FUSB15200 Dual Port USB Type-C/PD Controller Software Programming Guide

UM70103/D

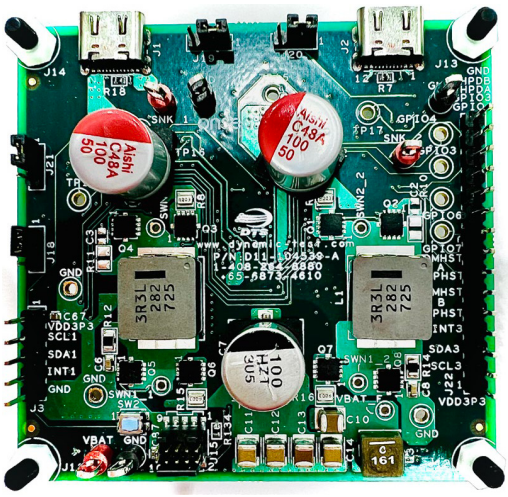


Figure 1. FUSB15200 DUAL PORT 60 W EVB

Introduction

The FUSB15200 firmware codebase is a highly optimized dual−port Type−C/PD controller driver that supports the integrated Arm® Cortex®−M0+ processor. Together with the FUSB15200DV EVB, this driver provides customers with a complete platform for evaluating a Type−C/PD solution.

The firmware provides the flexibility of supporting new power delivery (PD) messages as well as any additional Type−C state flows. The firmware also allows easy modification of the hardware−specific characteristics because of its Type−C/PD platform−agnostic architecture. When supplied with a desired configuration, the codebase can be used to quickly configure the device.

The code organization offers modularity, as it separates source code for application, hardware abstraction layer, platform dependent code, and the USB Type−C/PD core. Default configurations supported by the FUSB15200 Type−C/PD are listed in Table 2. FUSB15200 Supported Configuration in Port.

The PD core features are configurable using project build options or by modifying the vendor info file. The codebase includes a sample Eclipse project that can be compiled using the Eclipse based **onsemi** IDE, thus allowing a faster bring−up to evaluate the Type−C/PD standalone controller.

Supported Power Delivery

Table 1 FUSB15200 Supported Device Characteristics (60 W PDP) summarizes the PD options available on the FUSB15200DV.

Table 1. FUSB15200DV SUPPORTED DEVICE CHARACTERISTICS (60 W PDP)

Feature	Supported Type	Firmware
Type−C	Source/Sink	Yes
PD	DRP	Yes
Advertised PDOs	**PDO Type**	**Description**
PDO 1	Fixed	5 V / 3 A
PDO 2	Fixed	9 V / 3 A
PDO 3	Fixed	15 V / 3 A
PDO 4	Fixed	20 V / 3 A

NOTE: The PDOs supported are power supply dependent.

Port Configuration

Table 2, FUSB15200 Supported Configuration in Port, describes the port configuration of the FUSB15200.

NOTE: The features described in this table are part of the firmware and can be traced in file *vif_info.h*.

Table 2. FUSB15200 SUPPORTED CONFIGURATION IN PORT

Feature	Supported	Description
PD Specification Revision	3.1	Supported Revision
PD Specification Revision Version	1.8	Supported Revision Version
SOP* Communication	Yes	Supports SOP, SOP', SOP"
Manufacturer Info Message	Yes	Manufacturer Info Supported Port
Data Role Swap to DFP	Yes	Supports swap to DFP
Data Role Swap to UFP	Yes	Supports swap to UFP
VCONN Swap to ON	Yes	Support for VCONN swap to ON
VCONN Swap to OFF	Yes	Support for VCONN SWAP to OFF
Cable Discovery	Yes	Supports Cable Query Process
Chunked Message	Yes	Support for Chunked Messages
Long UnChunked Extended Messages	No	Support for Long UnChunked Extended Messages
Rp Value	3 A	CC Pin Current advertisement
VCONN Source	Yes	VCONN sourcing support
PD Power Source	60000 mW	PD port capability
USB Suspend May Clear	No	USB Suspend not supported
Modal Support	No	Disabled modal operation
Unconstrained Power	Yes	Sufficient external source of power is available
Port Type	4	DRP
USB4	Yes	Supported on Port 0

Firmware Build Options

The reference firmware and its default configuration support the FUSB15200 EVB platform for a complete evaluation of the Type−C/PD solution. By following the instructions in section Firmware Build Instructions, a firmware binary can be built and loaded into the EVB. The FUSB15200 default supported values are listed in Table 3 Supported Build Configurations.

Table 3. SUPPORTED BUILD CONFIGURATIONS

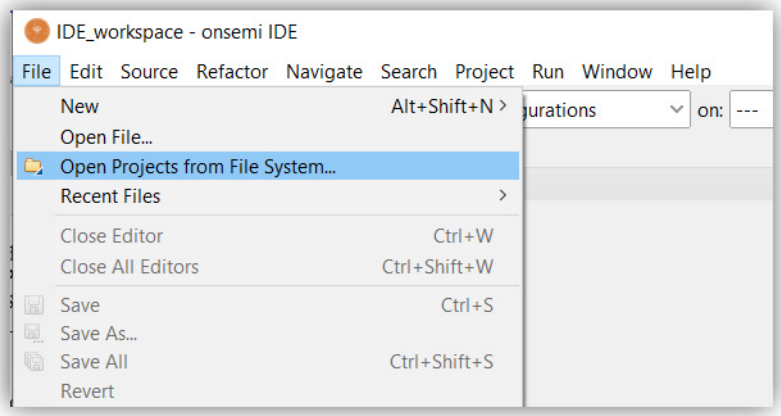
Feature	Supported	Description
CONFIG_BC1P2_CDP	0	Disable support for BC1P2 CDP
CONFIG_BC1P2_CSM	0	Disable support for BC1P2_CSM
CONFIG_BC1P2_DCP	1	Enable support for BC1P2_DCP
CONFIG_BC1P2_DCP_ADDIV	1	Enable support for BC1P2_DCP_ADDIV
CONFIG_DCDC	1	Enable DCDC power supply write via I2C
CONFIG_DRP	1	Enable DRP Support
CONFIG_EPR	0	Disable EPR Support (Unsupported by board)
CONFIG_EPR_TEST	0	Disable EPR_TEST Support (Unsupported by board)
CONFIG_MINIMAL	0	Disable support for CONFIG_MINIMAL
CONFIG_EXTMSG	1	Enable support for extended message length
CONFIG_LEGACY_CHARGING	0	Disable support for legacy charging
CONFIG_LOG	0	Disable support for logging
CONFIG_NOMINAL_PPS_CURRENT	0	Disable support for nominal current
CONFIG_POWER_LIMITED	1	Enable Power Limitation functionality
CONFIG_POWER_SHARING	0	Disable power sharing functionality
CONFIG_SLEEP	0	Enable support for deep sleep
CONFIG_SRC	1	Enable support for source characteristic
CONFIG_USB4	1	USB4 support
CONFIG_VDM	1	Enable support for Vendor Define Message
DEBUG_PORTB	1	Enable Debug functionality
FUSB15200		Define FUSB15200
HAL_USE_ASSERT		Define assertion of size check
I2C3_BOARD		Use I2C3 on board for Power Supply Communication

Firmware Build Instructions

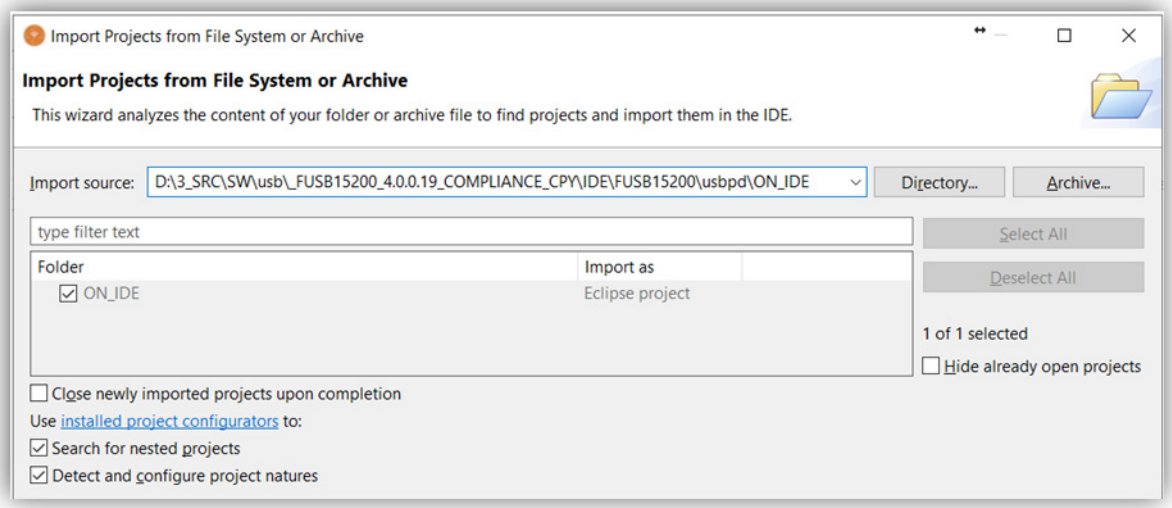
Build the firmware by performing the following steps:

- Download and install the **onsemi** IDE:
  - Click on this link: **onsemi** IDE.
  - Click Design Tools.
  - Click onsemi IDE installer and download it to a location in your system.
  - Follow the prompts to install the **onsemi** IDE.
- Download the 15200 firmware code release:
  - Click on this link.
  - Click FUSB15200 Reference Code and download the zip file.
  - Unzip the contents into a directory of your choice.

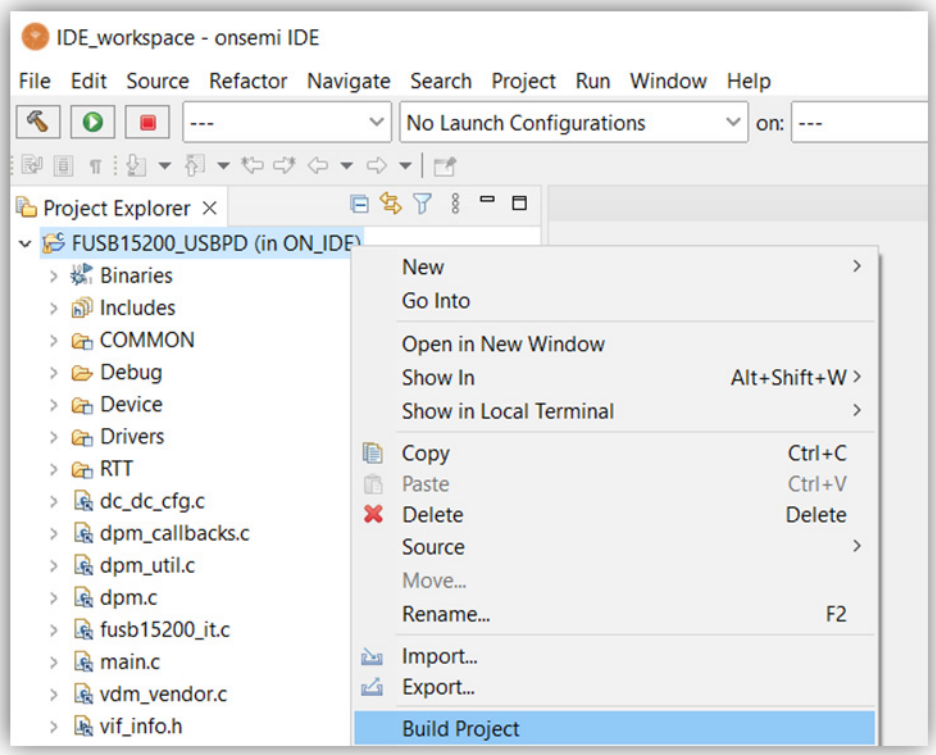
  NOTE: Make sure that the codebase has the directory structure as shown in section Code Organization.
- Open the **onsemi** IDE and load the project:
  - From the top menu, choose **File**.
  - Choose Open Projects from **File System...**.



  - From Import Source, click on Directory... .
  - From the firmware source directory, choose
    Go to fw_fusbdev → IDE → FUSB15200 → usbpd then select ON_IDE



- Build the FUSB15200DV firmware:
  - From the Project Explorer tab, right click on **FUSB15200 USBPD** (in **ON_IDE**), and select **Build Project**.



  - Upon a successful build, the binary FUSB15200 USBPD.bin is copied under *IDE\FUSB15200\usbpd\ON_IDE\Debug*.
  - Refer to the document *FUSB15200 Dual Port USB TYPE−C/PD Controller Flash Programming Guide* for the steps to program the FUSB15200DV.
- Optional: Changing build configuration:
  We recommend that you keep the default build configuration to test the EVB.
  Advanced users, can follow the steps below to change the config parameters listed on *Table 3. Supported Build Configuration*.
  - Press Alt−Enter to display the Properties for FUSB15200 USBPD.
  - Go to C/C++ General > Paths and Symbols > Symbols > GNU C.

Firmware Architecture

This section outlines the firmware architecture of the FUSB15200. Initially it covers in paragraphs Code Organization and Port Configuration a high−level overview of the code. Gradually, it tries to give a more in−depth description in subsequent paragraphs.

Code Organization

Table 4. FUSB15200 Project Subdirectory Descriptions outlines the directory structure and describes the content of each subdirectory.

Table 4. FUSB15200 PROJECT SUBDIRECTORY DESCRIPTIONS

Component	Description
Applications	This component contains custom specific source files including the sample Device Policy Manager. The vendor info file is also in this folder.
CMSIS	
Device	Platform specific source files
Drivers	Hardware abstraction layer source files
External	Type−C/PD state machine and abstraction layer
IDE/FUSB15200/usbpd/ON_IDE	Sample Eclipse based project
SVD	Jlink

Firmware Composition

The FUSB15200 platform integrates an Arm Cortex−M0+ processor with a nested vector interrupt controller, a wakeup interrupt controller, and a debug access port. The codebase includes peripheral drivers and support for multiple external source interrupts for peripheral devices.

- *PD Device Policy Manager*

  The firmware codebase provides reference code for the device policy manager (DPM). The sample DPM can manage platform−specific PD message requests and responses using event subscriptions and notification callbacks. It uses a hardware abstraction layer (HAL) to prevent policy engine or Type−C state machine direct access to hardware registers. The DPM manages a private structure that encapsulates TCPD (Type−C/PD) driver and port structure.

- *PD Policy Engine Core*

  The PD policy engine (PE) state machine is platform−agnostic. Most PE functions are statically defined and are only accessible through the TCPD driver, except for PE state machine core functions, PE state machine enable/disable functions, and a PE hard reset message interrupt handler. The Type−C/PD core in this codebase can support characteristics other than the ones listed in Table 1. FUSB15200 Supported Device Characteristics. These options are configurable, as described earlier in this document.

  Policy engine functions:

  `void policy_pd_enable(struct port *port, bool source)`

  This function enables the PE state machine, used on startup. It can be used in certain contexts with `policy_pd_disable()` when your device cannot offer power delivery.

  `void policy_pd_disable(struct port *port)`

  This function disables the PE state machine. This function is primarily used when the device cannot supply USB Type−C power delivery.

  `void policy_receive_hardreset(struct port*port)`

  This function processes a hard reset when a PD message is received through the PD controller interrupt bit.

  `Void policy_engine(struct port *port)`

  This is the main function for the PE core state machine.

  In the most recent code bases this function changed. In order to save space and avoid an excess of `if, else if` we switched to a function pointer approach. Since a state is a numeric enumerated data that is created at build time by CREATE_XXX_POLICY_STATES() for SRC, SNK, DRP, VDM and USB4 so therefore the state is associated to a numerical index. We used that same index to build a function pointer array whose array element index matches the state enum.

  For example, the index 0 in the `enum policy_state_t` is `PE_SRC_Startup`.

  At index 0 of the array `policy_state_run[]` we placed the function pointer to service that state; which is in this case; `policy_state_source_startup()`.

  IMPORTANT: The lineup of the states and the servicing functions have to match and ought to be aligned. The index alignment must be validated if any new additional state is added.

- *PD PE Message Handling*

  A few PE message handling function examples are listed below. A full list of PD message handlers can be found in *policy.c*. These functions are only accessible from the policy engine.

  `static void policy_state_source_get_sink_cap(struct port *port)`

  This function is called when a PD provider sends a request for sink capability.

`static void policy_state_source_give_sink_cap(struct port *port)`
This function is called when a PD provider responds to a request for sink capability.

`static void policy_state_source_send_drswap(struct port *port)`
This function is used for a PD provider to request `drswap`.

`static void policy_state_source_evaluate_drswap(struct port *port)`
This function is used for a PD provider to evaluate a received `drswap` request.

- *PD PE Message Queuing*
  New to the PE this release is the abstraction of message requests in a 32 bit message queue of bitmasks. By modifying port→msgtx variable you can easily queue supported messages on the FUSB15200. A list of compliance tested messages that are enabled by default on Source and Sink can be found in:

  `policy_reset_message_queue(struct port * port)`
  This function is used to reset the message queue when messages need to be requeued into the message queue. This is called by default on attach and on Hard Reset.

  When setting the msgtx bit for a message, you must also ensure that the bit is cleared. For the default set of enabled messages and some additional tested messages, this bit is cleared upon successful receipt of the message, but other messages must have bit clear behavior defined.
- *PD PE VIF Message Queuing*
  VIF Messages are now handled through the same Message Queuing structure, and has been abstracted into two functions:

  `source_vif_message_requests(struct port * port)`
  This function is used to queue up source vif messages when the vif would demand a message be sent to the port partner.

  `sink_vif_message_requests(struct port * port)`
  This function is used to queue up sink vif messages when the vif would demand a message be sent to the port partner.
- *Type−C State Machine*
  Port detection on attach/detach of a Type−C device is handled inside the USB TypeC state machine function, `typec_sm()`. As with PD, this is also platform−agnostic, and all access to the hardware is controlled by the TCPD driver.
- *Observer Files*
  These files are shared between the device policy manager and the policy engine. *observer.c* contains the function definitions of `event_subscribe`, `event_unsubscribe,` and `event_notify`.
  observer.h has all the declarations of all the event ID and structure definitions necessary for events.

Event Handling

The PD event messaging between DPM and PE is handled with no assumption that DPM subscribes to every notification. This provides flexibility for the DPM to only subscribe to events that are needed for the intended application. It also allows reduction of the binary size.

- *Adding New Events*
  While the events that are already defined might be adequate in the supported platform, if an application requires more event subscriptions/notifications between the policy engine and the device policy manager, additional events can be added as needed. To add a new event, add a definition to the enum type `event_t` in *observer.h*.
- *Registering Event Handlers*
  Event subscription/callback notification is used by the policy engine and the device policy manager to pass on PD message requests/responses and/or platform specific behavior changes to the Type−C/PD controller.

  An event is registered using the function `event_subscribe` following this format:
  `event_subscribe(EVENT_ID, callback_handler)`

  The device policy manager subscribes to applicable events, and the policy engine uses these events to notify the DPM by using the function `event_notify`, following this format:
  `event_notify(EVENT_ID, struct* tcpd_device, void *ctx)`

  Events in Table 5. Supported Events are defined in *observer.h*.
  IDs are declared as an enumerated type and use the ##preprocessor to generate the EVENT with “EVENT_” prepended to each ID in Table 5.

  Example: Event ID “TC_ATTACHED” Generates an event with descriptor “EVENT_TC_ATTACHED”.

Table 5. SUPPORTED EVENTS

Event ID	Description
TC_ATTACHED	Type−C device attached
TC_DETACHED	Type−C device detached
VBUS_REQ	VBUS value request for source
VBUS_SINK	VBUS value request for sink
VCONN_REQ	VCONN request to turn on/off sourcing
PD_DEVICE	PE notify PD device capable
PD_GET_SRC_CAP	PE notify source capability request
PD_GET_SNK_CAP	PE notify sink capability request
PD_GET_EXT_SRC_CAP	PE notify extended source capability request
PD_GET_EXT_SNK_CAP	PE notify extended sink capability request
PD_SNK_CAP_RECEIVED	PE notify sink capability message is received
EXT_SNK_CAP_RECEIVED	PE notify extended sink capability is received
PD_GET_BAT_CAP	PE notify get battery capability request
PD_GET_BAT_STAT	PE notify get battery status request
PD_BAT_CAP_RECEIVED	PE notify battery capability is received
PD_BAT_STAT_RECEIVED	PE notify battery status is received
PD_GET_MAN_INFO	PE notify get manufacturer info request
PD_SRC_EVAL_SNK_REQ	PE notify to evaluate sink request
PD_SNK_EVAL_SRC_CAP	PE notify to evaluate source capability
PD_CBL_ID_RECEIVED	PE notify cable ID is received on cable query
PD_GET_ALERT_REQ	PE notify to fill out alert request
PD_ALERT_RECEIVED	PE notify alert message is received
PPS_STATUS_RECIEVED	PE notify PPS status is received on PPS status request
PPS_STATUS_REQUEST	PE notify PPS status request
PPS_MONITOR	PE notify to activate PPS handling
PPS_ALARM	PE notify to set PPS alarm
ENTER_USB_REQUEST	PE notify when EnterUSB message is being sent
ENTER_USB_RESPONSE	PE notify when a response is received after sending EnterUSB
ENTER_USB_RECEIVED	PE notify when EnterUSB message is received
IDENTITY_RECEIVED	Not used
PD_STATUS	PE notify PD device status
MODE_ENTER_SUCCESS	Not used
MODE_EXIT_SUCCESS	Not used
MODE_VDM_ATTENTION	Not used
HARD_RESET	PE notify hard reset
UNSUPPORTED_ACCESSORY	PE notify for unsupported accessory attached
DEBUG_ACCESSORY	Not used
AUDIO_ACCESSORY	Not used
ILLEGAL_CBL	Not used
BIST_SHARED_TEST_MODE	PE notify BIST shared test
PD_NEW_CONTRACT	PE notify for new PD contract
DATA_RESET_ENTER	Not used

Table 5. SUPPORTED EVENTS (continued)

Event ID	Description
DATA_RESET_EXIT	Not used
PD_GET_FW_ID	PE notify get firmware ID request
PD_FW_INITIATE	PE notify to initiate firmware update
PD_INITIATE_RESP_SENT	PE notify firmware update response was sent
PD_GIVE_REVISION	PE notify to provide revision
PD_GIVE_SOURCE_INFO	PE notify to provide source info
PPS_CL	Event to grab PPS CV (Constant Voltage) or CL (Constant Load) mode

Vendor Info File

Device Vendor information is in file *vif.info.h*. If modifications are needed, follow the steps below:

a. If the information is already available in the header file, you only need to modify the default value there.

b. If the information is not yet defined in the header file, modify the header file by adding the new information to the applicable port, and add the entry to `PORT_VIF_T`, which represents the newly added information in *vif_info.c*.

Examples:

- Changing max current in PDO 4 from 20V/3A to 20V/3.25A in the Port:

  Current PDO values:

```
#define PORT_A_SRC_PDO_VOLTAGE_4            400 // 20000 mV
#define PORT_A_SRC_PDO_MAX_CURRENT_4        300 // 3.00A
```

  New PDO values:

```
#define PORT_A_SRC_PDO_VOLTAGE_4            400 // 20000 mV
#define PORT_A_SRC_PDO_MAX_CURRENT_4        325 // 3.25A
```

- Removing support for chunked extended messages in Port:

  Current value:

```
#define PORT_A_CHUNKING_IMPLEMENTED_SOP     1
```

  New Value:

```
#define PORT_A_CHUNKING_IMPLEMENTED_SOP     0
```

- Adding new entry in the vif_info.h in the Port:

  a. `#define PORT_A_NEW_ENTRY.                1`

  b. Add an entry in `PORT_VIF_T` in *Device/FUSB15200/vif_info.c*.

TCPD Driver – Changes from Previous Versions

The current firmware driver is based on a hardware abstraction layer software design. This design provides abstraction to/from PE and Type−C state machine. Neither PD nor Type−C can directly change the platform−specific behavior. The TCPD driver implements access to the port HAL and other supported peripherals. Differently from before, the code has removed pointer dereferencing to access HAL functions. Instead, driver function permissions are given to files that require it in order to function. This allowed saving a large amount of space and saving several kB of function pointer storage and pointer dereferencing. In addition, several high level interfaces built upon the high level HAL abstraction have been removed to save space.

Old flow example:

PE changes VBUS value for a contract being negotiated: the following logic path would be followed:
PE → port_vbus_src() → FUSBDEV HAL with Drivers Abstracted (port→dev→driv→XXX)
port→dev→driv→set_pd_source()→TCPD HAL Driver with Devices Abstracted (fusb15xxx_XXX)
fusb15xxx_set_pd_source()→HAL Driver with Registers Abstracted (XXX_DRIVER)
TCPORT_DRIVER.pd.Source() → Register level logic

New flow example:

PE changes VBUS value for a contract being negotiated: the following logic path would be followed:
PE() → port_vbus_src() → FUSBDEV HAL Abstraction with Drivers Abstracted (fusbdev_tcpd_XXX)
fusbdev_tcpd_set_pd_source()→TCPD Driver with Devices Abstracted (fusb15xxx_XXX)
fusb15xxx_set_pd_source()→HAL Driver with Registers Abstracted (XXX_DRIVER)
TCPORT_DRIVER.pd.Source() → Register level logic

ADDITIONAL INFORMATION

TECHNICAL PUBLICATIONS:
Technical Library: www.onsemi.com/design/resources/technical−documentation
onsemi Website: www.onsemi.com

ONLINE SUPPORT: www.onsemi.com/support
For additional information, please contact your local Sales Representative at www.onsemi.com/support/sales