emUSB-C PD

USB-C and Power Delivery
stack for embedded applications

User Guide & Reference Manual

Document: UM11001
Software Version: 1.20.0
Revision: 1
Date: December 22, 2023

A product of SEGGER Microcontroller GmbH

www.segger.com

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	`ticket_emusb@segger.com`[*]
Internet:	*www.segger.com*

[*]By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: December 22, 2023

Software	Date	By	Description
1.20.0	2023-12-13	RH	Added configuration item *DeviceDataRoleDelay*. Add new function `USBC_EnableTrimming_STM32Uxx()`.
1.10.0	2023-11-21	RH	Minor corrections.
1.00.0	2023-10-25	RH	Initial version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (compiler, linker, Integrated Development Environment).
- The C programming language.
- The target processor.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
Emphasis	Very important sections.

Table of contents

Chapter 1

Introduction

This chapter provides an introduction to using emUSB-C. It explains the basic concepts behind emUSB-C.

1.1 What is emUSB-C

USB-C is an industry-standard connector for transmitting both data and power on a single cable. emUSB-C is a library that enables applications to easily handle a USB Type-C port on an embedded device. It does not support USB communication but most functionality added to USB by the USB Type-C connector, like host / device connection detection and power delivery.

emUSB-C can be used in combination with emUSB-Host and/or emUSB-Device or even without any USB stack for devices that make use of a USB-C charger as a power source or to charge a dedicated battery.

1.2 emUSB-C features

Here is a list of emUSB-C features:

- Device connection detection on USB Type-C connectors.
- Power source / sink detection.
- Host / device USB data role detection.
- Dead battery signaling (if supported by hardware).
- Dynamic power supply negotiation up to 20V 3A.
- Use of chargers with variable supply voltage.
- ISO/ANSI C source code.
- Simple configuration.
- Configurable for minimal memory footprint.
- Very easy API.

1.3 Basic concepts

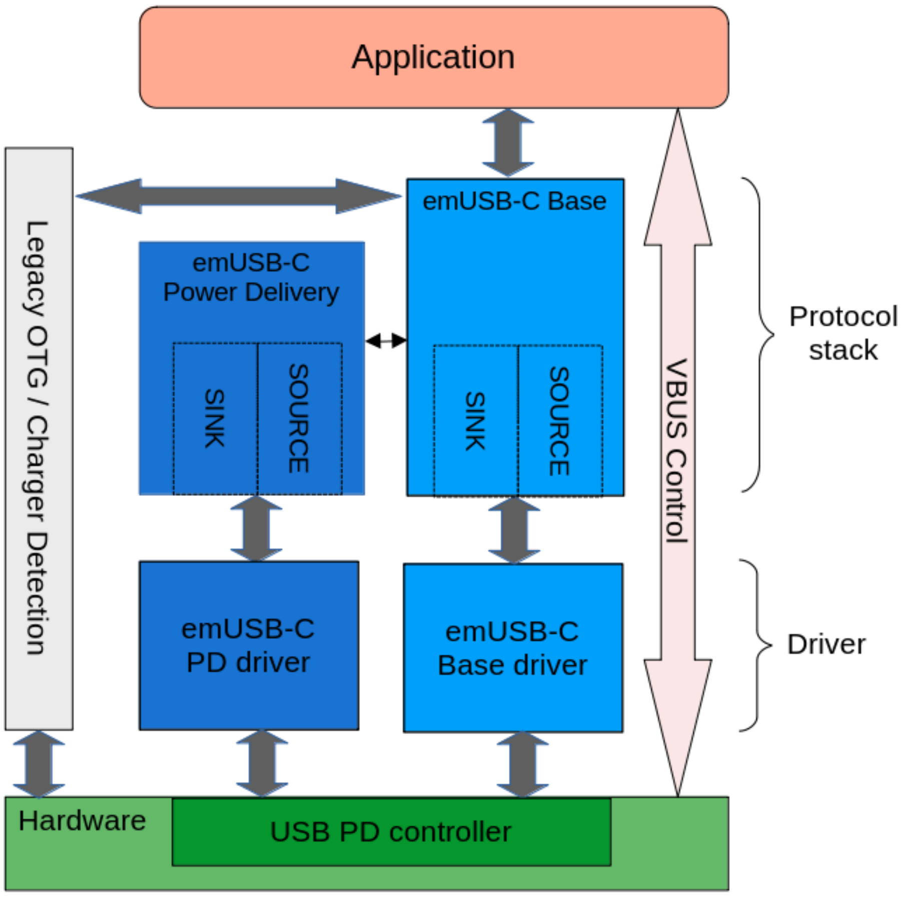
A USB-C connector has two special pins called CC1 and CC2. Initially these are used to detect a connection to another device by sensing the CC pins for a specified resistance to GND or Vcc. The power role of both connected devices are established in that way: One device becomes the Source (→ power provider) and the other becomes the Sink (→ power consumer). Power is not applied by the Source to the USB Type-C receptacle until it detects the presence of an attached device (Sink) port.

A Source may implement higher source current over VBUS than the default 5V 500mA. It may advertise 1.5A or 3A current (at 5V) via a specified resistance of the CC pins to Vcc to the Sink.

After (static) sensing of the CC pins and after a connection is established, both devices may start a power delivery communication by exchanging data packets serially via one of the CC pins. Using the power delivery protocol power contracts with much higher voltage and current, up to 20V 5A, may be negotiated between Source and Sink.

Power delivery communication is optional and necessary only, if such extended functionality are required by a device.

emUSB-C consists of two layers: The device independent emUSB-C protocol stack and a driver to handle the specific target hardware. Drivers are available for several different target hardware controllers. Both layers are divided into two functional modules: The "Base" modules, that provide the API for the application and handle the static sensing of the CC pins, and the "PD" module, that is responsible for the Power Delivery packet communication.



The base modules can be used separately for applications that do not need the extended functionality of the power delivery communication.

The application always uses the same API independent of the selected modules of emUSB-C and the target driver.

The emUSB-C API can also be used in connection with a legacy OTG driver to implement OTG functionality on a device without a USB-C connector.

Chapter 2

Running emUSB-C on target hardware

This chapter explains how to integrate and run emUSB-C on your target hardware.

2.1 Integrating emUSB-C

We assume that you are familiar with the tools you have selected for your project (compiler, project manager, linker, etc.). You should therefore be able to add files, add directories to the include search path, and so on. Any IDE or ANSI C toolchain can be used. It is also possible to use makefiles. In this case, when we say “add to the project”, this translates into “add to the makefile”.

Procedure to follow

Integration of emUSB-C is a relatively simple process, which consists of the following steps:

- Take a running project for your target hardware.
- Add emUSB-C files to the project.
- Add hardware dependent configuration to the project.
- Prepare and run the application.

2.2 Take a running project

The project to start with should include the setup for basic hardware (e.g. CPU, PLL, DDR SDRAM) and initialization of the RTOS (if used). If you are using SEGGER’s real-time operating system embOS, simply start with an embOS sample project and include emUSB-C into this project.

2.3 Add emUSB-C files

Add all necessary source files from the `USBC` folder to your project. You may simply add all files and let the linker drop everything not needed for your configuration.

Configuring the include path

The include path is the path in which the compiler looks for include files. In cases where the included files (typically header files, .h) do not reside in the same folder as the C file to compile, an include path needs to be set. In order to build the project with all added files, you will need to add the following directories to your include path:

- `Config`
- `Inc`
- `SEGGER`
- `USBC`

2.4 Configuring debugging output

While developing and testing emUSB-C, we recommend to use the DEBUG configuration of emUSB-C. This is enabled by setting the preprocessor symbol `DEBUG` to 1 (or `USBC_DEBUG` ≥ 2), see `USBC_Conf.h`. The DEBUG configuration contains many additional run-time checks and generate debug output messages which are very useful to identify problems that may occur during development. In case of a fatal problem (e.g. an invalid configuration) the program will end up in the function *`USBC_Panic()`* with a appropriate error message that describes the cause of the problem.

In order to use the debugging output, there must be any method to output messages, for example to a debug terminal. If you are already using SEGGER’s emUSB-Device in your project, then set

```
#define USBC_DEBUG_LOG_MODE  2
```

in `USBC_Conf.h`. emUSB-C will then use the logging mechanism from emUSB-Device.

If you are not using emUSB-Device but SEGGER’s emUSB-Host in your project, then set

```
#define USBC_DEBUG_LOG_MODE  1
```

in `USBC_Conf.h`. emUSB-C will then use the logging mechanism from emUSB-Host.

If you are not using either of these products, then set

```
#define USBC_DEBUG_LOG_MODE  0
```

in `USBC_Conf.h`, add the file `USBC_ConfigIO.c` found in the folder `Config` to your project and configure it to match the message output method used by your debugging tools. If possible use RTT.

To later compile a release configuration, which has a significant smaller code footprint, simply set the preprocessor symbol `DEBUG` (or `USBC_DEBUG`) to 0.

2.5 Add hardware dependent configuration

To perform target hardware dependent runtime configuration, the emUSB-C stack calls a function named `USBC_X_Config`. Typical tasks that may be done inside this function are:

- Select appropriate driver modules for the USB-C controller.
- Configure I/O pins of the MCU for USB.
- Configure PLL and clock divider necessary for USB power delivery operation.
- Configure functions to use operating system services.
- Install an interrupt service routine for USB power delivery and set interrupt priority.

Details can be found in *Runtime configuration* on page 34.

Sample configurations for popular evaluation boards are supplied with the driver shipment. They can be found in files called `USBC_Config_<TargetName>.c` in the folders `BSP/<BoardName>/Setup`. There are also some sample configuration files for emUSB-Device or emUSB-Host that contain a setup for emUSB-C.

Add the appropriate configuration file to your project. If there is no configuration file for your target hardware, take a file for a similar hardware and modify it if necessary.

If the file needs modifications, we recommend to copy it into the directory `Config` for easy updates to later versions of emUSB-C.

Add BSP file

Some targets require CPU specific functions for initialization, mainly for installing an interrupt service routine. They are contained in the file `BSP_USB.c`. USB interrupt priority can also be configured in `BSP_USB.c`.

Sample `BSP_USB.c` files for popular evaluation boards are supplied with the driver shipment. They can be found in the folders `BSP/<BoardName>/Setup`.

Add the appropriate `BSP_USB.c` file to your project. If there is no BSP file for your target hardware, take a file for a similar hardware and modify it if necessary.

If the file needs modifications, we recommend to copy it into the directory `Config` for easy updates to later versions of emUSB-C.

Note that a `BSP_USB.c` file is not always required, because for some target hardware all runtime configuration is done in `USBC_X_Config`.

2.6 Prepare and run the application

Choose a sample application from the folder `Application` and add it to your project. Sample applications are described in *Example applications*. Compile and run the application on the target hardware.

Write your own application

Take one of the sample applications as a starting point to write your own application. Details can be found in chapter *Using emUSB-C in an application*.

2.7 Updating emUSB-C

If an existing project should be updated to a later emUSB-C version, only files have to be replaced. You should have received the emUSB-C update as a zip file. Unzip this file to the location of your choice and replace all emUSB-C files in your project with the newer files from the emUSB-C update shipment.

In general, all files from the following directories have to be updated:

- `USBC`
- `Inc`
- `SEGGER`
- `Doc`

Some files may contain modification required for project specific customization. These files should reside in the folder `Config` and must not be overwritten. This includes:

- `USBC_Conf.h`
- `USBC_ConfigIO.c`
- `BSP_USB.c`
- `USBC_Config_<TargetName>.c`

Chapter 3

Using emUSB-C in an application

This chapter explains how build up an application with emUSB-C.

3.1 Basic application structure

For each USB port that should be handled with emUSB-C the application has to provide a memory structure of type `USBC_INSTANCE` and call the function `USBC_Init()` to initialize it. This structure must remain valid, until the emUSB-C is shut down using `USBC_Exit()`.

During initialization with `USBC_Init()` the function `USBC_X_Config()` is called to perform target and application specific configuration and select the target driver. `USBC_X_Config()` must be provided by the application. A detailed description of all tasks that need to be performed within this function can be found in *Configuring emUSB-C*.

After calling `USBC_Init()` the application has to configure the properties of the USB port by setting public members of the `USBC_INSTANCE` structure. This has to be done before the function `USBC_Process()` is called the first time. USB port properties include:

- Provided source or sink role of the port.
- Provided USB host or device role.
- Power requirements / capabilities.

In order to run the emUSB-C stack, the application has to call the function `USBC_Process()` periodically. We suggest to call it approximately every 10 ms, but there is no need for an exact interval timing. This approach was chosen to allow the use of emUSB-C in systems without an operating system, for example in a super loop architecture. If the target runs an operating system, the application may create a task with a loop, that calls `USBC_Process()` every 10 ms. Every time `USBC_Process()` returns, the application must respond to any state changes of the USB port, for example by starting / stopping the USB stack or switching VBUS on or off. This is described in *Running the application*.

3.2 Configuring properties of the USB port

The application should set the public members of the structure `USBC_INSTANCE` to configure the behavior of the USB-C port. The items must be set, after calling `USBC_Init()`, but before the function `USBC_Process()` is called the first time. Some of the items can be changed later during operation. emUSB-C will use reasonable defaults for members that are not set.

3.2.1 General configuration items.

3.2.1.1 CCDebounceTime

Debounce time in milliseconds for the CC lines. The signals on the CC lines must be stable for at least `CCDebounceTime` in order to detect a new USB-C connection.

3.2.1.2 UserContext

This member is not touched by the USB-C stack and can be arbitrarily used by the application to store context information. Either:

- `UserContext.N`: A 32 bit number containing a port / address or similar of the device instance, or
- `UserContext.p`: A pointer to any context information.

3.2.2 Configuration items used in sink role.

A device is in sink mode, if it consumes power from an attached 'source' device.

3.2.2.1 ProvidedSinkRole

`ProvidedSinkRole` is a bit mask containing options applied in sink mode. Multiple options can be or'ed together into `ProvidedSinkRole`. If the target can operate in sink mode, then at least the option `USBC_POWER_ROLE_SINK` must be set. Optionally `USBC_DATA_ROLE_DEVICE` and/or `USBC_DATA_ROLE_HOST` can be set, if the target can operate as USB device or USB host respectively. Please notice, that USB host operation is very unusual in sink mode and requires a data role swap procedure.

If the target doesn't support sink mode, `ProvidedSinkRole` must be `0`.

3.2.2.2 SinkPowerOptions and NumSinkPowerOptions

`SinkPowerOptions` is an array of `USBC_POWER_REQUEST` structures, each containing a power requirement profile the sink can work with. `NumSinkPowerOptions` must be set to the number of `USBC_POWER_REQUEST` structures provided in the array. If `NumSinkPowerOptions = 0`, then a default of 5V and 500 mA requirement is used. If more than one power requirement profiles are provided (`NumSinkPowerOptions > 1`), then the first one in the tables that matches any power profile of the source will be selected. If none matches, the default power supply of the source will be used.

Warning

Never specify any voltage in `SinkPowerOptions` that is higher than you hardware can handle! Otherwise the hardware may get damaged by high voltages supplied by a source.

If the power requirements change during operation, then the application may change `SinkPowerOptions` and set `RenegotiatePD` to `1`. This will trigger a new power negotiation with the source.

Example

A sink device uses `SinkPowerOptions = [ 15V, 2A ]` while charging it's battery. After the battery is full, it may set `NumSinkPowerOptions` to 0 and `RenegotiatePD` to 1 to return to the default power supply.

3.2.2.3 DeviceDataRoleDelay

A delay in milliseconds between the detection of a connected source and the signaling of the device data role (`USBC_DATA_ROLE_DEVICE` in `ActualRole`) to the application.

When a source is connected, it signals via the CC lines that it is a power source **and** a USB host. If then later a power delivery communication is established, the source may indicate via its capabilities that this device is newer a USB host. The USB-C stack will then reset the bit `USBC_DATA_ROLE_DEVICE` in `ActualRole` again. This is a typical behavior of USB power supplies that are capable of power delivery communication. The result will be usually, that the application starts the USB device stack at connection time and then stops the USB stack around 100-200 ms later after the source capabilities have been received via PD communication.

If `DeviceDataRoleDelay` is set to a non zero value, then after a new connection of a source the signaling of `USBC_DATA_ROLE_DEVICE` to the application is delayed until either the source capabilities have been received via PD communication or `DeviceDataRoleDelay` is expired. This should avoid any useless starting and stopping of the USB device stack.

At time of a new connection it is unknown, if the connected device is capable of power delivery communication or not. For devices that can't do power delivery communication, USB device role is signaled after expiring of `DeviceDataRoleDelay`, even if the device can't do any USB communication, which can't be distinguished.

If power delivery communication is not configured and therefore not used by the USB-C stack, `DeviceDataRoleDelay` should be left zero.

3.2.3 Configuration items used in source role.

A device is in source mode, if it provides power for an attached 'sink' device.

3.2.3.1 ProvidedSourceRole

`ProvidedSourceRole` is a bit mask containing options applied in source mode. Multiple options can be or'ed together into `ProvidedSourceRole`. If the target can operate in source mode, then at least the option `USBC_POWER_ROLE_SOURCE` must be set. Optionally `USBC_DATA_ROLE_DEVICE` and/or `USBC_DATA_ROLE_HOST` can be set, if the target can operate as USB device or USB host respectively. Please notice, that USB device operation is very unusual in source mode and requires a data role swap procedure.

If the target doesn't support source mode, `ProvidedSourceRole` must be 0.

3.2.3.2 ProvidedPowerProfiles and NumSourcePowerProfiles

`ProvidedPowerProfiles` is an array of `USBC_POWER_PROFILE` structures, each containing a power profile the source offers. `NumSourcePowerProfiles` must be set to the number of `USBC_POWER_PROFILE` structures provided in the array. If `NumSourcePowerProfiles =` 0, then a default of 5V and 500 mA requirement is used. If power profiles are provided (`NumSinkPowerOptions` ≠ 0), then the first one must be a 5V profile.

3.2.3.3 VBUSDischargeDelay

Delay time in milliseconds after disconnect of a sink device before a new connection can be detected. This is the time required for the VBUS voltage to drop below the threshold of 0.8 V. This field is ignored if a discharge function is registered, see `USBC_Reg_DISCHRG_Custom()`.

3.3 Running the application

The application has to call the function `USBC_Process()` approximately every 10 ms. This function returns a bit mask (see `USBC_CHANGED_…` macros) that reflects changes in the state of the USB-C port. The application has to respond to any changes as follows:

3.3.1 Bit USBC_CHANGED_POWER_ROLE

The application must inspect the member `ActualRole` of the USBC instance.

If `(ActualRole & USBC_POWER_ROLE_SOURCE) = 0`, then the application MUST immediately switch off VBUS supply (if active, source role only). Note that depending on the configuration, shutting down of a USB host stack may also switch VBUS off, see `USBC_CHANGED_USB_COMM` below.

If `(ActualRole & USBC_POWER_ROLE_SOURCE) <> 0`, then the application SHOULD switch on VBUS supply (if not already on, source role only). Note that depending on the configuration, starting up a USB host stack may also switch VBUS on, see `USBC_CHANGED_USB_COMM` below.

`ActualRole` shows the current connection state of the USB-C port:

- Bit `USBC_POWER_ROLE_SOURCE`: Connected to a sink.
- Bit `USBC_POWER_ROLE_SINK`: Connected to a source.
- None of these bits set: No connection.

3.3.2 Bit USBC_CHANGED_USB_COMM

The application must inspect the member `ActualRole` of the USBC instance.

If `(ActualRole & USBC_DATA_ROLE_HOST) = 0`, then the application MUST immediately shut down a USB host stack, if running.

If `(ActualRole & USBC_DATA_ROLE_DEVICE) = 0`, then the application MUST immediately shut down a USB device stack, if running. If a device only supports sink role and USB device data role, then the USB device stack may be running continuously.

If `(ActualRole & USBC_DATA_ROLE_DEVICE) <> 0`, then the application SHOULD start the USB device stack, if not already running.

If `(ActualRole & USBC_DATA_ROLE_HOST) <> 0`, then the application SHOULD start the USB host stack, if not already running.

3.3.3 Bit USBC_CHANGED_POWER

The application must inspect the member `ActivePowerProfile` of the USBC instance, see `USBC_POWER_PROFILE`. It shows the current power contract negotiated with the connected device.

In sink role the device SHOULD adjust it's power consumption accordingly.

In source role the device MUST configure it's power supplied to the USB-C port to match this setting and then set the member `PowerSupplyReady` to 1 after the power supply is ready.

3.3.4 Bit USBC_CHANGED_CAPABILITIES

Sink role only. The connected source has provided new power capabilities, which may be inspected using the function `USBC_GetPowerProfiles()`.

3.4 Example applications

The following sample applications are provided with the emUSB-C package:

USBC_OTG_Start.c

This sample demonstrates switching between device and host roles and combines the functionality of the host sample to mount a USB stick and the device sample implementing a USB mouse.

USBC_Device_BULK.c

Sample application for a USB device vendor class that request higher voltage and current from the source, e.g. for battery charging.

3.5 API reference of functions for the application

This section describes the functions that can be used by the target application.

Function	Description
API Functions	
USBC_Init()	Initialize a USBC instance.
USBC_Exit()	Shut down a USBC instance.
USBC_Process()	The main USBC state machine.
USBC_GetPowerProfiles()	Get a list of all power profiles (Power Data Objects) sent by the source.
USBC_SetOnHardReset()	Sets a callback function that is called when a hard reset is sent or received on the CC lines.
Data structures	
USBC_POWER_REQUEST	Power requirements when operating in sink role.
USBC_POWER_PROFILE	A power profile.
Function prototypes	
USBC_ON_HARD_RESET	Type of callback set in `USBC_SetOn-HardReset()`.

3.5.1 USBC_Init()

Description

Initialize a USBC instance. Must be called before any other USBC function.

Prototype

```
void USBC_Init(USBC_INSTANCE * pInst);
```

Parameters

Parameter	Description
pInst	Pointer to a structure where all information of an active USBC instance is stored and maintained. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running, and may be destroyed only after USBC_Exit() was called.

3.5.2 USBC_Exit()

Description

Shut down a USBC instance. The `USBC_INSTANCE` structure may be destroyed after return of this function.

Prototype

```
void USBC_Exit(USBC_INSTANCE * pInst);
```

Parameters

Parameter	Description
`pInst`	Pointer to the USBC instance.

3.5.3 USBC_Process()

Description

The main USBC state machine. This function must be called on at regular intervals to handle all USBC events. An interval of 10 ms is recommended.

Prototype

```
unsigned USBC_Process(USBC_INSTANCE * pInst);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.

Return value

Bit mask with a combination of USBC_CHANGED_... bits or 0 if port status has not changed.

- USBC_CHANGED_USB_COMM - USB data role of the port has changed.
- USBC_CHANGED_POWER_ROLE - Power role has changed, also connect or disconnect detected.
- USBC_CHANGED_POWER - Power characteristics have changed.
- USBC_CHANGED_CAPABILITIES - New power capabilities provided by the source.

3.5.4 USBC_GetPowerProfiles()

Description

Get a list of all power profiles (Power Data Objects) sent by the source. Only available when in sink mode and after successful PD communication.

Prototype

```
unsigned USBC_GetPowerProfiles(const USBC_INSTANCE      * pInst,
                                     USBC_POWER_PROFILE * pPowerProfiles);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPowerProfiles	Must point to an array of USBC_POWER_PROFILE structures with USBC_MAX_POWER_DATA_OBJECTS elements where this function fills in the information.

Return value

Number of elements stored into pPowerProfiles. May be 0 if no information is available.

3.5.5 USBC_SetOnHardReset()

Description

Sets a callback function that is called when a hard reset is sent or received on the CC lines.

Prototype

```
void USBC_SetOnHardReset(const USBC_INSTANCE      * pInst,
                               USBC_ON_HARD_RESET * pfOnHardReset);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pfOnHardReset	Pointer to the callback function.

Additional information

Note that the callback will be called within an ISR, therefore it should never block.

3.5.6 USBC_POWER_REQUEST

Description

Power requirements when operating in sink role.

The following must always apply: MaxVoltage ≥ 5000, MinVoltage ≥ 3300, MaxVoltage ≥ MinVoltage and MaxCurrent ≥ MinCurrent. If a sink is self powered and doesn't consume any power then MaxVoltage and MinVoltage should be set to 5000 and MaxCurrent and MinCurrent should be set to 0.

Type definition

```
typedef struct {
  U16  MaxVoltage;
  U16  MinVoltage;
  U16  MaxCurrent;
  U16  MinCurrent;
} USBC_POWER_REQUEST;
```

Structure members

Member	Description
MaxVoltage	Maximum voltage in mV the device can operate with.
MinVoltage	Minimum voltage in mV the device requires to operate.
MaxCurrent	Maximum current in mA the device can use in the specified voltage range.
MinCurrent	Minimum current in mA the device needs to operate.

3.5.7 USBC_POWER_PROFILE

Description

A power profile. Used to describe a power capability of a source or a negotiated power contract.

Type definition

```
typedef struct {
  U16  MaxVoltage;
  U16  MinVoltage;
  U16  MaxCurrent;
  U16  Flags;
} USBC_POWER_PROFILE;
```

Structure members

Member	Description
MaxVoltage	Maximum voltage in mV
MinVoltage	Minimum voltage in mV
MaxCurrent	Maximum current in mA
Flags	Bit mask of USBC_PDO_FLG_... flags. • USBC_PDO_FLG_EXPLICIT - This can only be set in USBC_INSTANCE.ActivePowerProfile and indicates an explicit power contract negotiated via USB power delivery. • USBC_PDO_FLG_PROGRAMMABLE - This flag indicates a programmable power supply of the source, where the sink can choose a specific voltage between MinVoltage and MinVoltage. If this flag is not set and MinVoltage ≠ MaxVoltage, then the source provides an unregulated supply in that voltage range. • USBC_PDO_FLG_INVALID - Internal use only.

3.5.8 USBC_ON_HARD_RESET

Description

Type of callback set in `USBC_SetOnHardReset()`. This function is called when a hard reset is sent or received on the CC lines. It must not block or call any other USBC functions, because it is called in an interrupt context.

Type definition

```
typedef void USBC_ON_HARD_RESET(USBC_INSTANCE * pInst,
                                int             Origin);
```

Parameters

Parameter	Description
`pInst`	Pointer to the USBC instance.
`Origin`	• 0 - Hard reset was received from the remote side. • 1 - Hard reset was issued by the USBC stack.

Chapter 4

Configuring emUSB-C

This chapter explains how to configure emUSB-C.

4.1 Runtime configuration

The configuration of emUSB-C for a target hardware is done at runtime: The emUSB-C stack calls a function named `USBC_X_Config()`, that must be provided by the application. This function has to perform the following tasks:

Board specific hardware initialization

This includes configuring I/O pins of the MCU, setting up PLL and clock divider necessary for USB-C and installing the interrupt service routine for USB-C.

OS service registration

A function that returns the current systems time in milliseconds must be registered with `USBC_SetTimeFunc()`. If emUSB-Device or emUSB-Host is used in the project, then one of the functions `USB_OS_GetTickCnt()` or `USBH_OS_GetTime32()` can be used.

Optional re-initialization function

In some configurations the USB hardware need to be re-initialized after each disconnect, because a USB host or device stack may leave the USB hardware in an inappropriate state after shut down. A function that performs a re-initialization of the USB hardware can be registered using `USBC_SetReInitFunc()`.

Driver installation:

See next section.

4.1.1 Driver installation

USB-C drivers consist of modules that can be registered separately at the USB-C stack. This modular approach was chosen to allow including only code that is required for a particular application, avoiding to include unused code and achieving a small footprint.

For example: If a device only works in sink role, then all code for handling the source role can be excluded from the project. Another device may not need power delivery communication, but just handle the static state of the CC lines, then the power delivery communication stack is not required in the project.

For this reason there are different functions to register driver modules in each driver packet, see *USB-C controller specifics*. Only those should be selected that are required for the actual application.

Additionally some functions are always board specific and must be provided by the user. This includes:

VBUS detection

There is no general way to detect a valid VBUS voltage. On some boards there is a GPIO for that purpose. Other boards uses an ADC of the microcontroller to measure the VBUS voltage. It’s also possible to have a separate chip on the board that is connected via I2C.

Therefore the application must implement a function `USBC_X_GetVBUS()` that checks for a valid VBUS voltage and register it during configuration using `USBC_Reg_VBUS_Func()`.

VBUS discharging

Some operations require VBUS to be discharged below 0.8V. The application may optionally implement a function `USBC_X_Discharge()` that controls discharging of VBUS and register it during configuration using `USBC_Reg_DISCHRG_Func()`.

All driver modules need some memory to store their internal states and data. This memory must be provided by the application. A pointer to a driver module individual private data structure (`USBC_PRV_`… type) must be passed to the registration function of a driver module. This structure may be implemented as a static variable or may be allocated by the application.

4.1.2 Configuration functions

Functions that may or must be used in `USBC_X_Config` are listed in the following table. Additional driver dependent functions exist for every USB-C controller driver, see *USB-C controller specifics*.

Function	Description
`USBC_SetTimeFunc()`	Sets a function that returns the current systems time in ms.
`USBC_SetReInitFunc()`	Sets a function that re-initializes the USB hardware after each disconnect.
`USBC_Reg_VBUS_Func()`	Register a custom driver function to determine the state of VBUS voltage.
`USBC_Reg_DISCHRG_Func()`	Register a custom driver function for VBUS discharging.
`USBC_Reg_IDPIN_Func()`	Register a custom driver function to determine the state of the ID pin.
User provided driver functions	
`USBC_TIME_FUNC`	Returns the current system time in milliseconds.
`USBC_CALLBACK_FUNC`	Callback user function, that may be called from the USB-C stack.
`USBC_GET_VBUS_FUNC`	VBUS detection.
`USBC_DISCHARGE_FUNC`	VBUS discharging.
`USBC_GET_ID_PIN_FUNC`	Determine state of ID pin.

4.1.3 USBC_SetTimeFunc()

Description

Sets a function that returns the current systems time in ms.

Prototype

```
void USBC_SetTimeFunc(USBC_INSTANCE  * pInst,
                      USBC_TIME_FUNC * pfGetTime);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pfGetTime	Pointer to the function.

4.1.4 USBC_SetReInitFunc()

Description

Sets a function that re-initializes the USB hardware after each disconnect.

Prototype

```
void USBC_SetReInitFunc(USBC_INSTANCE      * pInst,
                        USBC_CALLBACK_FUNC * pfReInit);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pfReInit	Pointer to the function.

4.1.5 USBC_Reg_VBUS_Func()

Description

Register a custom driver function to determine the state of VBUS voltage.

Prototype

```
void USBC_Reg_VBUS_Func(USBC_INSTANCE      * pInst,
                        USBC_GET_VBUS_FUNC * pf);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pf	Custom function.

4.1.6 USBC_Reg_DISCHRG_Func()

Description

Register a custom driver function for VBUS discharging. The callback function `USBC_X_DischargeVBUS()` will be called to actually perform discharging,

Prototype

```
void USBC_Reg_DISCHRG_Func(USBC_INSTANCE       * pInst,
                           USBC_DISCHARGE_FUNC * pf);
```

Parameters

Parameter	Description
`pInst`	Pointer to the USBC instance.
`pf`	Custom function.

4.1.7 USBC_Reg_IDPIN_Func()

Description

Register a custom driver function to determine the state of the ID pin.

Prototype

```
void USBC_Reg_IDPIN_Func(USBC_INSTANCE        * pInst,
                         USBC_GET_ID_PIN_FUNC * pf);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pf	Custom function.

4.1.8 USBC_GET_VBUS_FUNC

Description

VBUS detection. This function is called by the USB-C stack to check for a valid voltage on the VBUS pin.

Type definition

```
typedef int USBC_GET_VBUS_FUNC(const USBC_INSTANCE * pInst);
```

Parameters

Parameter	Description
`pInst`	Pointer to the USBC instance.

Return value

- 1 - if there is a valid voltage on the VBUS pin (≥ 4.75 V).
- 0 - otherwise.

4.1.9 USBC_DISCHARGE_FUNC

Description

VBUS discharging. This function is called by the USB-C stack to perform discharging of VBUS. This function will be called repeatedly until it returns 0.

Type definition

```
typedef int USBC_DISCHARGE_FUNC(const USBC_INSTANCE * pInst,
                                      int              Operation);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
Operation	• 0 - Start discharging of VBUS. • 1 - Continue discharging of VBUS. • -1 - Stop discharging / disable discharger unit.

Return value

- 1 - Discharging still in progress. VBUS not below 0.8V.
- 0 - Discharging completed.

4.1.10 USBC_GET_ID_PIN_FUNC

Description

Determine state of ID pin. This function is called by the USB-C stack to sense the ID pin of a USB connector. For legacy OTG only.

Type definition

```
typedef int USBC_GET_ID_PIN_FUNC(const USBC_INSTANCE * pInst);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.

Return value

- 0 - ID pin pulled to ground: Host mode.
- 1 - ID pin high or open.

4.1.11 USBC_TIME_FUNC

Description

Returns the current system time in milliseconds.

Type definition

```
typedef USBC_TIME USBC_TIME_FUNC(void);
```

Return value

Current system time.

4.1.12 USBC_CALLBACK_FUNC

Description

Callback user function, that may be called from the USB-C stack.

Type definition

```
typedef void USBC_CALLBACK_FUNC(USBC_INSTANCE * pInst);
```

Parameters

Parameter	Description
`pInst`	Pointer to the USBC instance.

4.2 Compile-time configuration

emUSB-C can be used without changing any of the compile-time switches. All compile-time configuration switches are preconfigured with valid values which match the requirements of most applications. All compile-time switches and their default values can be found in the file `USBC_ConfDefaults.h`.

To change the default configuration of emUSB-C compile-time switches can be added to `USBC_Conf.h`. Don't change the `USBC_ConfDefaults.h` file for easy updates of emUSB-C PD.

4.2.1 Compile-time switches for debugging

4.2.1.1 USBC_DEBUG

Description

emUSB-C can be configured to display debug messages and warnings to locate an error or potential problems. This can be useful for debugging. In a release (production) build of a target system, they are typically not required and should be switched off.

To output the messages, emUSB-C uses the logging routines contained in `USBC_ConfigIO.c` which can be customized.

`USBC_DEBUG` can be set to the following values:

- 0 - Used for release builds. Includes no debug options.
- 1 - Used in debug builds to include support for "panic" checks.
- 2 - Used in debug builds to include warning messages and "panic" checks.
- 3 - Used in debug builds to include warning, log messages and "panic" checks.

Definition

```
#define USBC_DEBUG    0
```

4.2.1.2 USBC_DEBUG_LOG_MODE

Description

Configure how debug messages are output:

- = 0: Implement own functions in `USBC_ConfigIO.c`
- = 1: Use emUSB-Host functions
- = 2: Use emUSB-Device functions

Definition

```
#define USBC_DEBUG_LOG_MODE    1
```

4.2.1.3 USBC_LOG_BUFFER_SIZE

Description

Maximum size of a debug / warning message (in characters) that can be output. A buffer of this size is created on the stack when a message is output.

Definition

```
#define USBC_LOG_BUFFER_SIZE    200
```

4.2.2 Use of standard C-library functions

emUSB-C PD calls some functions from the standard C-library. If the standard C-library should not be used, the following macros can be changed to call user defined functions instead:

```
#define USBC_MEMCPY   memcpy
#define USBC_MEMSET   memset
```

4.3 USBC controller specifics

For emUSB-C different drivers are provided, each containing multiple driver modules. This section explains driver specific configuration functions and options.

4.3.1 STM32Uxx driver

This driver is used for the UCPD controller of the MCUs:

- STM32U5xx
- STM32H56x
- STM32H57x

The driver consists of separate modules for sink / source functionality and for static CC line handling and power delivery communication. By calling the appropriate registration functions `USBC_Reg_…()` it can be configured to include only the functionality actually needed for the target application, avoiding unused code and achieving minimal footprint. If any `USBC_Reg_PD_…()` function is called to enable power delivery communication, then the corresponding `USBC_Reg_CC_…()` must also be called.

The static CC line handling ("base" module) does not use any interrupt or DMA.

Power delivery communication requires an interrupt handler of the UCPD controller to be installed. Optional DMA can be used for packet transfers: This requires two channels of the GPDMA controller to be allocated to the USBC driver. If DMA is not used, then the interrupt must be serviced at least every 30µs while power delivery packet transfers are in progress.

4.3.2 STM32Uxx driver specific configuration functions

Function	Description
`USBC_Reg_CC_STM32Uxx()`	Register a driver function to check for static CC lines states and detect cable connections.
`USBC_Reg_CC_SNK_STM32Uxx()`	Register a driver function to check for static CC lines states and detect cable connections.
`USBC_Reg_CC_SRC_STM32Uxx()`	Register a driver function to check for static CC lines states and detect cable connections.
`USBC_Reg_PD_STM32Uxx()`	Register a driver function to perform PD message communication.
`USBC_Reg_PD_SNK_STM32Uxx()`	Register a driver function to perform PD message communication.
`USBC_Reg_PD_SRC_STM32Uxx()`	Register a driver function to perform PD message communication.
`USBC_EnableTrimming_STM32Uxx()`	Enable trimming of the CC pull-up and pull-down resistors.
`USBC_STM32Uxx_ISR()`	Interrupt service routine for the STM32Uxx UCPD controller.
`USBC_STM32Uxx_CFG_PARAMS`	Configuration parameter for the STM32Uxx PD driver.
`USBC_STM32Uxx_CC_ACTIVATE`	Callback function, that is called after configuring the CC lines.

4.3.2.1 USBC_Reg_CC_STM32Uxx()

Description

Register a driver function to check for static CC lines states and detect cable connections.

Prototype

```
void USBC_Reg_CC_STM32Uxx(      USBC_INSTANCE            * pInst,
                                USBC_PRV_CC_STM32Uxx     * pPrv,
                          const USBC_STM32Uxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running.
pCfg	Pointer to a structure containing configuration parameters for the driver. The DMA specific members of pCfg are not used (ignored).

4.3.2.2 USBC_Reg_CC_SNK_STM32Uxx()

Description

Register a driver function to check for static CC lines states and detect cable connections. For SINK only use.

Prototype

```
void USBC_Reg_CC_SNK_STM32Uxx(      USBC_INSTANCE            * pInst,
                                    USBC_PRV_CC_STM32Uxx     * pPrv,
                              const USBC_STM32Uxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running.
pCfg	Pointer to a structure containing configuration parameters for the driver. The DMA specific members of pCfg are not used (ignored).

4.3.2.3 USBC_Reg_CC_SRC_STM32Uxx()

Description

Register a driver function to check for static CC lines states and detect cable connections. For SOURCE only use.

Prototype

```
void USBC_Reg_CC_SRC_STM32Uxx(      USBC_INSTANCE            * pInst,
                                    USBC_PRV_CC_STM32Uxx     * pPrv,
                              const USBC_STM32Uxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running.
pCfg	Pointer to a structure containing configuration parameters for the driver. The DMA specific members of pCfg are not used (ignored).

4.3.2.4 USBC_Reg_PD_STM32Uxx()

Description

Register a driver function to perform PD message communication.

Prototype

```
void USBC_Reg_PD_STM32Uxx(      USBC_INSTANCE            * pInst,
                                USBC_PRV_PD_STM32Uxx     * pPrv,
                          const USBC_STM32Uxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
`pInst`	Pointer to the USBC instance.
`pPrv`	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running. If DMA is used (`pCfg->GPDMA_RX/TX_BaseAddress ≠ NULL`) then this memory area must be accessible by DMA.
`pCfg`	Pointer to a structure containing configuration parameters for the driver.

Additional information

If DMA is not used, then the interrupt must be serviced at least every 30us for PD packet transfers.

4.3.2.5 USBC_Reg_PD_SNK_STM32Uxx()

Description

Register a driver function to perform PD message communication. For SINK only use.

Prototype

```
void USBC_Reg_PD_SNK_STM32Uxx(      USBC_INSTANCE             * pInst,
                                    USBC_PRV_PD_STM32Uxx      * pPrv,
                              const USBC_STM32Uxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running. If DMA is used (pCfg->GPDMA_RX/TX_BaseAddress ≠ NULL) then this memory area must be accessible by DMA.
pCfg	Pointer to a structure containing configuration parameters for the driver.

Additional information

If DMA is not used, then the interrupt must be serviced at least every 30us for PD packet transfers.

4.3.2.6 USBC_Reg_PD_SRC_STM32Uxx()

Description

Register a driver function to perform PD message communication. For SOURCE only use.

Prototype

```
void USBC_Reg_PD_SRC_STM32Uxx(      USBC_INSTANCE            * pInst,
                                    USBC_PRV_PD_STM32Uxx     * pPrv,
                              const USBC_STM32Uxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running. If DMA is used (pCfg->GPDMA_RX/TX_BaseAddress ≠ NULL) then this memory area must be accessible by DMA.
pCfg	Pointer to a structure containing configuration parameters for the driver.

Additional information

If DMA is not used, then the interrupt must be serviced at least every 30us for PD packet transfers.

4.3.2.7 USBC_EnableTrimming_STM32Uxx()

Description

Enable trimming of the CC pull-up and pull-down resistors. This function must only be used for devices that don’t support fully automatic trimming. Don’t use for devices other than STM32Uxx.

Prototype

```
void USBC_EnableTrimming_STM32Uxx(USBC_PRV_CC_STM32Uxx * pPrv);
```

Parameters

Parameter	Description
pPrv	Pointer to a structure containing the drivers private data, that was initialized by a call to any of the USBC_Reg_CC…() functions before.

4.3.2.8 USBC_STM32Uxx_ISR()

Description

Interrupt service routine for the STM32Uxx UCPD controller.

Prototype

```
void USBC_STM32Uxx_ISR(USBC_PRV_PD_STM32Uxx * pPrv);
```

Parameters

Parameter	Description
pPrv	Pointer to the private data structure of the driver.

4.3.2.9 USBC_STM32Uxx_CFG_PARAMS

Description

Configuration parameter for the STM32Uxx PD driver.

Type definition

```
typedef struct {
  PTR_ADDR                     BaseAddress;
  PTR_ADDR                     GPDMA_RX_BaseAddress;
  PTR_ADDR                     GPDMA_TX_BaseAddress;
  USBC_STM32Uxx_CC_ACTIVATE * pfActivateCC;
} USBC_STM32Uxx_CFG_PARAMS;
```

Structure members

Member	Description
BaseAddress	Base address of the UCPD controller.
GPDMA_RX_BaseAddress	Base address of the GPDMA channel used for RX or NULL, if DMA should not be used.
GPDMA_TX_BaseAddress	Base address of the GPDMA channel used for TX or NULL, if DMA should not be used.
pfActivateCC	(Optional) Use for configuration action after enable of the PD controller.

4.3.2.10 USBC_STM32Uxx_CC_ACTIVATE

Description

Callback function, that is called after configuring the CC lines. Can be used for device dependent configuration.

Type definition

```
typedef int USBC_STM32Uxx_CC_ACTIVATE(USBC_INSTANCE * pInst);
```

Parameters

Parameter	Description
`pInst`	Pointer to the USBC instance.

Return value

- 0 - Configuration done. This function is not called again any more.
- 1 - Action must be executed before every CC sensing. This function will be called again.

4.3.3 STM32Gxx driver

This driver is used for the UCPD controller of the MCUs:

- STM32G0x1
- STM32G4xx

The driver consists of separate modules for sink / source functionality and for static CC line handling and power delivery communication. By calling the appropriate registration functions `USBC_Reg_…()` it can be configured to include only the functionality actually needed for the target application, avoiding unused code and achieving minimal footprint. If any `USBC_Reg_PD_…()` function is called to enable power delivery communication, then the corresponding `USBC_Reg_CC_…()` must also be called.

The static CC line handling ("base" module) does not use any interrupt or DMA.

Power delivery communication requires an interrupt handler of the UCPD controller to be installed. Optional DMA can be used for packet transfers: This required two channels of the DMA controller to be allocated to the USBC driver. If DMA is not used, then the interrupt must be serviced at least every 30µs while power delivery packet transfers are in progress.

4.3.4 STM32Gxx driver specific configuration functions

Function	Description
`USBC_Reg_CC_STM32Gxx()`	Register a driver function to check for static CC lines states and detect cable connections.
`USBC_Reg_CC_SNK_STM32Gxx()`	Register a driver function to check for static CC lines states and detect cable connections.
`USBC_Reg_CC_SRC_STM32Gxx()`	Register a driver function to check for static CC lines states and detect cable connections.
`USBC_Reg_PD_STM32Gxx()`	Register a driver function to perform PD message communication.
`USBC_Reg_PD_SNK_STM32Gxx()`	Register a driver function to perform PD message communication.
`USBC_Reg_PD_SRC_STM32Gxx()`	Register a driver function to perform PD message communication.
`USBC_STM32Gxx_ISR()`	Interrupt service routine for the STM32Gxx UCPD controller.
`USBC_STM32Gxx_CFG_PARAMS`	Configuration parameter for the STM32Gxx PD driver.
`USBC_STM32Gxx_CC_ACTIVATE`	Callback function, that is called after configuring the CC lines.

4.3.4.1 USBC_Reg_CC_STM32Gxx()

Description

Register a driver function to check for static CC lines states and detect cable connections.

Prototype

```
void USBC_Reg_CC_STM32Gxx(      USBC_INSTANCE            * pInst,
                                USBC_PRV_CC_STM32Gxx     * pPrv,
                          const USBC_STM32Gxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
`pInst`	Pointer to the USBC instance.
`pPrv`	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running.
`pCfg`	Pointer to a structure containing configuration parameters for the driver. The DMA specific members of `pCfg` are not used (ignored).

4.3.4.2 USBC_Reg_CC_SNK_STM32Gxx()

Description

Register a driver function to check for static CC lines states and detect cable connections. For SINK only use.

Prototype

```
void USBC_Reg_CC_SNK_STM32Gxx(      USBC_INSTANCE             * pInst,
                                    USBC_PRV_CC_STM32Gxx      * pPrv,
                              const USBC_STM32Gxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
`pInst`	Pointer to the USBC instance.
`pPrv`	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running.
`pCfg`	Pointer to a structure containing configuration parameters for the driver. The DMA specific members of `pCfg` are not used (ignored).

4.3.4.3 USBC_Reg_CC_SRC_STM32Gxx()

Description

Register a driver function to check for static CC lines states and detect cable connections. For SOURCE only use.

Prototype

```
void USBC_Reg_CC_SRC_STM32Gxx(      USBC_INSTANCE            * pInst,
                                    USBC_PRV_CC_STM32Gxx     * pPrv,
                              const USBC_STM32Gxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running.
pCfg	Pointer to a structure containing configuration parameters for the driver. The DMA specific members of pCfg are not used (ignored).

4.3.4.4 USBC_Reg_PD_STM32Gxx()

Description

Register a driver function to perform PD message communication.

Prototype

```
void USBC_Reg_PD_STM32Gxx(      USBC_INSTANCE            * pInst,
                                USBC_PRV_PD_STM32Gxx     * pPrv,
                          const USBC_STM32Gxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running. If DMA is used (pCfg->DMA_RX/TX_Channel ≠ 0) then this memory area must be accessible by DMA.
pCfg	Pointer to a structure containing configuration parameters for the driver.

Additional information

If DMA is not used, then the interrupt must be serviced at least every 30us for PD packet transfers.

4.3.4.5 USBC_Reg_PD_SNK_STM32Gxx()

Description

Register a driver function to perform PD message communication. For SINK only use.

Prototype

```
void USBC_Reg_PD_SNK_STM32Gxx(      USBC_INSTANCE              * pInst,
                                    USBC_PRV_PD_STM32Gxx       * pPrv,
                              const USBC_STM32Gxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running. If DMA is used (pCfg->DMA_RX/TX_Channel ≠ 0) then this memory area must be accessible by DMA.
pCfg	Pointer to a structure containing configuration parameters for the driver.

Additional information

If DMA is not used, then the interrupt must be serviced at least every 30us for PD packet transfers.

4.3.4.6 USBC_Reg_PD_SRC_STM32Gxx()

Description

Register a driver function to perform PD message communication. For SOURCE only use.

Prototype

```
void USBC_Reg_PD_SRC_STM32Gxx(      USBC_INSTANCE           * pInst,
                                    USBC_PRV_PD_STM32Gxx    * pPrv,
                              const USBC_STM32Gxx_CFG_PARAMS * pCfg);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running. If DMA is used (pCfg->DMA_RX/TX_Channel ≠ 0) then this memory area must be accessible by DMA.
pCfg	Pointer to a structure containing configuration parameters for the driver.

Additional information

If DMA is not used, then the interrupt must be serviced at least every 30us for PD packet transfers.

4.3.4.7 USBC_STM32Gxx_ISR()

Description

Interrupt service routine for the STM32Gxx UCPD controller.

Prototype

```
void USBC_STM32Gxx_ISR(USBC_PRV_PD_STM32Gxx * pPrv);
```

Parameters

Parameter	Description
pPrv	Pointer to the private data structure of the driver.

4.3.4.8 USBC_STM32Gxx_CFG_PARAMS

Description

Configuration parameter for the STM32Gxx PD driver.

Type definition

```
typedef struct {
  PTR_ADDR                         BaseAddress;
  PTR_ADDR                         DMABaseAddress;
  U8                               DMA_RX_Channel;
  U8                               DMA_TX_Channel;
  USBC_STM32Gxx_CC_ACTIVATE * pfActivateCC;
} USBC_STM32Gxx_CFG_PARAMS;
```

Structure members

Member	Description
BaseAddress	Base address of the UCPD controller.
DMABaseAddress	Base address of the DMA controller. Ignored if DMA_RX_Channel and DMA_TX_Channel are both 0.
DMA_RX_Channel	DMA channel for RX (1-8), 0 means no DMA use.
DMA_TX_Channel	DMA channel for TX (1-8), 0 means no DMA use.
pfActivateCC	(Optional) Use for configuration action after enable of the PD controller.

4.3.4.9 USBC_STM32Gxx_CC_ACTIVATE

Description

Callback function, that is called after configuring the CC lines. Can be used for device dependent configuration.

Type definition

```
typedef int USBC_STM32Gxx_CC_ACTIVATE(USBC_INSTANCE * pInst);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.

Return value

- 0 - Configuration done. This function is not called again any more.
- 1 - Action must be executed before every CC sensing. This function will be called again.

4.3.5 STM32Fxx driver

This driver is used for legacy OTG operation (no USB type-C connector, no power delivery communication).

4.3.6 STM32Fxx driver specific configuration functions

Function	Description
`USBC_Reg_IDPIN_STM32F7xx()`	Register a driver function to determine the state of the ID pin.
`USBC_Reg_VBUS_STM32F7xx()`	Register a driver function to determine if VBUS is present.
`USBC_Reg_CHRDET_STM32F7xx()`	Register a driver function to perform charger detection.

4.3.6.1 USBC_Reg_IDPIN_STM32F7xx()

Description

Register a driver function to determine the state of the ID pin.

Prototype

```
void USBC_Reg_IDPIN_STM32F7xx(USBC_INSTANCE            * pInst,
                              USBC_PRV_IDPIN_STM32F7xx * pPrv,
                              PTR_ADDR                   BaseAddress);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running.
BaseAddress	BaseAdress of the USB controller.

4.3.6.2 USBC_Reg_VBUS_STM32F7xx()

Description

Register a driver function to determine if VBUS is present.

Prototype

```
void USBC_Reg_VBUS_STM32F7xx(USBC_INSTANCE           * pInst,
                             USBC_PRV_VBUS_STM32F7xx * pPrv,
                             PTR_ADDR                  BaseAddress);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running, and may be destroyed only after `USBC_Exit()` was called.
BaseAddress	BaseAdress of the USB controller.

4.3.6.3 USBC_Reg_CHRDET_STM32F7xx()

Description

Register a driver function to perform charger detection.

Prototype

```
void USBC_Reg_CHRDET_STM32F7xx(USBC_INSTANCE             * pInst,
                               USBC_PRV_CHRDET_STM32F7xx * pPrv,
                               PTR_ADDR                    BaseAddress);
```

Parameters

Parameter	Description
pInst	Pointer to the USBC instance.
pPrv	Pointer to a structure where driver will store its private data. The structure does not need to be initialized before this function is called, but the memory area must be valid as long as the USB-C stack is running.
BaseAddress	BaseAdress of the USB controller.

Chapter 5

Support

5.1 Contacting support

Before contacting support please make sure that you are using the latest version of the emUSB-C package. Also please check the chapter *Configuring debugging output* on page 14 and run your application with enabled debug support.

If you are a registered emUSB-C user there are different ways to contact the emUSB-C support:

1. You can create a support ticket via email to ticket_emusb@segger.com[*]
2. You can create a support ticket at {segger.com/ticket}.

Please include the following information in the email or ticket:

- The emUSB-C version.
- Your emUSB-C license number.
- If you are unsure about the above information you can also use the name of the emUSB-C zip file (which contains the above information).
- A detailed description of the problem
- The configuration files `USBC_Config*.*`
- Any error or debug messages messages.

5.1.1 Where can I find the license number?

The license number is part of the shipped zip file name.
For example `emUSBC_BASE_STM32Gxx_V1.00.0_USBC-01234_95C24726_230614.zip` where USBC-01234 is the license number. The license number is also part of every *.c- and *.h-file header. For example, if you open USBC.h you should find the license number as with the example below:

```
**********************************************************************
*                                                                    *
*       emUSB-C version: V1.00.0                                     *
*                                                                    *
**********************************************************************
----------------------------------------------------------------------
Licensing information
Licensor:                 SEGGER Microcontroller GmbH
Licensed to:              Customer name
Licensed SEGGER software: emUSB-C
License number:           USBC-01234
License model:            SSL
Licensed product:         -
Licensed platform:        Cortex-M, GCC
Licensed number of seats: 1
----------------------------------------------------------------------
Support and Update Agreement (SUA)
SUA period:               2023-05-30 - 2023-11-30
Contact to extend SUA:    sales@segger.com
----------------------------------------------------------------------
```

[*]By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.