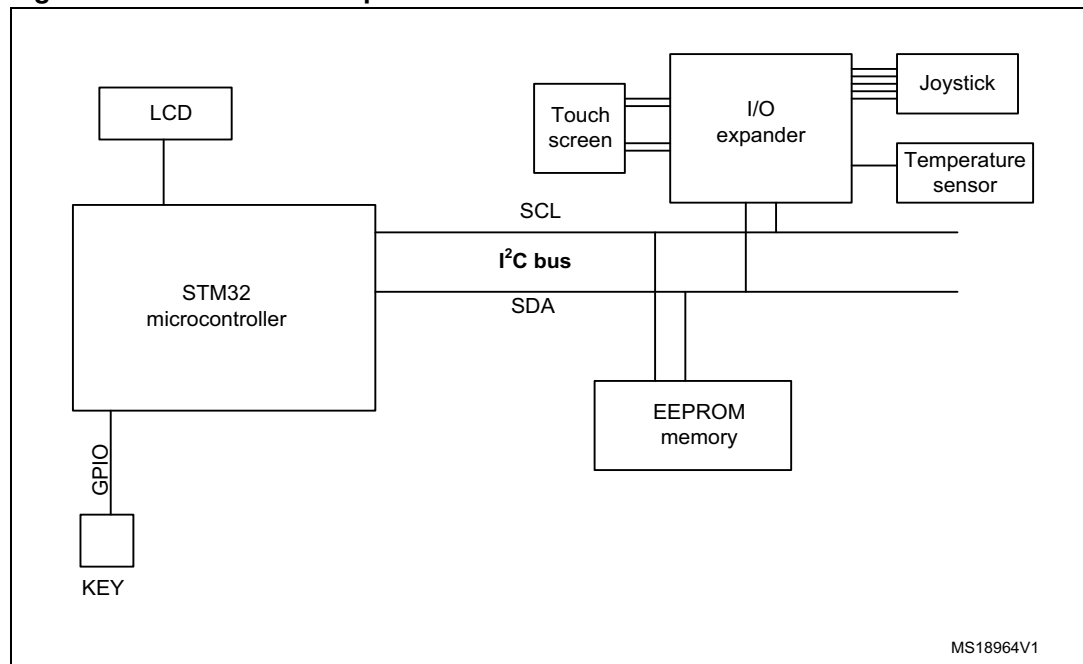
For more details about the hardware requirements and the supported devices and boards, refer to the readme file located in the example directory.

## 6.3 Advanced example

This example shows how to implement more complex modules using CPAL drivers simultaneously and for different controls. The main feature shown in this example is the possibility to perform different tasks simultaneously on same I2C bus (i.e. while the EEPROM memory is being written or read from, the touch screen and the joystick interfaces

remain functional. Besides, other periodic interrupts are enabled and update the status of the LCD or blinking LEDs ...).

**Figure 11. Advanced example architecture**



In this example the STM32 device controls the following modules through the I2C interface:

- EEPROM memory: the values of temperature provided from the Temperature Sensor module are stored into the EEPROM memory. The user may read the history of stored values.
- Joystick control (through I/O Expander): controls the menu navigation in an asynchronous way.
- Touch Screen control (through I/O Expander): controls a touch zone displaying four rectangles. For each rectangle, when it is touched, a specific LED blinks and a message is displayed on the LCD screen.
- Temperature Sensor (through I/O Expander): periodically (using a timer interrupt) the temperature Sensor is read and the temperature value is stored into the EEPROM memory. For each temperature value, a time value is associated (using RTC peripheral) and stored at the same level.

For more details about the hardware requirements and the supported devices and boards, refer to the readme file located in the example directory.

## 6.4 Two-board Listen mode example

This example shows how two microcontrollers can communicate on the same I2C bus using the CPAL drivers and how a slave device communicates in Listen mode. Also it shows how the I2C bus arbitration between multiple masters can be managed with the CPAL library and how errors can be managed and recovered using the CPAL error callbacks and timeout mechanism.

This example use the same architecture as the Two-board example in [Section 6.2](#) (refer to [Figure 9](#) for more details).

The two STM32 devices are configured as slaves in listen mode at the startup of the example.

When the user pushes the Key button the activated STM32 device switches to master transmitter mode.

It periodically sends a request to transmit status messages to the other slave. When the user pushes the Tamper button the activated STM32 device switches to master receiver mode. It periodically sends a request to receive status messages from the other slave. If a device receives a message while it is not configured yet in master mode, it remains in slave mode till the next reset.

When a button is pushed, the activated STM32 device sends or receives a message specific to each button and different from the status message.

The message reception and transmission status as well as the error detection and recovery status are displayed on the LCD screen.

For more details about the hardware requirements and the supported devices and boards, refer to the readme file located in the example directory.

## 7 Memory footprint of CPAL components

The table below details the footprint of each CPAL component in terms of Flash size and RAM size.

These figures have been determined using the IAR EWARM 6.1 tool with High Size optimization level, and are valid for STM32F10x devices. The footprint may change slightly for other devices.

All options are controlled by the defines listed in table below. All these defines are located in the `cpal_conf.h` file (that should be extracted from `cpal_conf_template.h` file)

Notes

- 1 "Option" means all controlling defines listed below except `CPAL_USE_I2Cx` (where *x* can be 1 or 2 for the I2C peripheral instance).
- 2 When not specified, the options are independent of the number of supported devices (`CPAL_USE_I2Cx`)
- 3 All options are enabled by uncommenting the related define and disabled by commenting the related define.
- 4 The main configurations (all options disabled and all options enabled) are grayed.
- 5 Except for addressing mode options (`CPAL_16BIT_REG_OPTION` and `CPAL_I2C_10BIT_ADDR_MODE`) and critical callback option (`USE_CPAL_CRITICAL_CALLBACK`) and `CPAL_I2C_DMA_1BYTE_CASE` option: for all other option groups, it is mandatory to select at least one of the provided options (i.e. in Mode option group, it is mandatory to enable Master mode or Slave mode or both. It is not allowed to disable both).

**Table 24. Memory footprint of CPAL components**

Options	Define	Code Size (Bytes)	
		Flash	RAM
CPAL Core (All options disabled)	1 Device	1142	80
	2 Devices	1198	112
Mode Option <sup>(1)</sup>	Master	3440	0
	Slave without listen mode	1412	0
	Slave with listen mode	664	0
Programming Model Option <sup>(1)</sup>	DMA	1700	68
	Interrupt	1148	0
	1Byte DMA	768	0
Addressing Mode Option <sup>(1)</sup>	16 Bit	216	0
	10 Bit	292	0



Table 24. Memory footprint of CPAL components (continued)

Options		Define	Code Size (Bytes)	
			Flash	RAM
Communication closedown Method (Master RX)	Method 1	CPAL_I2C_CLOSECOM_METHOD1	84	0
	Method 2	CPAL_I2C_CLOSECOM_METHOD2	8	0
Critical Callback		USE_CPAL_CRITICAL_CALLBACK	8	0
All Options enabled (Full CPAL set)	1 Device	CPAL_USE_I2C1 All other options enabled	6070	148
	2 Devices	CPAL_USE_I2C1 CPAL_USE_I2C1 All other options enabled	6050	180
Master and Slave Mode disabled, All other options enabled <sup>(2)</sup>		CPAL_I2C_MASTER_MODE CPAL_I2C_SLAVE_MODE disabled	1994	180
Master Mode disabled with Slave without Listen mode, All other options enabled <sup>(2)</sup>		CPAL_I2C_MASTER_MODE disabled	3406	180
Master Mode disabled with Slave in Listen mode, All other options enabled <sup>(2)</sup>		CPAL_I2C_MASTER_MODE disabled	2658	180
Slave Mode disabled, All other options enabled <sup>(2)</sup>		CPAL_I2C_SLAVE_MODE disabled	5434	180
DMA Mode disabled, All other options enabled <sup>(2)</sup>		CPAL_I2C_DMA_PROGMODEL disabled	4350	112
Interrupt Mode disabled, All other options enabled **		CPAL_I2C_IT_PROGMODEL disabled	5670	180
1Byte DMA disabled, All other options enabled <sup>(2)</sup>		CPAL_I2C_DMA_1BYTE_CASE disabled	4902	180
16 Bit Addressing mode disabled, All other options enabled <sup>(2)</sup>		CPAL_16BIT_REG_OPTION disabled	5834	180
10 Bit Addressing mode disabled, All other options enabled <sup>(2)</sup>		CPAL_I2C_10BIT_ADDR_MODE disabled	5758	180
Communication Closedown Method 1 disabled, All other options enabled <sup>(2)</sup>		CPAL_I2C_CLOSECOM_METHOD1 disabled	5966	180
Communication Closedown Method 2 disabled, All other options enabled <sup>(2)</sup>		CPAL_I2C_CLOSECOM_METHOD2 disabled	5542	180
Critical Callback disabled, All other options enabled <sup>(2)</sup>		USE_CPAL_CRITICAL_CALLBACK disabled	6042	180

1. Only option code size is taken into consideration (considering configuration for 2 devices).
2. All options enabled for 2 devices.

## 8 FAQ

This section gathers some of the most frequent questions CPAL users may ask. It gives some solutions and tips:

**Table 25. FAQ**

No.	Question	Possible answers / solutions
<b>Topic 1: General</b>		
1	Why would I use the CPAL driver rather than a standard peripheral library?	<p>The main advantage of using the CPAL library is the ease of use: you can use CPAL driver to control the I2C interface without any knowledge about I2C protocol. CPAL library also offers a higher level of abstraction allowing the “transparent” management of:</p> <ul style="list-style-type: none"> <li>- Hardware components used by communication peripherals (I/Os, DMA, interrupts ...)</li> <li>- Transfer buffers and status (managed through independent structures).</li> <li>- Peripheral states (i.e. event management for I2C peripheral ...)</li> <li>- Error detection and recovery when possible (through peripheral error detection and timeout mechanism).</li> <li>- Bug fixes and workaround selection.</li> <li>- Different device families.</li> </ul>
2	What is the cost of using CPAL drivers in term of code size and performance?	<p>As a generic driver, CPAL may involve a significant firmware overhead. But different customization levels allow you to reduce code size by removing any unused feature.</p> <p>CPAL drivers use the standard peripheral drivers only for initialization phase. For the communication phase, only direct register access (using macros) is used, which improves significantly the driver performance.</p>
3	How many files should I modify to configure the CPAL drivers?	<p>CPAL library offers multiple levels of customization:</p> <ul style="list-style-type: none"> <li>- <b>0 file:</b> No file needs to be modified: you can use the CPAL template or one of the provided examples without any modification in configuration files. In this case, the code size of the application may be too large. To reduce CPAL code size you may check the next case.</li> <li>- <b>1 file:</b> In most cases, only one file needs to be modified: <code>cpal_conf.h</code>. You can modify this file by disabling unused features, or adjusting some parameters (i.e. interrupt priority groups, timeout mechanism ...).</li> <li>- <b>2 files:</b> In addition to the <code>cpal_conf.h</code> file, you can customize the CPAL library hardware layer by modifying the file <code>cpal_i2c_stm32xxxx.h</code>. Through this file you can modify the I/O selection, the DMA channels, the interrupt priorities...</li> </ul>
4	Which header files should I include in my application in order to use the CPAL library?	Only <code>cpal_i2c.h</code> file has to be included.

**Table 25. FAQ (continued)**

No.	Question	Possible answers / solutions
<b>Topic 2: Configuration</b>		
5	How many fields are mandatory to fill for the CPAL initialization structure?	<p>The easiest way is to call the function <code>CPAL_I2C_StructInit()</code> to initialize the structure with default values. Then you have to set the following fields: <code>pCPAL_TransferTx</code> and/or <code>pCPAL_TransferRx</code> should be filled with pointers to the Tx/Rx transfer structures. These structures should be updated during execution with new values of buffers and addresses.</p> <p>For all other fields, you can keep default values in most cases (check <a href="#">Section 2.3.2</a> for more details on default values).</p>
6	I use more than one I2C peripheral and they are not configured correctly after I call the <code>CPAL_I2C_StructInit()</code> <code>CPAL_I2C_Init()</code> functions.	<p>The CPAL drivers use a unique structure for each peripheral. These structures are exported as external variables to the user application code. In this structure, some fields are pointers to initialization structures. When you call <code>CPAL_StructInit()</code> functions, all fields are set to default values, but pointers are set as follows:</p> <ul style="list-style-type: none"> <li>– Transfer structure pointers are set to Null value.</li> <li>– I2C initialization structure pointer (<code>pCPAL_I2C_Struct</code>) is set to a local structure containing default values for the I2C. So this structure will be used by all I2C peripherals at the same time. Any modification on it will affect all I2C peripherals.</li> </ul> <p>To make sure that each peripheral is correctly configured, declare a local structure for each one and assign it to the field <code>pCPAL_I2C_Struct</code> after calling <code>CPAL_I2C_StructInit()</code> function.</p>
<b>Topic 3: Interrupts</b>		
7	My program uses multiple interrupts. When adding CPAL drivers to the application, some/all interrupts do not work correctly or do not work at all.	<p>Priority group configuration may be modified by the CPAL driver. To make sure that the priority group configuration corresponds to what you expect, set its value in <code>cpal_conf.h</code> section 5 (<i>CPAL_NVIC_PRIORGROUP</i>) and remove any other settings of this parameter in your application.</p> <p>CPAL Priority level is too high compared to other application interrupts that need to be processed faster. In this case, modify the offset value of the CPAL interrupt in <code>cpal_conf.h</code> file section 5 (i.e. <i>I2C1_IT_OFFSET_SUBPRIO</i>)</p>
8	I cannot find I2C interrupt handler in the <code>stm32xxx_it.c</code> file, and if I add it, I have compiler warnings/errors.	CPAL drivers already declare and implement internally all needed interrupt handlers for the communication peripheral (i.e. I2C: <code>I2Cx_EV_IRQHandler</code> and <code>I2Cx_ER_IRQHandler</code> ).
9	When I implement CPAL drivers into my application, I have warnings/errors related to I2C interrupt	There is no need for additional configuration. You just have to use the CPAL callbacks provided for this peripheral in <code>cpal_usercallbacks.c</code> file and comment the related callback define in <code>cpal_conf.h</code> file section 3.

Table 25. FAQ (continued)

No.	Question	Possible answers / solutions
10	I need to use a DMA interrupt handler but if I implement it, I have compiler warnings/errors.	As for communication peripheral IRQ handlers, CPAL drivers already declare and implement internally all needed interrupt handlers for the used DMA channels (i.e. DMAx_Channely_IRQHandler if DMAx_Channely is used by CPAL drivers). This is true only if DMA configuration is enabled in cpal_conf.h file section 2 ( <i>CPAL_I2C_DMA_PROGMODEL</i> ).
11	When I implement CPAL drivers into my application, I have warnings/errors related to DMA interrupt handlers.	If these handlers are needed only for the communication peripheral controlled by the CPAL driver, then there is no need for any additional configuration. If you want to use this handler, you can implement related DMA callbacks in cpal_usercallback.c file ( <i>CPAL_I2C_DMATXTC_UserCallback ...</i> ) If these handlers are needed for other purposes, then try to change the configuration of the DMA channels in cpal_i2c_stm32xxxx.h file configuration section 2 in order to free the requested channels.
12	How /Why can I select the right interrupt priority level for communication peripherals?	I2C peripherals require specific interrupt scheme: Error interrupts should have the highest priority (and should be able to interrupt other processes). Then DMA interrupts (if DMA mode is enabled) have the second priority level because they are used for closing communication and this phase is time-sensitive. Finally I2C event interrupts may have the lowest priority and may be interruptible. Regarding other interrupts, the interrupt order and grouping entirely depend on application requirements and environment.
<b>Topic 3: Hardware</b>		
13	Which STM32 devices are supported by the CPAL Library?	Refer to the Release Notes of the CPAL package for the complete list of supported STM32 devices.

## 9 Revision history

**Table 26. Document revision history**

Date	Revision	Changes
17-Mar-2011	1	Initial release.
20-Jun-2011	2	Added <a href="#">Table 23: Hardware resources used in CPAL examples on page 59</a>
06-Feb-2013	3	Updated for STM32F4xx devices. Added <a href="#">Table 1: Applicable products</a> . Modified title, <a href="#">Introduction</a> , <a href="#">Table 11: CPAL option fields</a> , <a href="#">Table 12: wCPAL_Options field values</a> , <a href="#">Table 18: CPAL low layer interface function description</a> and <a href="#">Table 23: Hardware resources used in CPAL examples</a> .

**Please Read Carefully:**

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST's terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

**UNLESS OTHERWISE SET FORTH IN ST'S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.**

**UNLESS EXPRESSLY APPROVED IN WRITING BY TWO AUTHORIZED ST REPRESENTATIVES, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER'S OWN RISK.**

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2013 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

[www.st.com](http://www.st.com)