UM1566
User manual

STM32F0xx and STM32F3xx
I2C communication peripheral application library (CPAL v2)

Introduction

The purpose of this document is to explain the architecture and the implementation of the I^2C CPAL v2 (Communication Peripheral Application Library).

The CPAL v2 provides the same high layer API implemented in CPAL v1, thus you can migrate between these two libraries easily.

The CPAL v1 is a library which provides API for I^2C peripherals of STM32F1, STM32F2, STM32F4 and STM32L1 devices. While CPAL v2 provides API for I^2C peripherals of STM32F0 and STM32F3 devices.

It provides CPAL drivers for communication peripherals and some examples showing how to use and customize the CPAL drivers.

The CPAL v2 drivers and examples are supplied within STM32F0xx, STM32F37x and STM32F30x Standard Peripherals library; the drivers are under *Libraries\STM32xxxx_CPAL_Driver* and the examples under *Project\STM32xxxx_StdPeriph_Examples\I2C*.

Table 1 lists applicable products concerned by this user manual.

Table 1. Applicable products

Type	Part numbers
Microcontrollers	STM32F0xxxx and STM32F3xxxx

Contents

List of tables

List of figures

1 CPAL overview

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

CPAL v2 is a Library providing high layer API for STM32F0/F3 communication peripherals (I^2C). It provides CPAL drivers for each device, some illustrating examples showing how to use and customize the CPAL drivers.

It mainly aims to:

- Provide intuitive, easy to use and sufficient API (Init, Deinit, Read, Write):
  - All configurations needed for the 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 structure instance for each device) holding all necessary configuration parameters (device configuration, buffers addresses and sizes…) as well as the current communication status and error codes.
- Provide efficient and complete management of device and communication errors. Device events and errors are managed by the CPAL low layer drivers and allow user to integrate easily a specific code for each event and error. Communication errors are also managed by a timeout mechanism that can be customized by user application.
- Provide high customization and integration level:
  - Several static configurations allow reducing code size when some options are not used (i.e., Reduce the number of used devices, disable management of some modes: 10-bit addressing, General call …).
  - Dynamic and easy configuration through a unique control structure allowing dynamically enabling/disabling device and communication options.
  - Several user callbacks: functions declared and called by the CPAL drivers and implemented by user application when needed. These functions allow user application to perform specific actions relative to specific communication events/errors.
- Provide device abstraction layer: CPAL library supports all STM32 device families.
- Provide efficient and simple debug feature through CPAL_DEBUG option: debug messages are sent through customizable macro. Log messages are sent at each step of the CPAL driver (this macro can be customized to send messages through serial interface, debug IDE interface, LCD screen…).
- In order to optimize CPAL driver performances, all operations, except device initialization, are preformed through direct register access.

2 CPAL architecture description

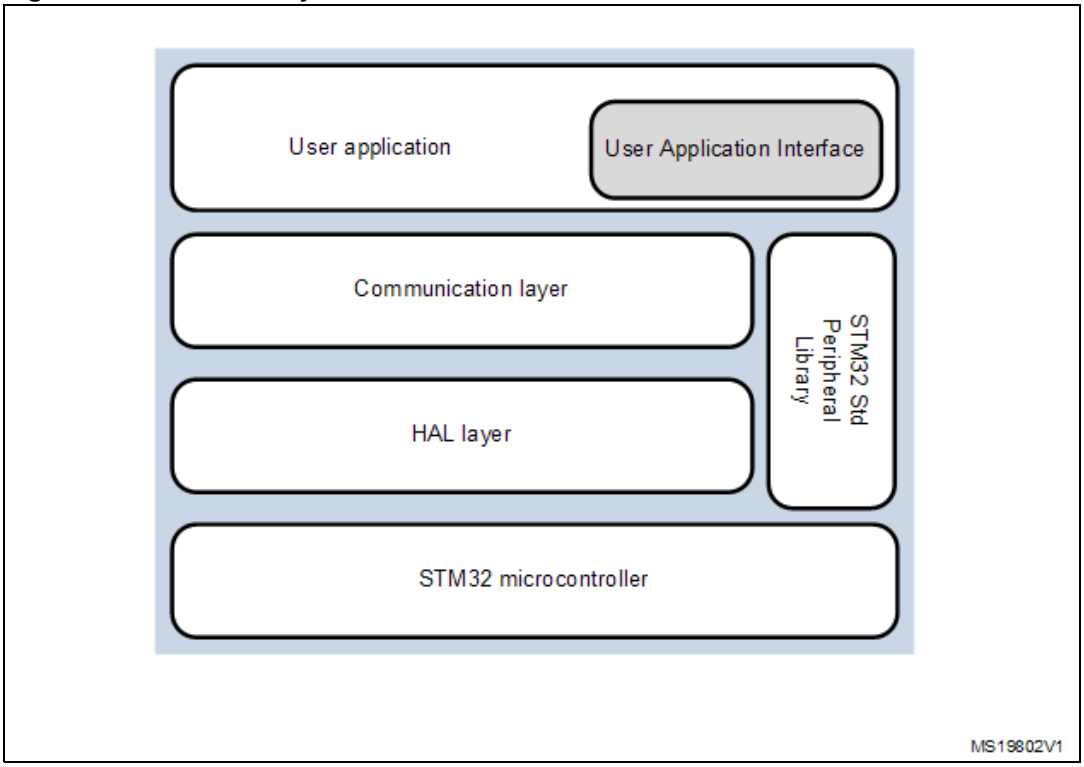
2.1 CPAL application hierarchy

The CPAL library consists of three layers:

- User Application interface: allows the customization of CPAL library besides of user callbacks implementation. This layer consists of files which user may modify according to application requirements.
- Communication Layer: contains the communication API for each device (I^2C).
- HAL Layer: is a Hardware Abstraction Layer (HAL) that allows controlling the different devices registers independently of the device family.

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

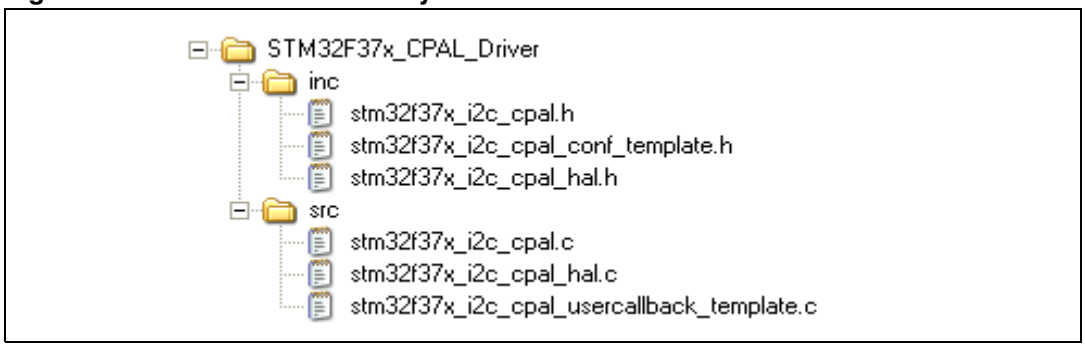
Figure 1. CPAL library architecture



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.

Each product family have a specific CPAL library package. This package consists of STM32xxxx_CPAL_Driver folder located in the Libraries repository at the same level as CMSIS (Cortex™ microcontroller software interface standard) and the STM32 standard peripheral drivers. This folder contains all the CPAL drivers and header files as well as templates for user files. Also some examples illustrating the use of CPAL drivers are located in the project folder under folder containing examples based on the standard peripheral drivers.

Figure 2. CPAL driver 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	`stm32xxxx_i2c_cpal_conf_template.h`	User file which can be modified to customize and/or configure the CPAL library drivers. This file is provided with each example.
	`stm32xxxx_i2c_cpal_usercallback_template.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	`stm32xxxx_i2c_cpal.h`	Contains the main structure definitions and the global CPAL structure declaration (as extern). It also contains all option definitions and the peripheral-specific error definitions
	`stm32xxxx_i2c_cpal.c`	This file contains the main operation functions for the peripheral (Init, DeInit, Read, Write…). It also contains all the peripheral-related interrupt handlers (peripheral and DMA interrupts) as well as the error management functions.
Hardware Abstraction Layer	`stm32xxxx_i2c_cpal_hal.c`	This file provides low layer functions specific to the device family for controlling the I^2C peripheral.
	`stm32xxxx_i2c_cpal_hal.h`	This file provides low layer configuration options (I/O 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:

- `stm32xxxx_i2c_cpal.h`
- `stm32xxxx_i2c_cpal.c`

Important notes:

- All I^2C 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 `stm32xxxx_i2c_cpal_conf.h` file.

2.2.1 CPAL main structures (stm32xxxx_i2c_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 `stm32xxxx_i2c_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 `stm32xxxx_i2c_cpal.h` file (so no need for you to declare them in your application files).

Example:

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

#ifdef CPAL_USE_I2C2
  extern CPAL_InitTypeDef I2C2_DevStructure;
#endif /* CPAL_USE_I2C2 */
…
```

The CPAL configuration structure is defined as follows:

```
typedef struct
{
CPAL_DevTypeDef            CPAL_Dev;
CPAL_DirectionTypeDef      CPAL_Direction;
CPAL_ModeTypeDef           CPAL_Mode;
CPAL_ProgModelTypeDef      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 *Table 5* 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 *Table 6* 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 *Table 7* can be assigned to this field. *Note: For I^2C, when DMA mode is enabled, the addressing phase cannot be managed by DMA but only by interrupts.*
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 *Table 8*.
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 *Table 8*.
__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 *Section 3.3*. One of the values listed in *Table 9* can be assigned to this field. *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.*

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 *Table 10* can be assigned to this field. *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.*
uint32_t	wCPAL_Options	This field allows you to configure additional options for the transfer configuration. These options are described in more detail in *Table 11*: 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 stm32xxxx_i2c_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). *Note: For I^2C peripherals, DMA cannot handle1-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.*

Table 8. CPAL_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. CPAL_TransferTypeDef structure fields

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 (for I^2C peripherals)

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).
CPAL_I2C_ERR_BERR	A bus error is detected when a START or a STOP condition is detected and is not located after a multiple of 9 SCL clock pulses. A START or a STOP condition is detected when a SDA edge occurs while SCL is high. The bus error flag is set only in case the I^2C is involved in the transfer as master or addressed slave (i.e not during address phase in slave mode). In case of a misplaced START or ReSTART detection in slave mode, the I^2C enters address recognition state as for a correct START condition. When a bus error is detected, BERR flag is set in I2C_ISR register, and an interrupt is generated if ERRIE is set in I2C_CR1 register. It is cleared by software by setting BERRCF bit.

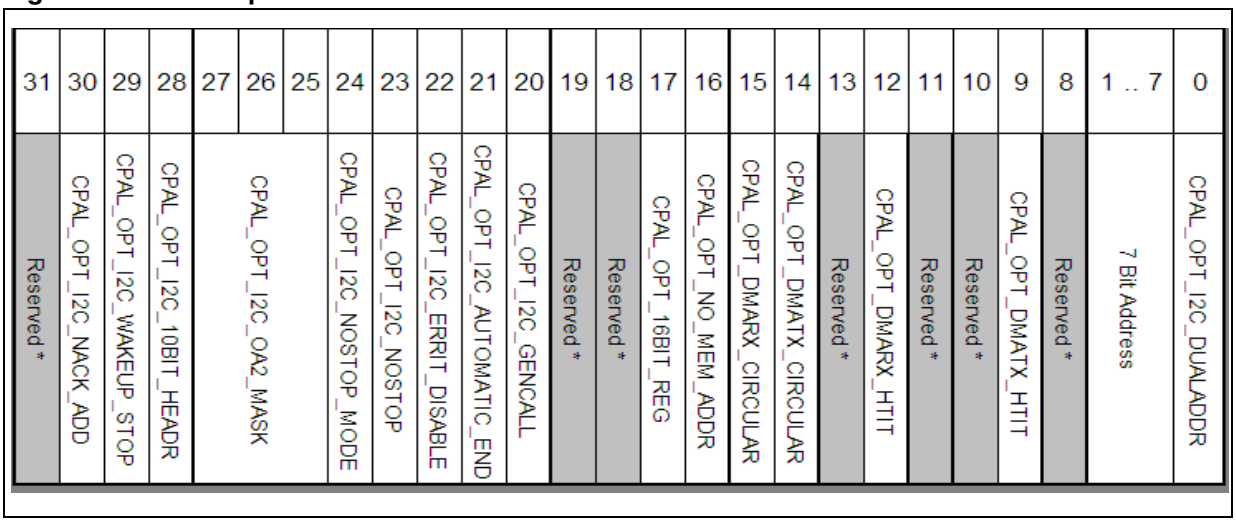
Table 10. wCPAL_DevError field values (for I²C peripherals) (continued)

Field value	Description
CPAL_I2C_ERR_ARLO	An arbitration loss is detected when a high level is sent on SDA, but a low level is sampled on the SCL rising edge. – In master mode, the arbitration loss is detected during address phase, data phase and data acknowledge phase. In that case, SDA and SCL lines are released, START control bit is cleared by hardware and the master switches automatically to slave mode. – In slave mode, arbitration loss is detected during data phase and data acknowledge phase. In that case, the transfer is stopped, and SCL and SDA lines are released. When an arbitration loss is detected, ARLO flag is set in I2C_ISR register, and an interrupt is generated if ERRIE is set in I2C_CR1 register. It is cleared by software by setting ARLOCF bit.
CPAL_I2C_ERR_AF	Not Acknowledge is detected as an error only in master mode when a NACK is received after sending slave address. NACKF flag is set in I2C_ISR register, and an interrupt is generated if NACKIE is set in I2C_CR1 register. It is cleared by software by setting NACKCF bit.
CPAL_I2C_ERR_OVR	An overrun or underrun error is detected in slave mode when NOSTRETCH=1 and: – In reception when a new byte is received and the RXDR register has not been read yet. New received byte is lost, and a NACK is automatically sent as a response to the new byte. – In transmission: – when STOPF=1 and the first data should be sent. The content of TXDATA is sent if TXE=0, 0xFF if not. – when a new byte should be sent and the TXDR register has not been written yet. 0xFF is sent. When an overrun or underrun error is detected, OVR flag is set in I2C_ISR register, and an interrupt is generated if ERRIE is set in I2C_CR1 register. It is cleared by software by setting OVRCF bit.

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.

Figure 3. CPAL option fields



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 11. wCPAL_Options field values

Field value	Description
CPAL_OPT_I2C_DUALADDR	Enable the I^2C Dual Addressing mode for the related peripheral(1).
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_DUALADDR` option is enabled.
CPAL_OPT_DMATX_HTIT	Enable the DMA Tx Half Transfer Complete interrupt for the related peripheral.
CPAL_OPT_DMARX_HTIT	Enable the DMA Rx Half Transfer Complete interrupt for the related peripheral.
CPAL_OPT_DMATX_CIRCULAR	Enable the Circular mode for the Tx DMA transfers on the related peripheral.
CPAL_OPT_DMARX_CIRCULAR	Enable the Circular mode for the Rx DMA transfers on the related peripheral.
CPAL_OPT_NO_MEM_ADDR	Enable No Memory addressing mode for the peripheral-related I^2C. This means that the master sends only the peripheral slave address (no physical addresses into the slave). (2)
CPAL_OPT_16BIT_REG	Enable 16-bit register addressing mode. Thus the register/physical address (sent after the peripheral slave address) is considered as a 2-byte address.(3)
CPAL_OPT_I2C_GENCALL	Enable the I^2C General Call mode for the related peripheral.
CPAL_OPT_I2C_AUTOMATIC_END	Enable Automatic end mode of STOP generation for master.
CPAL_OPT_I2C_ERRIT_DISABLE	Disable the I^2C error interrupt (Bus error, Arbitration loss, Acknowledge failure and Overrun/Underrun errors). By default the error interrupts are enabled.

Table 11. wCPAL_Options field values (continued)

Field value	Description
CPAL_OPT_I2C_NOSTOP	Disable the sending of STOP condition at the end of the current buffer transfer for the relative I^2C device. This option may be enabled when multiple packets have to be sent consecutively without STOP generation. This option can be used only if software end mode is selected (CPAL_OPT_I2C_AUTOMATIC_END option is disabled).
CPAL_OPT_I2C_NOSTOP_MODE	Start communication in No STOP generation mode and close communication by Generating stop
CPAL_OPT_I2C_OA2_MASK	Contain Own Address 2 Mask (OA2MSK is coded on 3 bit)
CPAL_OPT_I2C_10BIT_HEADR	Enable the send of slave address-10bit-header only when switching from master transmitter to master receiver mode with No stop generation option enabled
CPAL_OPT_I2C_WAKEUP_STOP	Enable the WakeUp from stop capability for the I^2C slave device
CPAL_OPT_I2C_NACK_ADD	Enable the Initialization of the I^2C 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 assign 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 I^2C peripherals in Master mode. The physical address is an address into the slave peripheral into/from which the write/read operation is performed (that is, 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.

2.2.2 CPAL communication functions (stm32xxxx_i2c_cpal.c)

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

Table 12. 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 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 13. 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 `pDevInitStructure`.
`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). *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 I^2C and one other peripheral, then make sure to re-enable this DMA channel and its interrupt after calling CPAL_I2C_DeInit function).*
`CPAL_I2C_StructInit()`	This function initializes the related peripheral structure (pointed by `pDevInitStruct`) by filling all fields with their default values. **Caution:** 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 *Table 8*. 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.(1)
`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 *Table 8*. 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.(2)
`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()`	This function can be used to: – Wait until the target peripheral is ready for communication (i.e., for memories). – Verify that the external slave peripheral is connected to the bus (using its address). 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. 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.

1. When `CPAL_I2C_Read()` function is called, the user application may perform other tasks while the transfer is ongoing and the `CPAL_State` field can be used to monitor transfer.

2. When `CPAL_I2C_Write()` function is called, the user application may perform other tasks while the transfer is ongoing and the `CPAL_State` field can be used to monitor transfer.

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 (`stm32xxxx_i2c_cpal_conf.h` and `stm32xxxx_i2c_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 I^2C 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 `stm32xxxx_i2c_cpal_conf.h` file.

2.3.1 Configuration interface

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 (`stm32xxxx_i2c_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.

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 14. CPAL configuration sections

Section	Options	Description
Section 1: Peripheral selection	CPAL_USE_I2CX	- X = device instance: 1 or 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 I^2C 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 I^2C 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 I^2C 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.
	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_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 handle10-bit addressing mode. When this option is enabled, it does not mean that all I^2C peripherals communicate in 10-bit addressing mode: to select this mode for an I^2C 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.
Section 3: User callback configuration	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 stm32xxxx_i2c_cpal_conf.h file, then implement the callback body into stm32xxxx_i2c_cpal_usercallback.c file (callback prototype is already declared in CPAL library). For more details about callbacks, refer to *Section 2.3.2* **Caution:** 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 14. CPAL configuration sections (continued)

Section	Options	Description
Section 4: Timeout configuration	`CPAL_TIMEOUT_INIT()`	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). The timeout counter functions as follows: – The counter generates fixed-period ticks and calls `CPAL_I2C_TIMEOUT_Manager()` callback at each tick. – The `CPAL_I2C_TIMEOUT_Manager()` checks the value of `wCPAL_Timeout` of all the available I^2C peripheral structures: – If `wCPAL_Timeout` = `CPAL_I2C_TIMEOUT_DEFAULT` then no action is performed. – 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. – If `wCPAL_Timeout` has any other value, the function decrements its value by 1 and exit. – The User may implement his own timeout mechanism (i.e., using SysTick timer or other timers). The counting unit should preferably be set to 1 millisecond (ms). This function should configure the counting unit and enable the counting start. Other timeout initialization procedures may be implemented depending on application needs.
	`CPAL_TIMEOUT_DEINIT()`	This macro is used to deinitialize the countdown mechanism. It is called whenever a peripheral is deinitialized (when calling `CPAL_I2C_DeInit()` function). Other timeout initialization procedures may be implemented depending on application needs.This function may be performed for each peripheral separately.

Table 14. CPAL configuration sections (continued)

Section	Options	Description

Section 4: Timeout configuration (continued)	CPAL_I2C_TIMEOUT_Manager	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.
		In case of multiple peripheral types managed by the same interrupt handler, an intermediate function may be implemented and called into the interrupt handler.
		Example:
		In stm32xxxx_i2c_cpal_conf.h:
		`#define CPAL_I2C_TIMEOUT_Manager UserFunction1`
		In stm32fxxx_it.c file:
		`void SysTick_Handler(void)`
		`{ …..`
		`UserFunction1();`
		`….`
		`}`
		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.

	CPAL_I2C_TIMEOUT_MIN	The minimum timeout value for the peripheral timeout counter when enabled (this value is applied to the device structure timeout field wCPAL_Timeout).
	CPAL_I2C_TIMEOUT_DEFAULT	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.
	CPAL_I2C_TIMEOUT_WWW	Where WWW can be replaced by the peripheral event (i.e., SB, ADDR …) 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). The user may specify different timeout periods for each event in order to meet the requirements and constraints of the application.

Table 14. CPAL configuration sections (continued)

Section	Options	Description
Section 5: Interrupt priority selection	CPAL_NVIC_PRIOGROUP	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	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	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

The callback interface (`stm32xxxx_i2c_cpal_usercallback.c`) allows the implementation of user callbacks when needed. A template file is provided in the library (`stm32xxxx_i2c_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 `stm32xxxx_i2c_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 15. CPAL configuration sections

Callback	Description
Transfer callbacks	
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 I^2C 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.
CPAL_I2C_SLAVE_ZZ_UserCallback	– Where ZZ is the transfer direction: Read or Write. These functions are called when the slave receives its own address in listen mode.
Error Callbacks	
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 I^2C peripheral. All peripherals of the same type (i.e., all I^2C peripherals) share the same error callback. The user has to check which peripheral caused the entering in this callback.

Table 15. CPAL configuration sections (continued)

Callback	Description
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 I²C peripherals) share the same error callback. The user has to check which peripheral caused the entry into this callback.
Address mode callbacks	
CPAL_I2C_GENCALL_UserCallback	This callback is valid only when CPAL_OPT_I2C_GENCALL is enabled (see *Section 3: CPAL functional description*) and when I²C is configured in Slave mode. This function is called when a General call event occurs on an I²C peripheral.
CPAL_I2C_DUALF_UserCallback	This callback is valid only when CPAL_OPT_I2C_DUALADDR is enabled (see *Section 3: CPAL functional description*). 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.
Listen Mode callbacks	
CPAL_I2C_SLAVE_XXXX_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.

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 the family identifier, for example stm32f0xx):

- `stm32xxxx_i2c_cpal_hal.c`
- `stm32xxxx_i2c_cpal_hal.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 `stm32xxxx_i2c_cpal_hal.h` configuration sections.

The supported configurations are listed in the following table:

Table 16. HAL configuration sections

Section	Options	Description

Section 1: Peripheral pin selection	CPAL_I2CX_YYY_GPIO_PORT	– Where X is the peripheral instance (1, 2 or 3...), YYY is the pin function (i.e. SCL, SDA for I2C peripheral).
		For each parameter, set the define value to use the related configuration, i.e:
		`#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_SCL_GPIO_PINSOURCE GPIO_PinSource6`
		For each I/O, one single configuration is allowed. A table in the stm32xxxx_i2c_cpal.h file shows the possible configurations for each I/O and each peripheral.

	CPAL_I2CX_YYY_GPIO_CLK	
	CPAL_I2CX_YYY_GPIO_PIN	
	CPAL_I2CX_YYY_GPIO_PINSOURCE	
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 stm32xxxx_i2c_cpal.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 stm32xxxx_i2c_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 17. 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.
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 mode) 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	ProgModel	

Table 17. 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 I^2C peripheral in the current configuration and according to the enabled options (interrupt/DMA mode). 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	ProgModel	

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 `stm32xxxx_i2c_cpal.h` file for the I^2C 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.

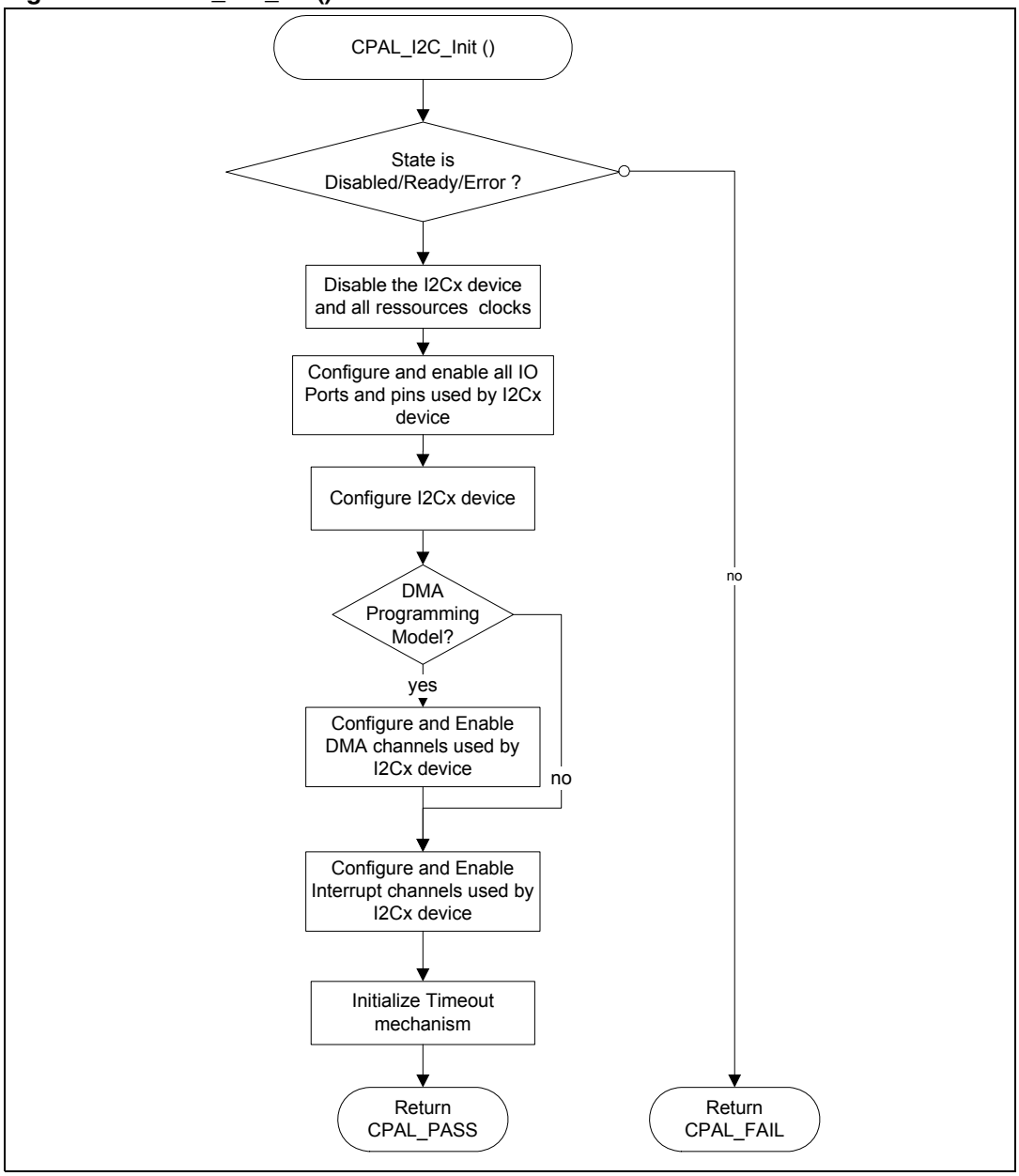
The `CPAL_I2C_Init()` function performs the following actions:

- 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 4. CPAL_I2C_Init() function flowchart



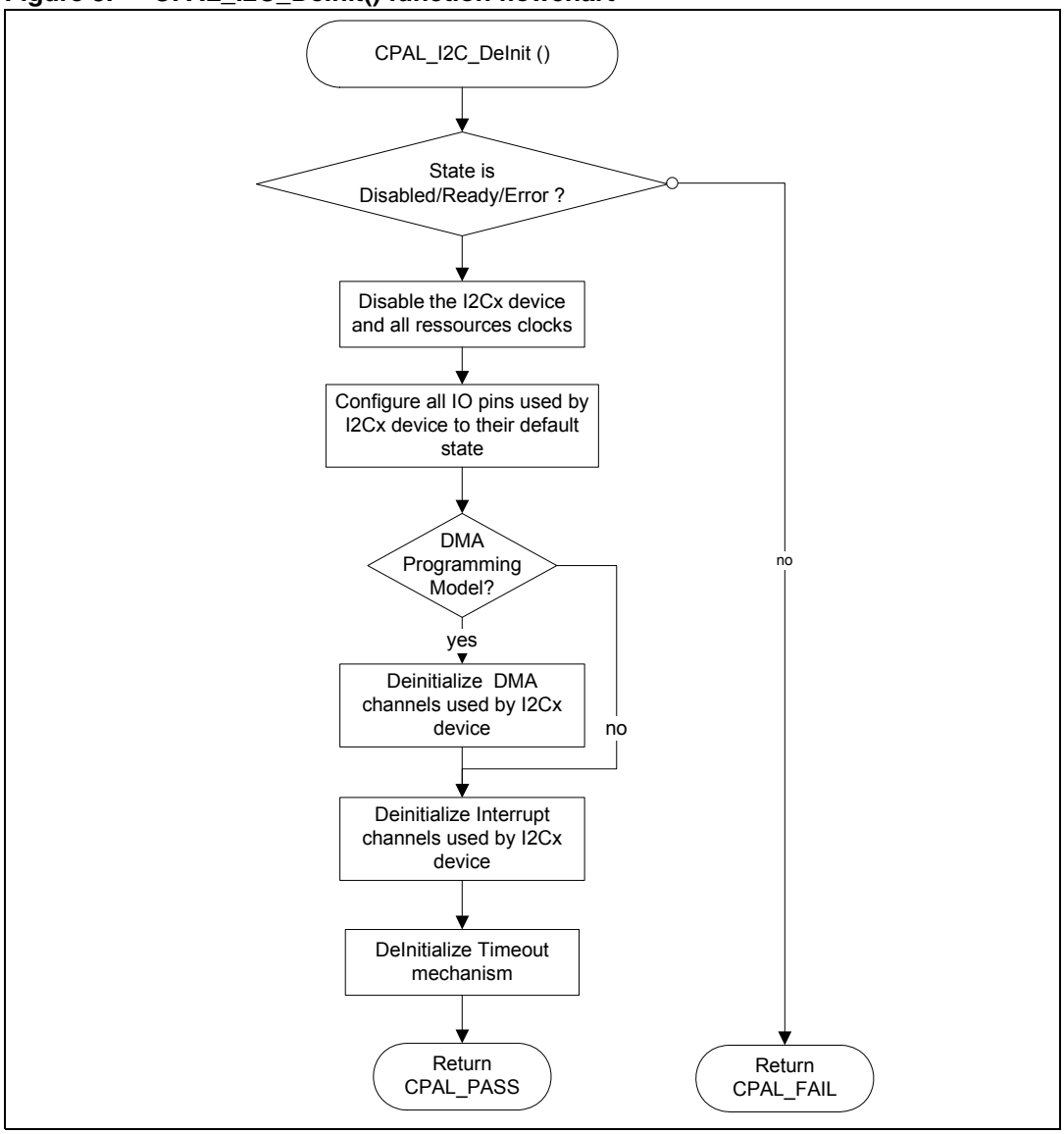
3.1.2 CPAL_I2C_DeInit() functional description

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 5. 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 18. 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_Timing`	0
	`I2C_Mode`	I2C_Mode_I2C
	`I2C_AnalogFilter`	I2C_AnalogFilter_Enable
	`I2C_DigitalFilter`	0
	`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: CPAL_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.

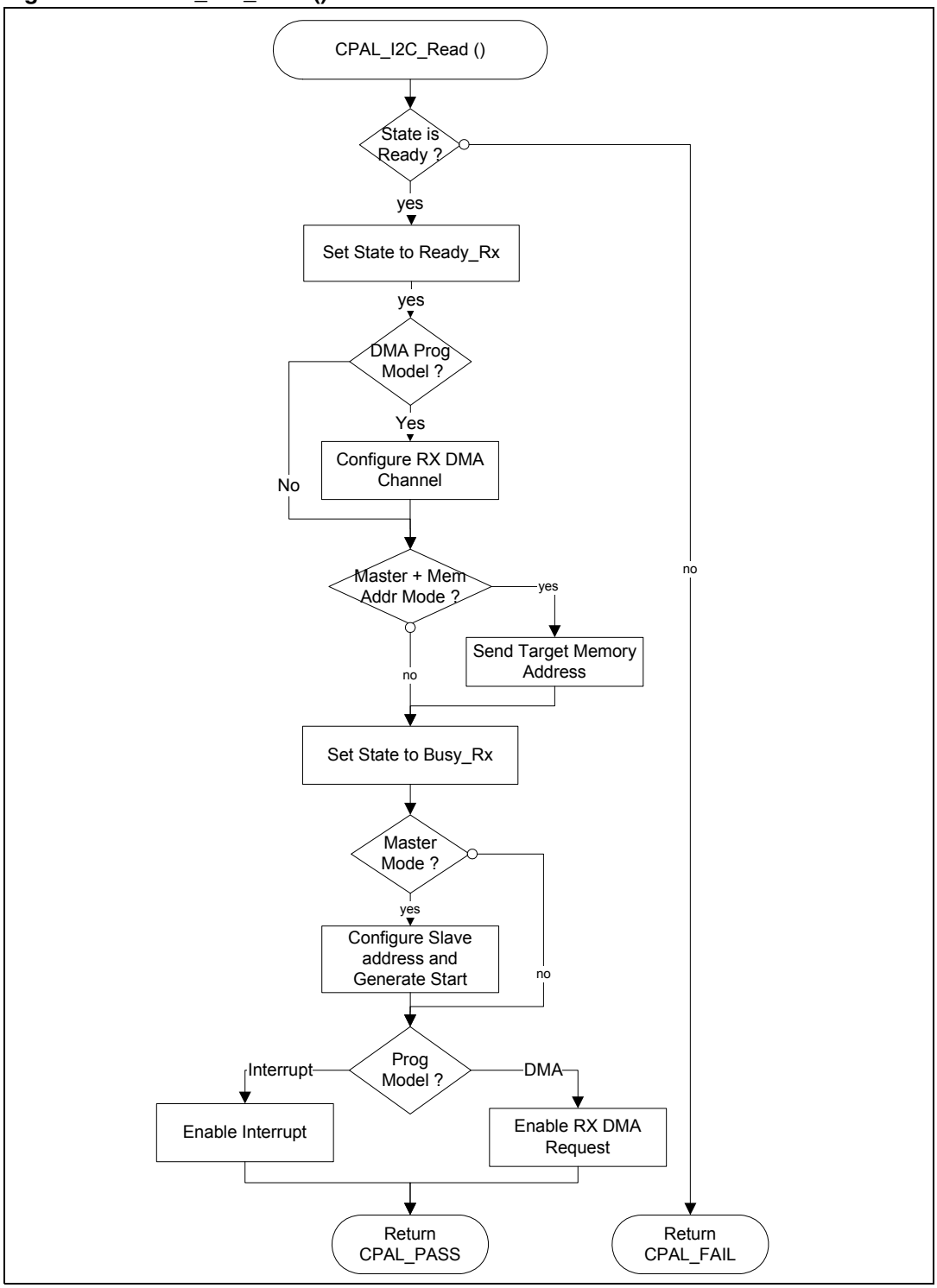
The `CPAL_I2C_Read()`, `CPAL_I2C_Write()` and `CPAL_I2C_Listen()` functions just prepare and configure the communication.

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.

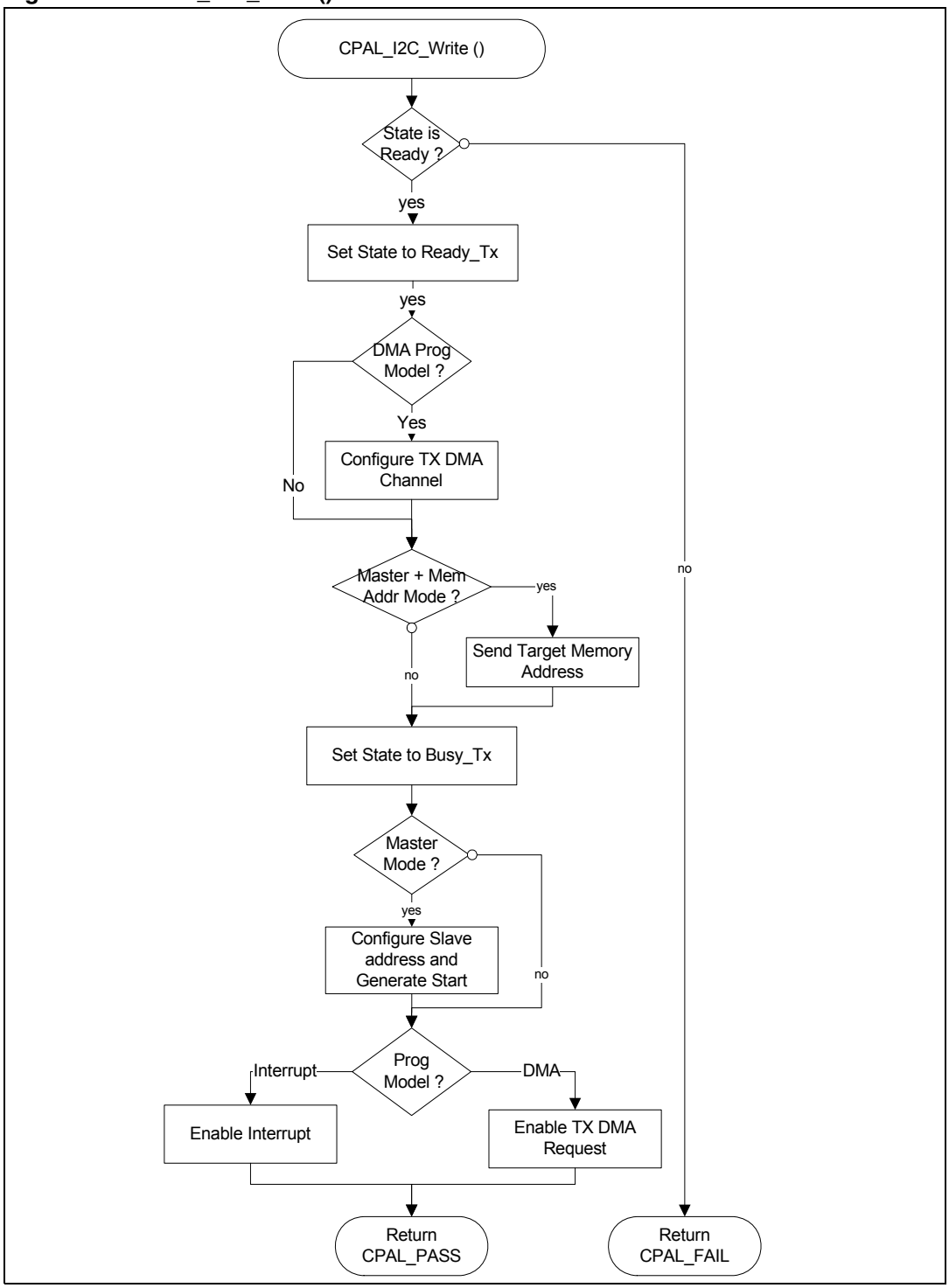
Figure 6. CPAL_I2C_Read() function flowchart



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 7. CPAL_I2C_Write() function flowchart



3.2.3 CPAL_I2C_Listen () functional description:

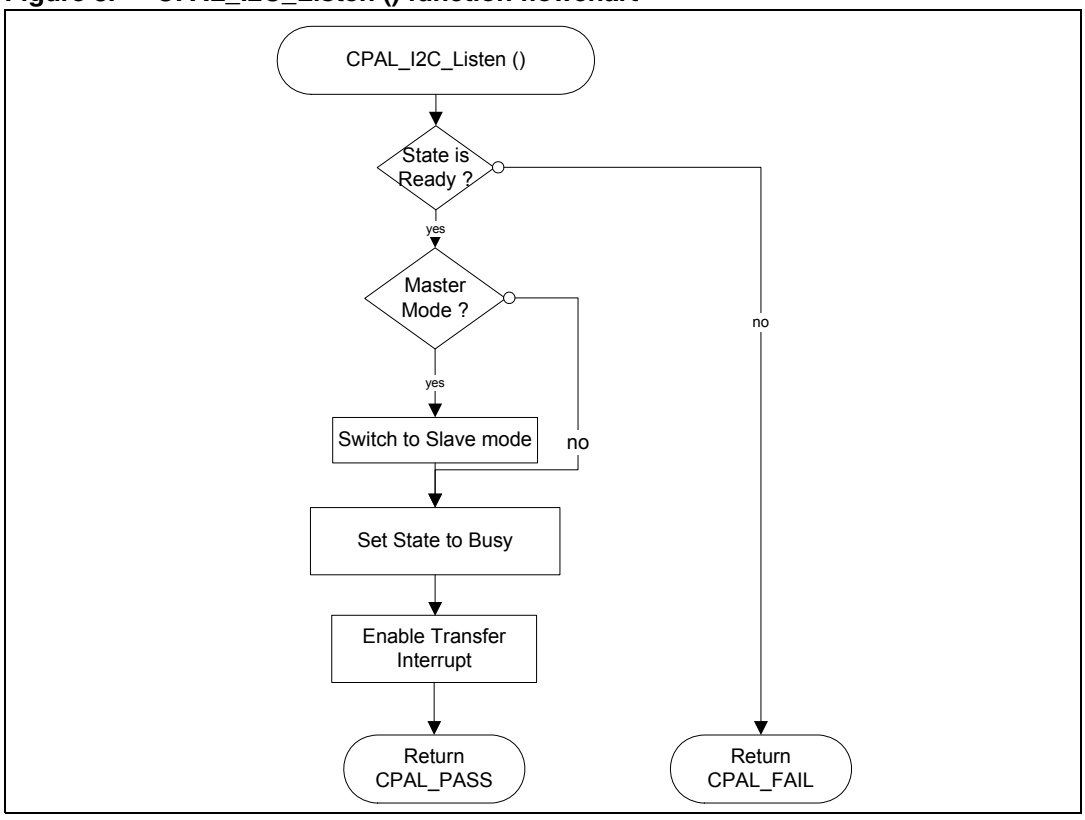
The `CPAL_I2C_Listen ()` function is only used with slave device when Listen mode is activated. If this function is called in master mode the device is forced to slave mode.

`CPAL_I2C_Listen()` function replace `CPAL_I2C_Read()` and `CPAL_I2C_Write()` functions for Slave mode. 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 8. 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_TransferTx` 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 mode (Master, Slave), the programming model (Interrupt, DMA) and the option configuration (No memory addressing mode, General call mode, …).

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. Transfer interrupt will be enabled only when `CPAL_PROGMODEL_INTERRUPT` mode is selected.

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

Table 19. I2C interrupt management order

Interrupt	Details	Callback
TXIS	Manages the event “Transmit Interrupt Status” which means a new data shall be written in the I^2C data register for the next transfer.	`CPAL_I2C_TX_UserCallback`
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`
TCR	Available in Master mode only. Manages Transfer Complete Reload event which means the master or slave received or transmitted nbytes and Reload =1.	`NA`
TC	Available in Master mode only. Manages Transfer Complete event which means the master received or transmitted all data and communication will be closed (Generate stop) or another one will start (Repeated start).	`CPAL_I2C_TXTC_UserCallback` `CPAL_I2C_RXTC_UserCallback`

Table 19. I2C interrupt management order

Interrupt	Details	Callback
ADDR	Manages the event “Address phase done” which means that the device in mode slave received start bit followed by its own address and acknowledged it.	CPAL_I2C_GENCALL_UserCallback CPAL_I2C_DUALF_UserCallback CPAL_I2C_SLAVE_READ_UserCallback CPAL_I2C_SLAVE_WRITE_UserCallback
NACK	Manages the event “Not Acknowledge received flag” which means the device received a NACK. In master mode this flag indicates an error (slave don’t respond to sent address). In slave mode, when master received all data it send NACK to indicate to slave that all data are received.	CPAL_I2C_AF_UserCallback CPAL_I2C_ERR_UserCallback CPAL_I2C_TXTC_UserCallback
STOP	Manages the event “Stop bit received” which means that the master has closed the communication.	CPAL_I2C_TXTC_UserCallback CPAL_I2C_RXTC_UserCallback

For I^2C 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 20. 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* 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 `stm32xxxx_i2c_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 21. CPAL I2C user callback list

Callback name	Description
Communication User Callbacks 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.
`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`	

Table 21. CPAL I2C user callback list

Callback name	Description
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 I^2C 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_HAL_DMATXConfig( ), __CPAL_I2C_HAL_ENABLE_DMATX() and __CPAL_I2C_HAL_ENABLE_TXDMAREQ() in DMA mode. In interrupt mode only __CPAL_I2C_HAL_ENABLE_SLAVE_TXIT() should be called. In these cases these functions should be called 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_HAL_DMARXConfig(), __CPAL_I2C_HAL_ENABLE_DMARX() and __CPAL_I2C_HAL_ENABLE_RXDMAREQ() in DMA mode. In interrupt mode only __CPAL_I2C_HAL_ENABLE_SLAVE_RXIT() should be called. In these cases these functions should be called after configuring Transfer parameters (pDevInitStruct).
Error User Callbacks 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.	
CPAL_I2C_BERR_UserCallback	Multiple Error callbacks are available only when the define USE_MULTIPLE_ERROR_CALLBACK is enabled in the file stm32xxxx_i2c_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	

Table 21. CPAL I2C user callback list

Callback name	Description
CPAL_I2C_ERR_UserCallback	Single Error callback is available only when the define USE_SINGLE_ERROR_CALLBACK is enabled in the stm32xxxx_i2c_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).
Timeout User Callbacks 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.

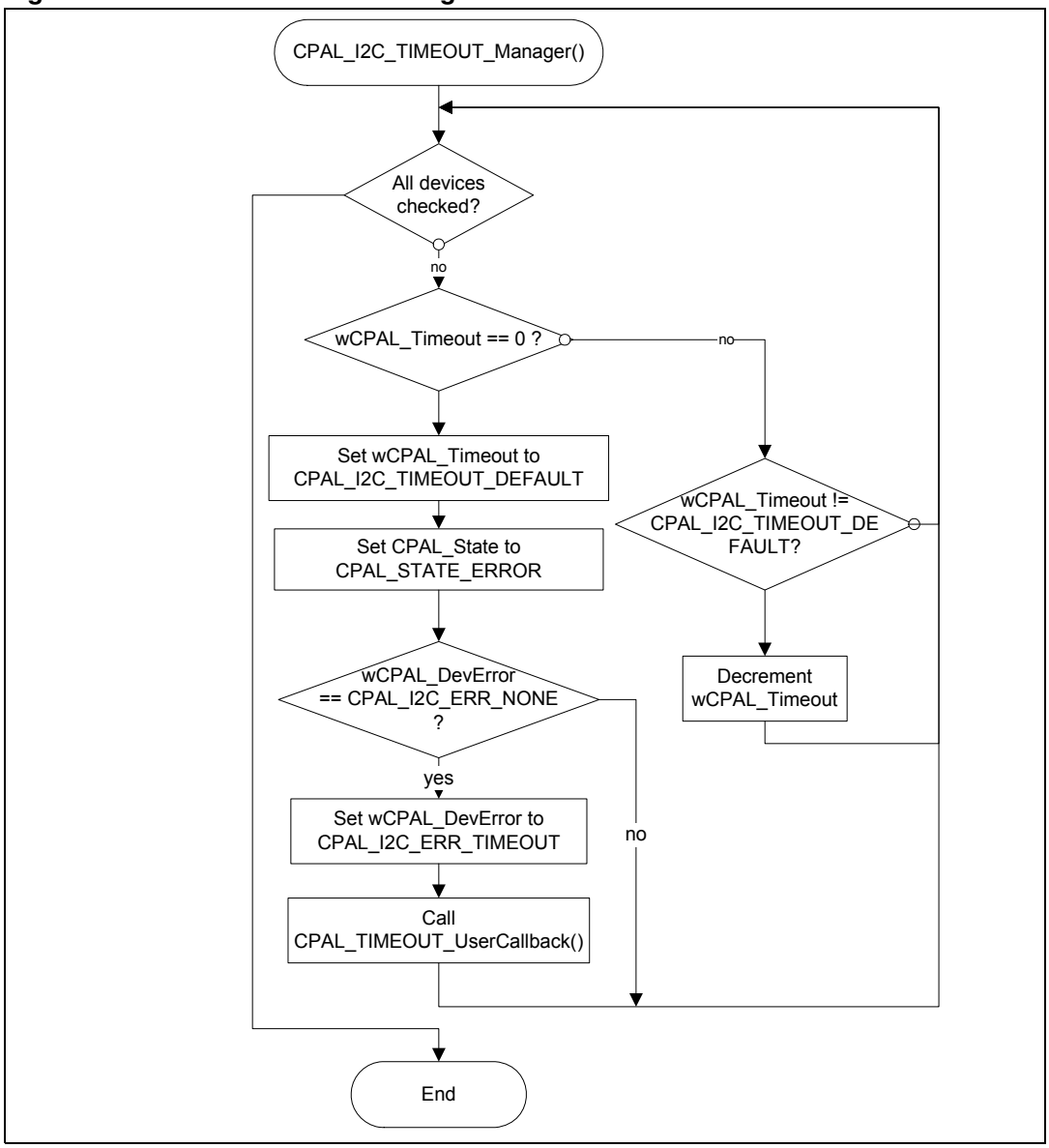
The timeout mechanism is based on three elements:

- 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 I^{2}C 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 `stm32xxxx_i2c_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 9. CPAL I2C timeout manager flowchart



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 `stm32xxxx_i2c_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 `stm32xxxx_i2c_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 I^2C 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 `stm32xxxx_i2c_cpal_conf.h` file.

The `stm32xxxx_i2c_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 `stm32xxxx_i2c_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 `stm32xxxx_i2c_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_BUSY 5`).

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_TC          5
#define CPAL_I2C_TIMEOUT_TCR         5
#define CPAL_I2C_TIMEOUT_TXIS        5
#define CPAL_I2C_TIMEOUT_BUSY        5
```

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 `stm32xxxx_i2c_cpal.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 18: 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 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: CPAL_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 peripheral is communicating or an error occurred. In both cases, it is not possible to use the 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 I^2C 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 restart 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 stm32xxxx_i2c_cpal_hal.h file (which is specific for each device family).

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

stm32xxxx_i2c_cpal_hal.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 I^2C 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_SCL_GPIO_PINSOURCE    GPIO_PinSource6

#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
#define CPAL_I2C1_SDA_GPIO_PINSOURCE    GPIO_PinSource7
```

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 stm32xxxx_i2c_cpal_conf.h file (I2CX_IT_OFFSET_SUBPRIO and I2CX_IT_OFFSET_PREPRIO).

Generally the following interrupt priority order should be applied:

For I^2C peripherals: error interrupts should have the highest priority level, then DMA interrupts (allowing the application to close communication) and finally the I^2C 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 I^{2}Cs (I2C1 and I2C2) to control an EEPROM memory and a temperature sensor (each on separate I^{2}C 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 one of the example 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:

- `stm32xxxx_i2c_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)
- `stm32xxxx_i2c_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 stm32xxxx_i2c_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 two I^{2}Cs. 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.

Both I^{2}Cs 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_INIT()      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 I^2C 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:

```
#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__ */

/**
  * @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 stm32xxxx_i2c_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 stm32xxxx_i2c_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 I^2C interface and then control it to communicate with the EEPROM and Temperature Sensor.

5.4.1 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;
```

5.4.2 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 I^{2}C structures should be initialized (note that for each I^{2}C 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_Timing = 0xC062121F;
  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_Timing = 0xC062121F; /* 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);
```

5.4.3 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 CPAL firmware library, the CPAL package provides a set of examples for I^2C peripheral, aiming at providing different levels of implementation complexity.

This release comes with three examples running on STM320518-EVAL (STM32F0xx), STM32373C-EVAL (STM32F37x) and STM32303C-EVAL (STM32F30x) 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.

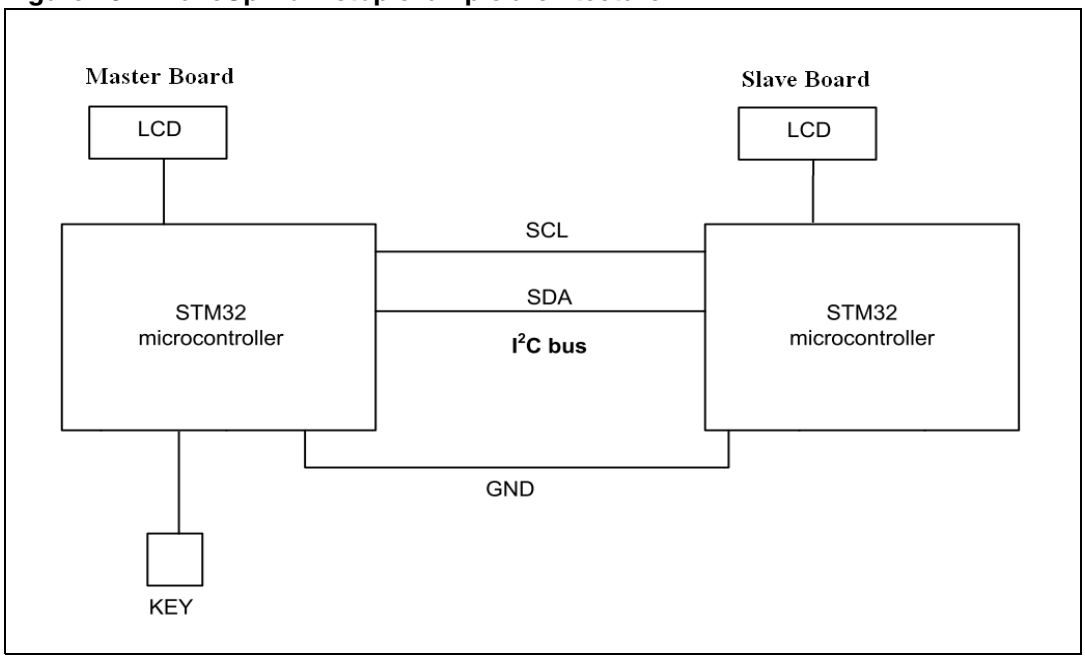
Table 22. Hardware resources used in CPAL examples

Example	Board	Used Resources		
		LCD	Buttons	LED
WakeUp from Stop mode	STM320518_EVAL	X	X	X
	STM32373C_EVAL			
	STM32303C_EVAL			
Two Boards	STM320518_EVAL	X	X	X
	STM32373C_EVAL			
	STM32303C_EVAL			
Two Boards Listen mode	STM320518_EVAL	X	X	X
	STM32373C_EVAL			
	STM32303C_EVAL			

6.1 Wakeup from Stop mode example

This example shows how to use the Wakeup from Stop feature implemented in I^2C peripheral using the CPAL drivers.

Figure 10. WakeUp from stop example architecture



One of the STM32 devices is configured as slave and the other one as master. At the example startup, the slave enters in Stop mode and wait until it detects its own address on the I^2C bus line. User should push button key of the master board to send slave address and data. When the slave board detects its own address, it wakes up from Stop, shows a message on LCD display, then returns to Stop mode.

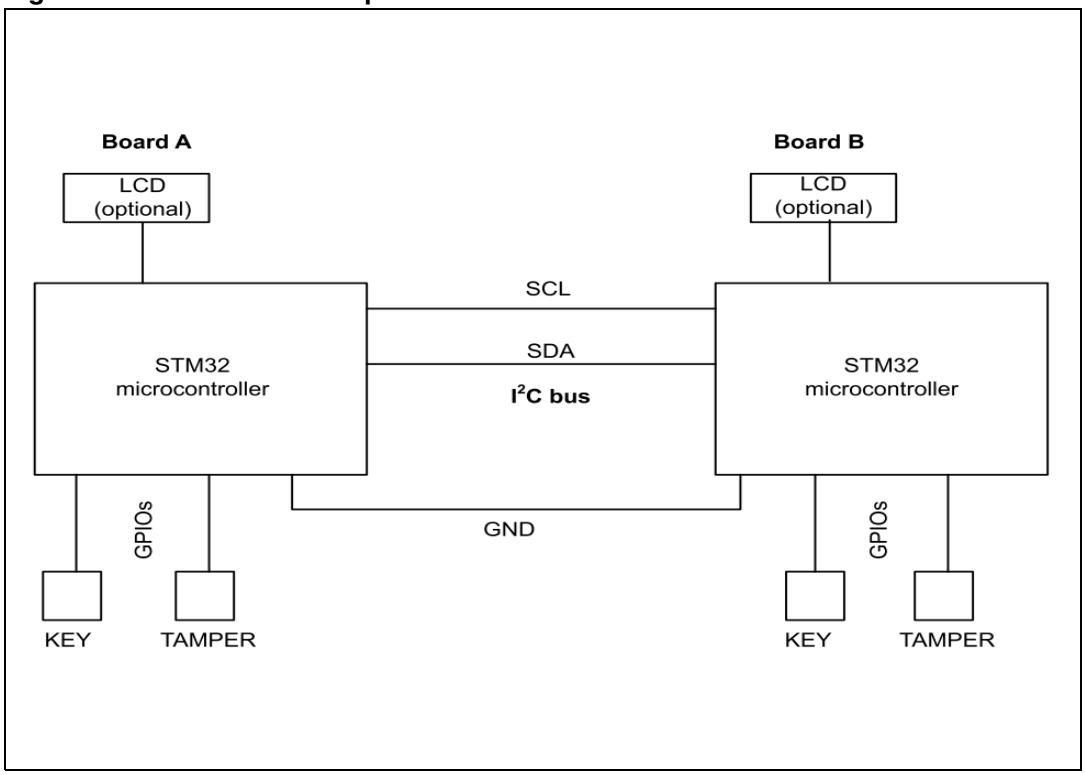
For more details about the hardware requirements and the supported devices and boards, refer to the readme file located in the example directory.

6.2 Two-board example

This example shows how two microcontrollers may communicate on the same I^2C bus using the CPAL drivers.

This example shows how the I^2C 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 11. 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 Two-board Listen mode example

This example shows how two microcontrollers can communicate on the same I^2C bus using the CPAL drivers and how a slave device communicates in Listen mode. Also it shows how the I^2C 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 11* 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 6.40 tool with High Size optimization level, and are valid for STM32F37x 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 *stm32xxxx_i2c_cpal_conf.h* file (that should be extracted from *stm32xxxx_i2c_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 I^2C 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 greyed.*

5 *Except for addressing mode options (CPAL_16BIT_REG_OPTION and CPAL_I2C_10BIT_ADDR_MODE): 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 23. Memory footprint of CPAL components

Options		Define	Code Size (Bytes)	
			Flash	RAM
CPAL Core (All options disabled)	1 Device	CPAL_USE_I2C1 All other options disabled	1692	96
	2 Devices	CPAL_USE_I2C1 CPAL_USE_I2C2 All other options disabled	1748	128
Mode Option(1)	Master	CPAL_I2C_MASTER_MODE	1408	4
	Slave without Listen mode	CPAL_I2C_SLAVE_MODE	496	0
	Slave with Listen mode	CPAL_I2C_SLAVE_MODE CPAL_I2C_LISTEN_MODE	88	0
Programming Model Option (1)	DMA	CPAL_I2C_DMA_PROGMODEL	1005	68
	Interrupt	CPAL_I2C_IT_PROGMODEL	332	0

Table 23. Memory footprint of CPAL components (continued)

Options		Define	Code Size (Bytes)	
			Flash	RAM
Addressing Mode Option[1]	Memory Address	CPAL_16BIT_REG_OPTION	608	0
	16 Bit	CPAL_16BIT_REG_OPTION	344	0
	10 Bit	CPAL_I2C_10BIT_ADDR_MODE	76	0
All Options enabled (Full CPAL set)	1 Device	CPAL_USE_I2C1 All other options enabled	4806	168
	2 Devices (with listen mode)	CPAL_USE_I2C1 CPAL_USE_I2C1 All other options enabled	4779	200
	2 Devices	CPAL_USE_I2C1 CPAL_USE_I2C1 All other options enabled	4691	200
Master Mode disabled, All other options enabled[2]		CPAL_I2C_MASTER_MODE disabled	3283	196
Slave Mode disabled, All other options enabled[2]		CPAL_I2C_SLAVE_MODE disabled	4195	200
Slave in Listen Mode disabled, All other options enabled[2]		CPAL_I2C_LISTEN_MODE disabled	4691	200
DMA Mode disabled, All other options enabled[2]		CPAL_I2C_DMA_PROGMODEL disabled	3686	132
Interrupt Mode disabled, All other options enabled[2]		CPAL_I2C_IT_PROGMODEL disabled	4359	200
Memory Address mode disabled, All other options enabled[2]		CPAL_I2C_MEM_ADDR disabled	4083	200
16 Bit Addressing mode disabled, All other options enabled[2]		CPAL_16BIT_REG_OPTION disabled	4347	200
10 Bit Addressing mode disabled, All other options enabled[2]		CPAL_I2C_10BIT_ADDR_MODE disabled	4615	200

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

8 Frequently asked questions (FAQ)

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

Table 24. Frequently asked questions

No.	Question	Possible answers / solutions
Topic 1: General		
1	Why would I use the CPAL driver rather than a standard peripheral library?	The main advantage of using the CPAL library is the ease of use: you can use CPAL driver to control the I^2C interface without any knowledge about I^2C protocol. CPAL library also offers a higher level of abstraction allowing the “transparent” management of: - Hardware components used by communication peripherals (I/Os, DMA, interrupts …) - Transfer buffers and status (managed through independent structures). - Peripheral states (i.e. event management for I^2C peripheral …) - Error detection and recovery when possible (through peripheral error detection and timeout mechanism). - Bug fixes and workaround selection. - Different device families.
2	What is the cost of using CPAL drivers in term of code size and performance?	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. 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.
3	How many files should I modify to configure the CPAL drivers?	CPAL library offers multiple levels of customization: - **0 file**: No file needs to be modified: you can use the 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. - **1 file**: In most cases, only one file needs to be modified: `stm32xxxx_i2c_cpal_conf.h`. You can modify this file by disabling unused features, or adjusting some parameters (i.e. interrupt priority groups, timeout mechanism …). - **2 files**: In addition to the `stm32xxxx_i2c_cpal_conf.h` file, you can customize the CPAL library hardware layer by modifying the file `stm32f30x_i2c_cpal_hal.h`. Through this file you can modify the I/O selection, the DMA channels, the interrupt priorities…

Table 24. Frequently asked questions (continued)

No.	Question	Possible answers / solutions
4	Which header files should I include in my application in order to use the CPAL library?	Only `stm32xxxx_i2c_cpal.h` file has to be included.
Topic 2: Configuration		
5	How many fields are mandatory to fill for the CPAL initialization structure?	The easiest way is to call the function `CPAL_I2C_StructInit()` to initialize the structure with default values. Then you have to set the following fields: `pCPAL_TransferTx` and/or `pCPAL_TransferRx` 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. For all other fields, you can keep default values in most cases (check *Section 2.3.2*) for more details on default values.
6	I use more than one I^2C peripherals and they are not configured correctly after I call the `CPAL_I2C_StructInit()` `CPAL_I2C_Init()` functions.	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 `CPAL_StructInit()` functions, all fields are set to default values, but pointers are set as follows: – Transfer structure pointers are set to Null value. – I^2C initialization structure pointer (`pCPAL_I2C_Struct`) is set to a local structure containing default values for the I^2C. So this structure will be used by all I^2C peripherals at the same time. Any modification on it will affect all I^2C peripherals. To make sure that each peripheral is correctly configured, declare a local structure for each one and assign it to the field `pCPAL_I2C_Struct` after calling `CPAL_I2C_StructInit()` function.
Topic 3: Interrupts		
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.	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 `stm32xxxx_i2c_cpal_conf.h` *Section 5: Interrupt priority selection* (*`CPAL_NVIC_PRIOGROUP`*) and remove any other settings of this parameter in your application. 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 `stm32xxxx_i2c_cpal_conf.h` file *Section 5: Interrupt priority selection* (i.e., *`I2CX_IT_OFFSET_SUBPRIO`*)

Table 24. Frequently asked questions (continued)

No.	Question	Possible answers / solutions
8	I cannot find I²C interrupt handler in the `stm32xxxx_it.c` 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: `I2Cx_EV_IRQHandler` and `I2Cx_ER_IRQHandler`). There is no need for additional configuration. You just have to use the CPAL callbacks provided for this peripheral in `cpal_usercallbacks.c` file and comment the related callback define in `stm32xxxx_i2c_cpal_conf.h` file section 3.
9	When I implement CPAL drivers into my application, I have warnings/errors related to I²C interrupt	
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 `stm32xxxx_i2c_cpal_conf.h` file, *`CPAL_I2C_DMA_PROGMODEL`*. 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 `stm32xxxx_i2c_cpal_usercallback.c` file (`CPAL_I2C_DMATXTC_UserCallback` ...) If these handlers are needed for other purposes, then try to change the configuration of the DMA channels in `cpal_ppp_stm32xxxx.h` file configuration section 2 in order to free the requested channels.
11	When I implement CPAL drivers into my application, I have warnings/errors related to DMA interrupt handlers.	
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.
Topic 3: Hardware		
13	Which STM32F0/F3 series devices are supported by the CPAL Library?	Refer to the Release Notes of the CPAL package for the complete list of supported STM32F0/F3 series devices.

9 Naming conventions

The communication peripheral access library (CPAL) uses the following device naming conventions:

- STM32F37x is used to refer to STM32F37x and STM32F38x devices.
- STM32F30x is used to refer to STM32F30x and STM32F31x devices.
- STM32F3 is used to refer to STM32F37x and STM32F30x.
- STM32xxxx is used to refer to STM32F0xx, STM32F37x and STM32F30x devices.

10 Revision history

Table 25. Document revision history

Date	Revision	Changes
25-Oct-2012	1	Initial release.