Warsaw University of Technology

FACULTY OF
ELECTRONICS AND INFORMATION TECHNOLOGY

Institute of Computer Science

Bachelor's diploma thesis

in the field of study Computer Science
and specialisation Computer Systems and Networks

USB Power Delivery protocol firmware for a Sink device

Taras Bolzhelarskyi
student record book number 324018

thesis supervisor
Grzegorz Mazur

WARSAW 2025

USB Power Delivery protocol firmware for a Sink device

Abstract. Specyfikacja USB Power Delivery standaryzuje sposób, w jaki urządzenia Type-C, peryferia i zasilacze o różnych wymaganiach oraz możliwościach pobierają i dostarczają energię. Definiuje protokół umożliwiający komunikację możliwości oraz negocjację kontraktów zasilania, zgodnie z którymi energia jest przekazywana w ramach standardowych lub niestandardowo zdefiniowanych zakresów. W szczególności rozszerzenie Programmable Power Supply (PPS) pozwala urządzeniom-odbiornikom (sink) na precyzyjną kontrolę napięcia wyjściowego i limitu prądu w dostępnych zakresach. Istniejące otwarte implementacje tego protokołu często stosują nieefektywne podejścia: są ściśle powiązane z określonymi stosami programowymi, wykorzystują projekty niepraktyczne w systemach o ograniczonych zasobach i cechują się znacznym zużyciem pamięci. Niniejsza praca proponuje projekt firmware'u mikrokontrolera implementujący protokół zgodnie ze specyfikacją PD 3.0, w zakresie obejmującym Fixed PDO, PPS APDO, obsługę soft/hard reset oraz sekwencję attach/detach. Implementacja wykorzystuje podejście sterowane zdarzeniami, korzystając z preempcji sprzętowych przerwań do obsługi zdarzeń zachodzących podczas pracy protokołu. Przedstawione wyniki pokazują praktyczność tego rozwiązania oraz jego zdolność do integracji z istniejącymi implementacjami programowymi.

Keywords: USB PD, Power Delivery, STM32, C, mikrokontrolery, przerwania, oprogramowanie sterowane zdarzeniami

USB Power Delivery protocol firmware for a Sink device

Abstract. The USB Power Delivery specification standardizes the way Type-C devices, peripherals, and power supplies with different requirements and capabilities consume and provide power. It defines a protocol that allows communication of capabilities and negotiation of power contracts, according to which power from standard or custom set of ranges will be transferred. In particular, its Programmable Power Supply (PPS) extension allows sink devices to get granular control of output voltage and current limit in the available range. Existing open implementations of this protocol often use inefficient approaches that are either coupled with specific software stacks, use designs that are not practical to interact with in a constrained system and carry sizable memory footprints. This thesis proposes a microcontroller firmware solution that implements the protocol guided by PD 3.0 specification with Fixed PDOs, PPS APDOs, soft/hard reset handling and attach/detach sequencing in its scope. This implementation uses an event-driven design approach leveraging hardware interrupt preemption to handle events that occur during protocol operation. The results presented demonstrate practicality of this solution and its ability to be integrated into existing software implementations.

Keywords: USB PD, Power Delivery, STM32, C, Microcontrollers, Interrupts, Event-Driven Software

załącznik nr 10 do zarządzenia
nr 46 /2016 Rektora PW

Politechnika Warszawska
Warsaw University of Technology

……….......................
miejscowość i data
place and date

……………………………..
imię i nazwisko studenta
name and surname of the student

……………………………..
numer albumu
student record book number

……………………….……….
kierunek studiów
field of study

OŚWIADCZENIE

DECLARATION

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.
Under the penalty of perjury, I hereby certify that I wrote my diploma thesis on my own, under the guidance of the thesis supervisor.

Jednocześnie oświadczam, że:
I also declare that:

– niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
– *this diploma thesis does not constitute infringement of copyright following the act of 4 February 1994 on copyright and related rights (Journal of Acts of 2006 no. 90, item 631 with further amendments) or personal rights protected under the civil law,*

– niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
– *the diploma thesis does not contain data or information acquired in an illegal way,*

– niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
– *the diploma thesis has never been the basis of any other official proceedings leading to the award of diplomas or professional degrees,*

– wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
– *all information included in the diploma thesis, derived from printed and electronic sources, has been documented with relevant references in the literature section,*

– znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.
– *I am aware of the regulations at Warsaw University of Technology on management of copyright and related rights, industrial property rights and commercialisation.*

1 / 2

Politechnika Warszawska
Warsaw University of Technology

załącznik nr 10 do zarządzenia
nr /2016 Rektora PW

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.
I certify that the content of the printed version of the diploma thesis, the content of the electronic version of the diploma thesis (on a CD) and the content of the diploma thesis in the Archive of Diploma Theses (APD module) of the USOS system are identical.

....
czytelny podpis studenta
legible signature of the student

Contents

1. Background

In modern electronics, the Universal Serial Bus (USB) has evolved from a simple data interface into a ubiquitous power and data delivery ecosystem. As device complexity and power demands grew, from smartphones to laptops and beyond, the initial power capabilities of USB became a significant limitation. This led to a fragmented market of proprietary fast-charging technologies, where chargers and devices from different manufacturers were often incompatible, creating consumer frustration and contributing to electronic waste.

The introduction of the USB Type-C connector marked a pivotal moment, offering a more robust, reversible physical interface designed to handle higher power levels. However, the connector itself was only part of the solution. To safely and efficiently manage power levels far exceeding the original USB specification, a sophisticated communication protocol was required. This is the role of USB Power Delivery (PD).

USB Power Delivery (PD) is a standardized protocol for negotiated power over USB that enables devices to establish explicit contracts for voltage and current beyond the default Type-C current at 5 V. PD defines roles (Source and Sink), a reliable message exchange on the Configuration Channel, and state machines that sequence discovery, negotiation, power transitions, and recovery. Its purpose is to provide interoperable and safe delivery of higher power levels across diverse device classes, replacing fragmented charging approaches that predated Type-C. PD operates independently of USB data transfer; it uses the CC link for signaling and authorizes increases in VBUS voltage or current only after an explicit contract is agreed, with defined mechanisms for acknowledgment, retries, and resets [1], [2].

1.1. Context and motivation

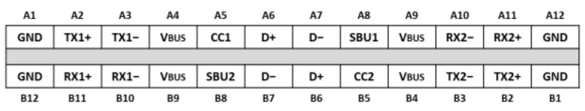
The evolution of charging over USB has been driven by the ever-increasing power demands of portable electronics. Early USB standards (1.0/2.0) provided a mere 2.5 W (5 V at 500 mA), sufficient for low-power peripherals but inadequate for charging large batteries. The USB Battery Charging (BC) 1.2 specification [3] improved this by allowing up to 7.5 W (5 V at 1.5 A) through dedicated charging ports on the legacy USB-A connector, but this was still insufficient for rapidly charging modern devices.

To overcome this limitation, device manufacturers developed a variety of proprietary fast-charging protocols. Technologies such as Qualcomm's Quick Charge [4], [5], Samsung's Adaptive Fast Charging [6], [7], OPPO's VOOC [8], [9], and Huawei's SuperCharge [10], [11] emerged. These systems often achieved higher power by manipulating voltage levels on the D+/D- data lines of the USB-A or Micro-USB connectors [12]. While effective, this created a highly fragmented ecosystem. A charger supporting one standard was typically unable to fast-charge a device supporting another, forcing consumers to use specific charger-device pairings to achieve optimal speeds. This lack of interoperability not only caused confusion but also contributed significantly to e-waste, as users accumulated incompatible chargers.

The introduction of the USB Type-C connector provided the physical foundation to solve this problem. Its 24-pin design, symmetric form-factor, and dedicated Configuration Channel (CC) pins were built to support higher power and more complex communication [1]. However, it was the USB Power Delivery (PD) protocol that provided the intelligence. PD moved power negotiation from voltage manipulation on data lines to a dedicated, packet-based communication protocol over the CC line. This created a single, unified, and interoperable standard for advanced power negotiation, capable of managing power contracts up to 240 W. The primary motivation behind PD was therefore to replace the fragmented landscape of proprietary solutions with a universal standard that ensures safety, interoperability, and flexibility across all vendors and device types [2], a goal that aligns with recent regulatory efforts to standardize charging solutions [Find ref later maybe].

1.2. Type-C [1]

The USB Power Delivery (PD) specification is tightly coupled with the USB Type-C specification and since revision 3.0 [2] supports only Type-C devices. A Type-C receptacle and plug both have 24 pins; for PD, the relevant ones are GND, VBUS, and the two Configuration Channel pins CC1/CC2 (Fig. 1.1). During operation only one CC pin carries PD signaling, while the other may serve as VCONN, the power source for an electronically marked cable if present. Devices assert specific resistances on the CC pins to signal their roles (Sink or Source) and to avoid collisions (SinkTxOk resistance is asserted by the Source to tell Sink that it is permissible to initiate message sequences). Rd is used by the Sink; Rp encodes the Source's default current capability at 5 V. The Sink reads this value before establishing a PD contract.



A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12
GND	TX1+	TX1−	VBUS	CC1	D+	D−	SBU1	VBUS	RX2−	RX2+	GND
GND	RX1+	RX1−	VBUS	SBU2	D−	D+	CC2	VBUS	TX2−	TX2+	GND
B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1

Figure 1.1. USB Type-C Receptacle Interface (Front View)

The Type-C specification defines attach and detach through debounced CC voltage windows and VBUS presence. Attach occurs when Rp is present on exactly one CC pin for at least $t_{CCDebounce}$ and VBUS is detected. Detach occurs when both CC pins are open for at least $t_{PDDebounce}$. Upon entry to vSafe5V (the Source has applied 5 V), a Sink may draw only the CC-advertised current until a PD contract is established. Higher voltages or currents require an explicit PD contract.

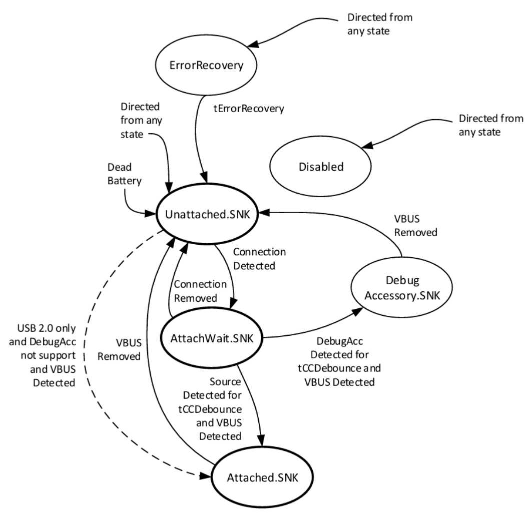


Figure 1.2. Connection State Diagram: Sink

1.3. Roles terminology

Two power roles are defined: Source provides power on VBUS; Sink consumes power. Data role is orthogonal (DFP/UFP) but is not required for power-only operation. Devices may be dual-role power (DRP) capable, alternating Rp/Rd on CC to discover a partner; as will be stated further, this work focuses on Sink behavior. Communication is addressed to SOP* tokens (SOP for the partner port; SOP'/SOP" for cable plugs) [1], [2].

1.4. PD protocol and message format

Every PD message begins with a 16-bit Message Header that encodes Message Type, Number of Data Objects (NDO), MessageID, Spec Revision, Power Role, Data Role, and the Extended bit. The header is followed by zero or more 32-bit Data Objects. Messages with header only are Control Messages; messages with one to seven data objects are Data Messages; messages marked Extended include an Extended Header and a variable-length payload segmented into chunks in PD 3.0 and later (formats illustrated in Figs. 1.4 and 1.5). Addressing uses SOP* tokens for the partner port and cable plugs as shown in Fig. 1.3 [2].

Delivery uses a positive acknowledgment scheme. Each message must be acknowledged by a GoodCRC response carrying the same MessageID (example exchange in Fig. 1.6). If GoodCRC is not received, the sender retries a limited number of times and may initiate a Soft Reset on failure.

1.5. Power models and ranges

PD defines operating points in terms of voltage and current limits communicated as Power Data Objects (PDOs) from Source to Sink and Request Data Objects (RDOs) from

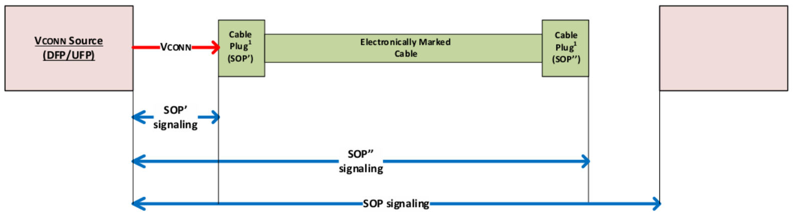


Figure 1.3. SOP* communication structure

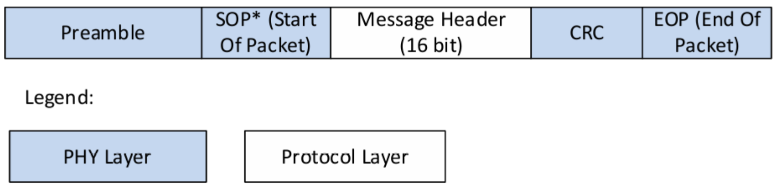


Figure 1.4. Control message format

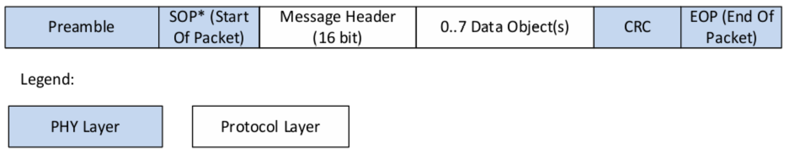


Figure 1.5. Data message format

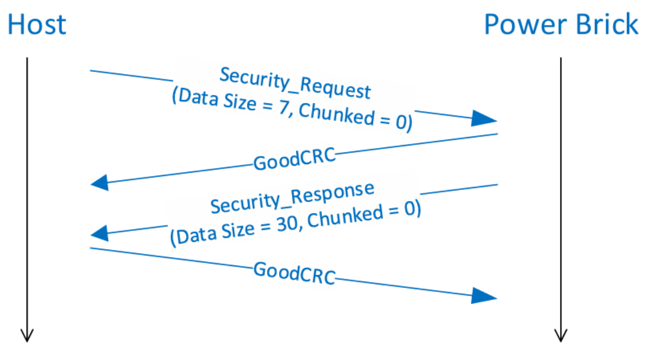


Figure 1.6. Example message exchange with GoodCRC response

Sink to Source. Negotiation establishes an explicit contract that authorizes drawing power above the default Type-C current at 5 V [2].

1.5.1. SPR (Standard Power Range)

SPR covers operation up to 100 W and is expressed through Fixed PDOs and optionally Battery or Variable PDOs. Fixed PDOs advertise discrete voltages; the default current limits are 3 A with up to 5 A (only at 20 V) permitted when an electronically marked cable is present. When needed, the Source checks the cable's capability.

Table 1.1. SPR Fixed Voltage Operating Points

Voltage (V)	Typical Max Current (A)	Power (W)
5	up to 3	up to 25
9	up to 3	up to 27
15	up to 3	up to 45
20	up to 3 (5 with e-marked cable)	up to 60 (or 100)

A Sink selects one advertised PDO by sending an RDO that identifies the object position and encodes the operating current and the limit current together with auxiliary flags. Upon receiving Accept followed by PS_RDY, an explicit contract is in place.

1.5.2. PPS (Programmable Power Supply)

PPS augments SPR with continuous voltage selection within a defined range. Sources advertise one or more PPS APDOs (Augmented PDOs) that specify a minimum and maximum voltage, and a maximum current capability. The Sink requests a specific output voltage in 20 mV steps and an operating current in 50 mA steps using a PPS RDO. During a PPS contract the Sink must periodically send a keep-alive request to preserve the contract and may request a new voltage level or current limit during this exchange [2].

Table 1.2. PPS APDO Examples

Max Current (A)	Min Voltage (V)	Max Voltage (V)	Notes
3	3.3	11	Common range
3	3.3	16	Common range
3	3.3	21	Common range
5	3.3	21	Requires cable with 5 A capability

1.5.3. EPR (Extended Power Range)

PD 3.1 introduced EPR [2]to extend the maximum power to 240 W. EPR adds new Fixed EPR voltages and adjustable operation subject to additional cable requirements and transition sequencing. EPR operation requires EPR-capable Source, Sink, and cable (EPR cable with appropriate e-marker). Entry into EPR follows a defined sequence starting from an SPR contract(Figure 1.7), with capability advertisement indicating EPR support, negotiation of EPR operating points, and Source confirmation before voltage is raised beyond SPR limits. Exit from EPR requires an orderly transition back to SPR or to vSafe5V and is triggered by error conditions, detach, or policy. EPR imposes additional constraints on cable and connector parameters to ensure safety and reliability at higher voltages. The fixed operating points defined for EPR can be found in Table 1.3 .

Table 1.3. EPR Fixed Voltage Operating Points

Voltage (V)	Max Current (A)	Power (W)
28	up to 5	up to 140
36	up to 5	up to 180
48	up to 5	up to 240

It should be noted that even in EPR mode the port partners can establish SPR contracts and standard power ranges.

1.6. Common message types

Control Messages: GoodCRC acknowledges receipt of a message with a matching MessageID: the ID of the received GoodCRC must match the ID of the transmitted message; Accept indicates that a request or proposal was accepted and the sequence may proceed, used during power negotiation and Soft Reset; Reject indicates the request for power cannot be fulfilled by the Source; Soft Reset reinitializes protocol-layer counters and state without changing the power contract; PS_RDY indicates that the Source has completed a power transition and that the new power state is ready; Get Source Capabilities asks the Source to advertise its PDOs/APDOs; Get Sink Capabilities asks the Sink to advertise its PDOs/constraints; Get PPS Status requests live status during a PPS contract (e.g., output voltage/current and flags). Data Messages: Source Capabilities advertises available PDOs/APDOs from the Source; Request (RDO or PPS RDO) selects an advertised capability and specifies current and, for PPS, requested voltage; Sink Capabilities advertises the Sink's operating points and constraints; PPS Status reports Source measurements/flags defined for PPS operation. Not every message support is mandatory; availability depends on supported features defined by the specification [2].

1.7. Protocol timers and counters

The specification defines normative timers and retry counters that bound protocol behavior and ensure progress [2]. Examples include: $t_{SenderResponse}$, the maximum time to respond to a received message within an Atomic Message Sequence; $t_{Receive}$, the time window for expecting a GoodCRC message as a response to transmitted message; $t_{PSTransition}$ for bounding power-supply transition time prior to receiving PS_RDY message indicating that the new power level is provided; $t_{CCDebounce}$ and $t_{PDDebounce}$ inherited from Type-C for attach debouncing and detach detach debouncing - when resistance on CC line disappears, the corresponding timer is started to confirm if the CC line is disconnected. MessageID counter is maintained per direction and incremented modulo 8 (PD 3.0), with a defined retry limit after which Soft Reset procedure is initiated. This counter tracks the ID of the incoming messages to handle repeated message reception. Retry counter tracks the amount of times the Protocol Layer tried to resend the message that has failed before due to wrong busy line or incorrect MessageID in the GoodCRC. These timers and counters structure the sequencing and recovery of the PRL and PE.

1.8. Contracts and atomic message sequences

Message exchanges are conducted as Atomic Message Sequences that define scenarios by which partners establish a contract, exchange information, or change roles. Each transmitted message must be followed by a GoodCRC response. Absent GoodCRC after the retry limit, a Soft Reset is issued to recover.

1.8.1. SPR (establishing a fixed-voltage contract)

1. Source sends Source Capabilities.
2. Sink chooses PDO number N and sends Request (RDO) with operating and limit current.
3. Source sends Accept.
4. Source sends PS_RDY.

1.8.2. PPS (establishing a programmable-voltage contract)

1. Source sends Source Capabilities including PPS APDO.
2. Sink sends Request (PPS RDO) with desired voltage and current.
3. Source sends Accept.
4. Source sends PS_RDY.
5. Sink periodically sends out a keep-alive Request to preserve the contract.
6. Sink may later adjust voltage by sending another PPS RDO and query live values with Get PPS Status / PPS Status.

1.8.3. EPR (establishing an extended-power contract)

Entry from SPR to an EPR contract

1. Preconditions: EPR-capable Source and Sink are in an SPR explicit contract; EPR-capable cable present (e-marker indicates EPR).
2. Sink sends EPR_Mode (Enter) message to Source.
3. Source responds with EPR_Mode (Enter Acknowledge) message.
4. Source sends EPR_Mode (Enter Succeeded) message

Negotiating EPR AVS voltage/current during operation

1. Source sends its EPR_Source_Capabilities message with PDOs containing extended ranges of voltages.
2. Sink replies with EPR_Request message.
3. Source follows with Accept and PS_RDY - identical to SPR contract procedure.

Keep-alive while in EPR AVS

1. Within the normative keep-alive interval defined by the specification, the Sink periodically sends a EPR_KeepAlive to maintain the contract.
2. Source replies with EPR_KeepAlive_Ack.

Exit from EPR

1. Sink sends a EPR_Mode (Exit) message.
2. Same sequence as with SPR explicit contract negotiation.

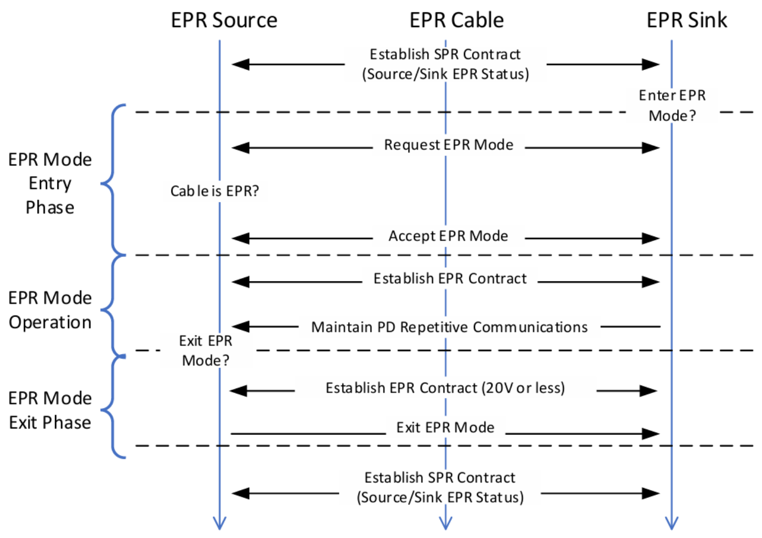


Figure 1.7. Example of a Normal EPR Mode Operational Flow

If policy or protocol errors (1.9) occur, port partners use Soft Reset or Hard Reset to recover, as defined by the specification.

1.9. Resets

Soft Reset resets protocol-layer state such as message counters and timers without tearing down the power contract. Hard Reset is signaled electrically over CC by the Source or Sink; it returns VBUS to vSafe0V and then restarts at 5 V, after which negotiation can begin anew [2]. Common reasons for Soft or Hard reset are: reception of GoodCRC with incorrect MessageID, failure to send message after retries, failure to send Soft Reset message, reception of SPR capabilities in EPR mode and the other way around, etc. Specific cases and conditions for entering either mode are diverse and can be quite convoluted so please refer to [2] for details. In general Soft Reset is used to fix a minor protocol error like incorrect message ID or unexpected message being received. Hard Reset fully resets the stack in order to fix a bigger error like missing answer to a message after a timeout.

1.10. Architecture outline

The specification describes a logical layering of responsibilities rather than a strict implementation prescription (Fig. 1.8). The Device Policy Manager (DPM) represents system policy outside the protocol. The Policy Engine (PE) embodies protocol state machines and atomic sequences. The Protocol Layer (PRL) provides reliable delivery semantics (MessageID, GoodCRC, retries, extended-message chunking). The Physical Layer (PHY) represents electrical and coding aspects on the CC line. These components are conceptual; concrete implementations may organize software differently while preserving these responsibilities [2].

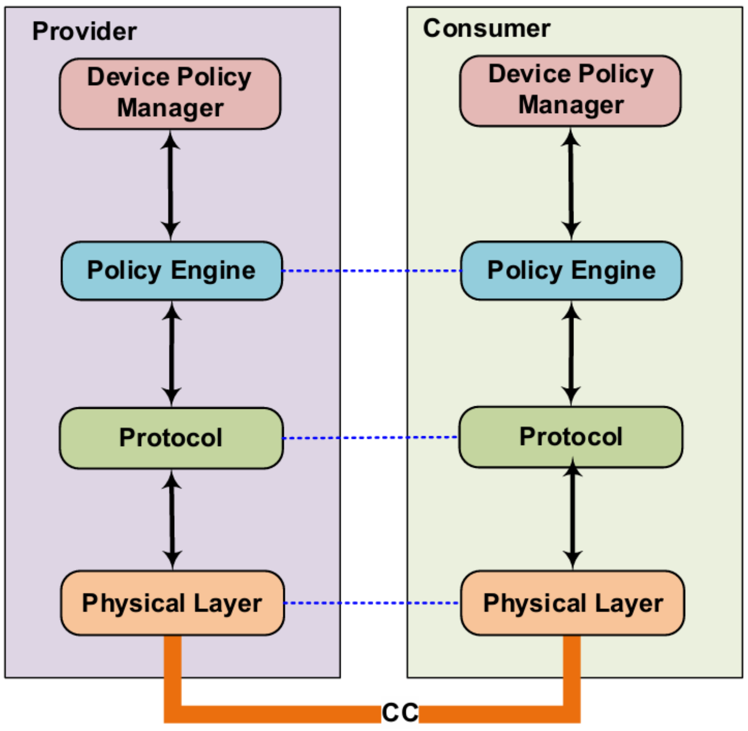


Figure 1.8. General view of PD architecture [2]

1.10.1. State machines (conceptual)

The specification defines behavioral state machines for the Policy Engine (PE) and the Protocol Layer (PRL). These describe normative behavior and sequencing, not implementation prescriptions; readers should refer to the specification for the official PE and PRL diagrams [2].

The PE sequences contract establishment and maintenance. After attach and entry to vSafe5V, the Sink PE awaits Source Capabilities within the specified response time; if not received, it may solicit them with Get Source Capabilities. Upon receiving capabilities, the PE evaluates advertised PDOs or APDOs against the Sink's constraints and policy and constructs a Request. The PE then transmits the Request and waits for Accept. If Accept is received, the PE awaits PS_RDY within the power-transition time to confirm that the new power state is ready. If Reject is received, the PE may choose an alternative operating point or remain at the current level. Absent required responses within normative timers, or upon receipt of unexpected or unsupported messages, the PE initiates recovery via Soft Reset or, if required, handles or initiates a Hard Reset as defined by the specification.

For PPS maintenance, the PE periodically refreshes the contract by sending a PPS Request to keep the supply active; at these times it may adjust the requested voltage or current. It may also query the Source using Get PPS Status and process the returned PPS Status to monitor output conditions and flags.

The PRL provides reliable delivery semantics over the CC link. On transmit, the PRL builds the header and any data objects, emits the appropriate SOP* ordered set, and waits for a GoodCRC carrying the matching MessageID. Success increments the MessageID modulo eight; timeout triggers a retry up to the normative retry limit, after which the PRL reports an error to the PE. On receive, the PRL detects SOP*, captures the message, verifies CRC, responds with GoodCRC using the received MessageID, and delivers the message to the PE. Duplicate or out-of-sequence MessageIDs are handled as specified to preserve exactly-once delivery semantics within an Atomic Message Sequence. For extended messages, the PRL manages the Extended Header and chunking protocol when supported, and follows the defined behavior and timers when chunking is not supported.

Soft Reset reinitializes protocol-layer state including MessageID counters and retry context and returns the PRL to the idle state without affecting the power contract. Hard Reset tears down power to vSafe0V and restarts at 5 V; its detection and signaling are handled at the electrical level, with the PRL and PE coordinating reentry to discovery after completion. Transitions between these states are bounded by normative timers such as $t_{SenderResponse}$, $t_{Receive}$, and $t_{PSTransition}$, as defined in the specification.

2. Related work

Publicly available USB Power Delivery implementations generally follow either a main-loop (polling) design or an RTOS-centric design. Both styles can pose integration challenges in heterogeneous projects: interfaces and build systems are often tailored to specific toolchains or evaluation boards, licenses may restrict modification or redistribution, and portability layers are frequently optimized for a narrow set of platforms.

2.1. STMicroelectronics

For the target microcontroller family in this work, STMicroelectronics provides a PD software stack [13] intended for use on the vendor's boards. The stack can be configured to operate atop an RTOS or a simple event loop. In both configurations, hardware interrupts primarily serve as wake-up or unblocking signals, while the substantive protocol processing executes within RTOS tasks or within the body of the central loop.

This execution model imposes a centralized control flow: either an RTOS scheduler or an indefinitely executing loop remains in control of progression, and application code is integrated as a task or as per-iteration callbacks. While this arrangement simplifies certain portability and middleware concerns, it constrains designs that favor direct, in-interrupt handling of protocol events or that aim to minimize scheduler involvement for latency and energy reasons. In practice, adopting such a stack requires aligning the application architecture with the provided scheduling paradigm rather than treating the PD layer as a standalone component of the software. Please refer to 5.2 for excerpt from implementation code of the event loop variant.

Moreover, the Policy Engine (PE) and Protocol Layer (PRL) are distributed only in a form of a precompiled library [13] with no source code available. Although several variants are offered with different feature sets enabled, fine-grained control is not possible. In the open-source portion, features can be disabled via preprocessor directives; however, the stack's extensive layering, convoluted codebase and the dispersion of user-implemented hooks across the project structure complicate integration. The software is clearly designed to be primarily configured through graphical interface that is part of STM32CubeIDE [14] software.

2.2. Google ChromeOS

The project offers two implementations: a legacy stack that supports devices beyond Chromebook products and a newer stack restricted to Chromebooks only [15]. The maintainers note that the legacy codebase "has aged … has grown to the point where it is difficult to add new features and address bugs" [16]. Documentation for both variants is limited, which complicates adoption and maintenance. The documentation further states: "the PD_C task runs the state machine (old or new) for the port and communicates with the TCPC, MUX, and PPC. This task needs a large task stack" [15]. This description indicates an RTOS-oriented architecture with substantial per-task stack requirements, which may be ill-suited to the tight RAM budgets typical of small microcontrollers.

3. Requirements

This section describes the requirements for the project. Its goal is to implement a software library that implements the Power Delivery protocol and can be used in microcontroller firmware so it can function as a Sink device.

3.1. Functional Requirements

- **Lifecycle** The library shall provide an explicit lifecycle: initialize with a Policy callback and deinitialize.
- **Multi-port** The library shall support operation of multiple independend Sink ports.
- **Capabilities access (API)** The application shall be able to query the partner's advertised capabilities (Source PDO/APDO list) via the librarie's API.
- **Request power (API)** The application shall be able to request power via the API:
  - — *SPR (FixedSupply)*: request by object index with operating current and limit current.
  - — *PPS*: request by PPS APDO index with target voltage and maximum current limit.
  - — *EPR (Fixed only)*: request by EPR Fixed PDO index with operating current and limit current.
- **Policy on init** A policy callback is provided at initialization. All selection and decision logic resides in policy; the library performs no automatic selection nor range clamping beyond basic argument integrity.
- **Policy events** The library shall notify the policy of significant events (e.g., capabilities received, general message received, request for information, errors) and the port number; the policy can access the state of the global port object through API and modify it to provide required data.
- **Synchronous/asynchronous requests** The API shall provide synchronous and asynchronous variants for power requests. These requests either block execution until operation is finished or accept a callback that will be invoked when operation is finished with status as argument, the API procedure itself is non-blocking and returns immediately.
- **Optional debug tracing** A compile-time toggle shall enable/disable debug tracing intended only for development; tracing is excluded from release builds. It uses primarily compile time strings and a ring buffer with constant pointers to those strings to transmit them over the USART connection.

3.2. Non-functional Requirements

- **Concurrency model** Operation is fully asynchronous and interrupt-driven. No RTOS and no central event loop are required.

- **Memory footprint** The RAM required for library-managed buffers and flags is on the order of a few hundred bytes. The code size is on the order of several kilobytes.
- **Primary Target Platform** The primary target for the software is STMicroelectronics STM32G0B1x [17] family of microcontrollers with support for other options provisioned for future.
- **Separation from hardware** Device-specific code is kept separate from the protocol stack code. The stack itself contains no device-specific logic, but no clear hardware abstraction API is provided apart from set of procedures that hide the invocation of device specific code.
- **Configuration** All protocol timing parameters required by the specification and scope of feature set are implemented - no need for Source specific timers in a Sink device; their values are compile-time constants with the ability to override by changing the source code.
- **Language** The public API is provided in the C programming language.

3.3. Protocol Scope

- **Specification level** USB Power Delivery 3.1 with EPR support limited to Fixed EPR operating points is implemented by the protocol stack. AVS/adjustable EPR operation is out of scope.
- **Roles** Sink-only. No data-role, power-role, or VCONN swaps. No Fast Role Swap.
- **Addressing** SOP only. Cable SOP'/SOP" interactions are out of scope for the sink library. Those interactions are relevant only to Source devices as they are the ones that have to check for presence of proper e-marked cable that is able to support EPR voltage ranges and currents.
- **Discovery and negotiation**
  - Receive Source_Capabilities message as integral part to establishing explicit contracts; optionally transmit Get Source_Capabilities message to query the capabilities of connected Source device.
  - Transmit Request query for SPR Fixed PDOs, PPS APDOs, and EPR Fixed PDOs, as evaluated and selected by Policy.
  - Receive Accept/Reject and PS_RDY messages as signs of successful contract negotiation.
- **Control and recovery**
  - GoodCRC handling in the protocol layer per specification.
  - Soft Reset and Hard Reset handling per specification. The Policy Engine should react to errors in Protocol layer and issue a Soft Reset message to fix the error or issue a Hard Reset for complete power reset and clean initialization, still without physical CC line connection severance.
- **Optional messages**

  - Optionally support Sink_Capabilities message exposure at request of the Source device.
  - Optionally support Get PPS Status / PPS Status messages for more flexible handling of PPS operation mode.

- **Out of scope** Vendor Defined Messages (VDMs), role swaps, and any data-path features. They are not normative for SPR, PPS and EPR operation and are left for future development.
- **Timers and counters** All normative timers and counters relevant to the scope of implemented features.

3.4. Verification Targets

- Correct handling of attachment and detachment during operation. Proper initialization on attach; reset and PE/PRL shutdown on detach.
- Successful operation when attached to sources advertising SPR, PPS, and Fixed EPR PDOs, including establishing and maintaining explicit contracts as requested by policy.
- Correct handling of Soft Reset and Hard Reset sequences per specification.

4. Hardware and software platform

This section describes the hardware and software platform used to implement the project defined in Section 3, using the context established in Section 1. The target micro-controller is STM32G0B1RE on the NUCLEO-G0B1RE board (Fig. 4.1). For the purposes of this project, among other features, the device provides the on-chip UCPD peripheral, ADC, general-purpose timers, DMA, and a USART for tracing, and it operates with the X-NUCLEO-SNK1M1 USB Type-C PD Sink expansion board(Fig. 4.2).

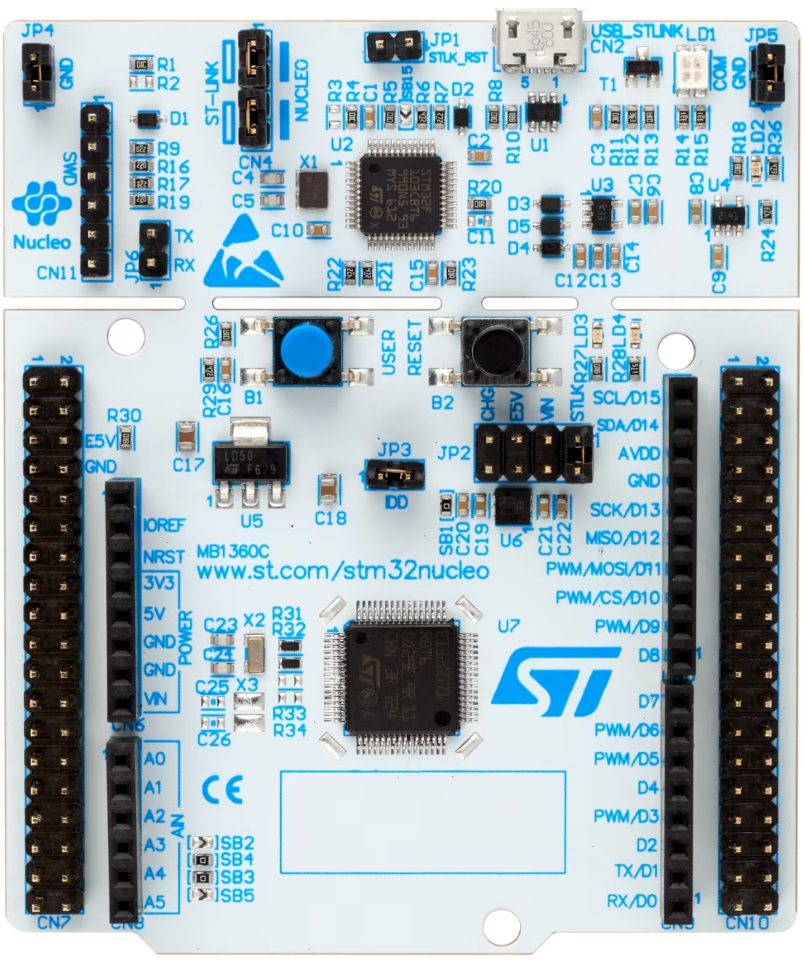


Figure 4.1. STM32G0B1RE development board [18]

4.1. HAL and LL driver layers

The vendor supplies two software layers for peripheral access in the STM32CubeG0 package [19]: the Hardware Abstraction Layer (HAL) and the Low-Layer (LL) drivers. HAL provides feature-oriented, portable APIs with handle structures, initialization helpers, and callback hooks; LL provides thin, register-near primitives as static inline functions that map closely to the reference manual fields. Both layers are generated and distributed in the STM32CubeG0 package and are documented in [20]; their symbols and options are aligned with the device reference manual and the CMSIS-Core [17], [21]. definitions.

4.1.1. HAL programming model

HAL is organized by peripheral and exposes initialization functions that fill a handle, configure clocks and GPIO through msp hooks, and program the peripheral registers. For data transfer it offers blocking calls with timeouts, interrupt-driven calls that return immediately and complete in the IRQ context via callbacks, and DMA-driven calls that complete on DMA events with completion callbacks. Error and state reporting use status codes and state fields in the handle; timeouts are based on the HAL time base. Typical operations include HAL_UART_Init, HAL_UART_Transmit_DMA, HAL_ADC_Start_DMA, HAL_TIM_Base_Start_IT, and the corresponding deinitialization and callback functions [20].

4.1.2. LL programming model

LL drivers provide per-bit and per-field accessors for control and status, following the register names in the reference manual and the CMSIS device header. Programming sequences follow the reference manual order: enable the peripheral clock, configure the peripheral via LL setters, route DMA requests if needed, configure and enable interrupts in the peripheral, and then enable the peripheral. Functions follow a consistent pattern such as LL_USART_Enable, LL_USART_SetBaudRate, LL_DMA_EnableChannel, LL_ADC_SetChannelSamplingTime, LL_TIM_EnableIT_UPDATE, and can be called from interrupt context with minimal overhead [20].

4.1.3. Mixing layers and scope

HAL and LL may be used together so that high-level initialization and noncritical paths use HAL while critical paths, tight ISRs, or footprint-sensitive code use LL. The peripheral coverage spans the devices used here, including GPIO, EXTI, DMA, timers, ADC, USART, and the UCPD peripheral; device-level enable bits, status flags, and interrupt masks remain accessible through LL even when HAL is present.

4.1.4. Interrupt and DMA integration

Both layers provide interrupt and DMA integration points. HAL exposes interrupt-mode and DMA-mode entry points with weak callback symbols for completion, half-transfer, error, and peripheral-specific events; NVIC and DMAMUX setup may be done by HAL msp hooks or manually. LL exposes fine-grained enable and flag operations for peripheral interrupts and DMA request lines, and leaves NVIC priority and pending control to CMSIS-Core functions. This allows using CMSIS for NVIC_EnableIRQ, NVIC_SetPriority, and NVIC_SetPendingIRQ while using LL for the peripheral masks and flags or HAL for callback dispatch.

4.1.5. Code size and performance notes

HAL provides portability and convenience at a cost in code size and indirection due to handles, parameter checks, and callback plumbing; LL minimizes footprint and call depth but requires explicit sequencing and knowledge of register fields. For quantitative evaluation, the build analyzer and static stack analyzer can be used to compare configurations that favor HAL, LL, or a hybrid approach [22].

4.2. UCPD peripheral

The UCPD peripheral [17]. provides the physical layer for USB Power Delivery on the CC pins. It performs BMC line coding and decoding, 4b/5b symbol mapping, ordered-set and SOP* detection, CRC generation and checking, inter-frame timing, and hard-reset transmission and detection. It exposes a transmit and receive datapath with optional DMA on both directions, prescalers for PD half-bit timing, filters and comparators for CC attach/orientation, and wake-up paths. The role boundary in the protocol stack is that UCPD terminates at the PD physical layer: it produces and consumes bit-accurate PD packets; the protocol and policy engines, power role management, and VBUS control remain in firmware that uses these events and payloads.

The programming model consists of enabling UCPD, selecting the Type-C mode and analog submode, configuring timing prescalers and filters, and arming interrupts. For transmit, software programs the data and payload size registers and feeds the data path (or enables TXDMA), then starts transmission; the peripheral reports progression and completion through status and interrupts. For receive, software enables the RX path and either drains bytes on message end or uses RXDMA; the peripheral raises ordered-set detection and message complete events, and it validates CRC and size in hardware. Hard reset can be asserted by software or detected on the line; both cases are reported to firmware [17].

4.3. ADC

The 12-bit ADC [17] provides single-shot and continuous regular conversions with configurable sampling times, optional oversampling, and hardware triggers from timers or software start. Channel selection is through CHSELR register; sampling time via SMPR; calibration and clocking are configured in CFGR2; operating modes and overrun behavior are set in CFGR1; conversion results are read from DR. Interrupts include ADRDY, EOC, EOS, and analog watchdog events.

The analog watchdog compares conversion results against a programmable window defined by TR1 and raises a flag and optional interrupt on out-of-window conditions. It can monitor one selected channel or all regular conversions. A common usage pattern is to select the channel(s), configure sampling time and trigger, enable DMA if needed, start conversions with software or a timer trigger, and enable EOC/EOS/AWD interrupts as required [20].

4.4. Timers

General-purpose and advanced-control timers [17] provide time bases, capture/compare, PWM generation, one-pulse mode, input capture, and trigger routing to other IPs. The core registers include CR1 and CR2 for basic control, SMCR for synchronization and trigger selection, DIER to enable interrupts and DMA requests, SR for status flags, EGR for event generation, and PSC, ARR, and CCRx for prescaler, auto-reload, and compare values; advanced timers also expose BDTR for output stage control.

Timer events include update and capture/compare events that can generate interrupts or DMA requests. Hardware triggers (internal TRGI/TRGO and external inputs) allow chaining with peripherals such as ADC, and master/slave modes allow timers to be synchronized. Typical usage is to program PSC and ARR for the base period, configure CCRx and mode for PWM or compare, select optional triggers in SMCR, enable DIER bits for the required events, and set the NVIC line for the timer [17], [20].

4.5. DMA and DMAMUX

The DMA controller [17] transfers data between peripherals and memory without CPU intervention using channels configured by CCR, CNDTR, CPAR, and CMAR. Events include transfer complete, half-transfer, and transfer error, each with corresponding flags and interrupts; the global and per-channel flags are cleared via IFCR. The DMAMUX routes peripheral request signals to DMA channels; each request line is selected in the DMAMUX and then serviced by the corresponding DMA channel configuration.

Typical usage is to select a request in DMAMUX, program the channel control (direction, increment modes, data sizes, circular/normal), set the source and destination addresses and the number of items, clear any pending status, enable the channel, and configure the NVIC for DMA interrupts. Many peripherals expose enable bits for DMA requests alongside their interrupt enables and status flags; both sides must be configured for a functional transfer [17], [20].

4.6. Interrupts and software-pended workers

The STM32G0 integrates the ARM Cortex-M0+ NVIC [17], [23], which arbitrates all maskable exceptions and peripheral interrupts. Each interrupt has a programmable priority; on Cortex-M0+ a lower numerical value denotes a higher urgency, and a higher-priority handler can preempt a lower-priority one. If two pending requests have the same programmed priority, no preemption occurs between handlers of equal priority; when they become active at the same time, the NVIC serves the one with the lower exception number first.

The programming interface follows CMSIS [21] and the reference manual [17]. Enabling and disabling a peripheral IRQ is done with NVIC registers ISER and ICER or with the CMSIS functions NVIC_EnableIRQ and NVIC_DisableIRQ; pending status is controlled with ISPR and ICPR or with NVIC_SetPendingIRQ and NVIC_ClearPendingIRQ; the active state can be read via NVIC_GetActive. Interrupt priority is programmed per interrupt using the IPR fields or via NVIC_SetPriority with an implementation-defined number of upper bits taking effect. This mechanism provides a general way to request deferred execution from any context without relying on a specific peripheral.

For peripheral sources such as UCPD, DMA, USART, and timers, the device-level mask and flag logic is separate from the NVIC. Each peripheral exposes its own interrupt enable and status/clear registers (for example, IMR/ISR/ICR in UCPD, CR/ISR/ICR in USART, ISR/IFCR in DMA, DIER/SR/EGR in timers); firmware typically unmasks the event at the peripheral, clears any stale status, and then enables the corresponding IRQ line in the NVIC.

4.7. UART for tracing

The universal synchronous/asynchronous receiver-transmitter [17] provides full-duplex serial communication with configurable word length, parity, stop bits, and baud rate. Receiver oversampling by 16 or 8 is selectable to trade maximum baud rate for noise tolerance. Data moves through RDR and TDR, with status and interrupt flags for event-driven operation. References:

Control is split across CR1, CR2, and CR3 registers. CR1 enables the block and core functions and controls key interrupts such as RXNE, TXE, TC, and IDLE detection; CR2 programs frame format and advanced modes; CR3 enables DMA requests (DMAT, DMAR) and optional flow control. In interrupt-driven operation, software enables the relevant interrupts and services events by reading RDR or writing TDR; in DMA-driven operation, the peripheral asserts TX and RX requests and the DMA controller transfers data autonomously, with completion and errors signaled by the DMA controller. LL and HAL layers map these features onto portable APIs [17], [20].

4.8. Expansion board X-NUCLEO-SNK1M1

The X-NUCLEO-SNK1M1 expansion board integrates TCPP01-M12 protection and dead-battery support, providing ESD and overvoltage protection on VBUS and CC lines, surge and thermal protection, and the ability to power the base board from an attached Source. It mates with NUCLEO-64 headers and is designed to leverage the on-chip UCPD of compatible microcontrollers; for this project the Type-C receptacle and the CC and VBUS networks are used to exercise attach, cable orientation, and PD messaging. See Fig. 4.2.[24], [25].

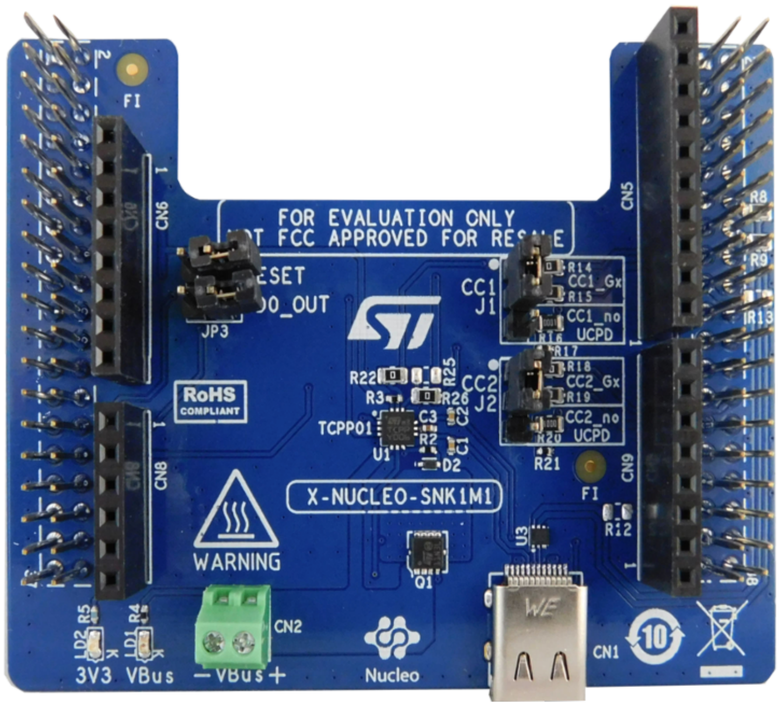
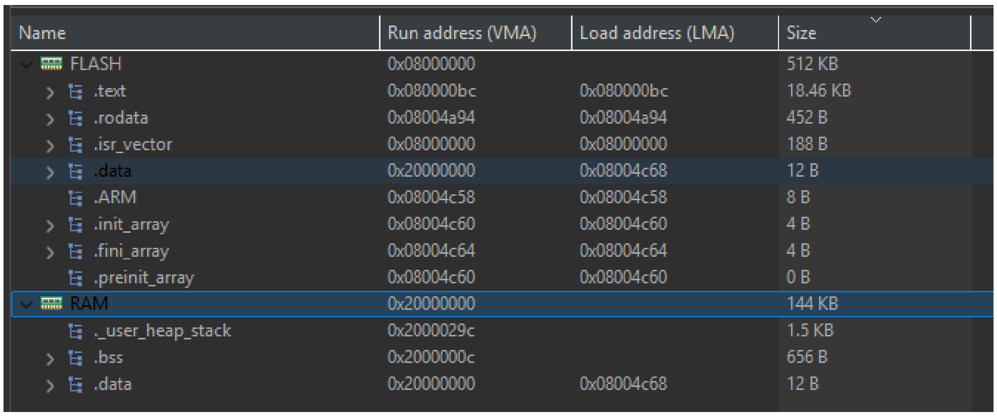


Figure 4.2. X-NUCLEO-SNK1M1 PD sink expansion board [24], [25]

4.9. Software environment

Development uses STM32CubeIDE [14], [22] with the integrated toolchain, project generator, build analyzer, and static stack analyzer for code-size and memory estimates, along with register and memory views for debugging. The on-board ST-LINK is used for download and debug. Low-level access in firmware uses the STM32G0 HAL and Low-Layer drivers described in [20].



Name	Run address (VMA)	Load address (LMA)	Size
FLASH	0x08000000		512 KB
.text	0x080000bc	0x080000bc	18.46 KB
.rodata	0x08004a94	0x08004a94	452 B
.isr_vector	0x08000000	0x08000000	188 B
.data	0x20000000	0x08004c68	12 B
.ARM	0x08004c58	0x08004c58	8 B
.init_array	0x08004c60	0x08004c60	4 B
.fini_array	0x08004c64	0x08004c64	4 B
.preinit_array	0x08004c60	0x08004c60	0 B
RAM	0x20000000		144 KB
._user_heap_stack	0x2000029c		1.5 KB
.bss	0x2000000c		656 B
.data	0x20000000	0x08004c68	12 B

Figure 4.3. Memory footprint snapshot example from the build analyzer

The IDE integrates the build analyzer (Figure 4.3) and the static stack analyzer. The build analyzer parses the linker map and the ELF to present flash and RAM consumption per section, object, and library, with navigation to symbols and delta tracking across builds;

this allows identification of large contributors to ROM and RAM and verification against budget. The static stack analyzer estimates stack usage by analyzing compiler-emitted summaries and call graphs for each entry point and reports per-task or per-function maxima, with known limitations around recursion, indirect calls, and hand-written assembly; results are presented alongside source and disassembly views [14], [22], [26]. The environment also provides register and memory views, live expressions, fault analyzer, and SFR views, which help during debug process.

5. Firmware Design and Architecture

This section provides a detailed examination of the firmware's design, detailing the architecture, its components, and the key design principles that enable it to meet the requirements outlined in Section 3. The design is founded on an event-driven model, a choice made to achieve high flexibility and efficiency on a resource-constrained microcontroller without the complexity of an RTOS or pitfalls of event-loops [27].

The architecture is organized into distinct layers. At the highest level, a Device Policy Manager (DPM) interfaces with the application and its power policy. Below this lies the core protocol stack, containing three tightly-coupled state machines for the Policy Engine (PE), Protocol Layer (PRL), and Type-C Connection Detection (CAD). The lowest layer (PHY) consists of a set of hardware-specific procedures that separate the portable protocol logic from the target MCU's peripherals.

The firmware's concurrency is managed through a "top-half" and "bottom-half" interrupt scheme. Time-critical hardware events are handled by minimal ISRs that capture state and then pend a single, deferred worker task. This worker serializes all state machine processing, preventing re-entrancy and simplifying the logic.

The following subsections will explore each of these aspects in depth. They describe the responsibilities of each layer, the flow of events through the system, the implementation of the state machines, the key data structures that enable efficient message handling, and the specific ways in which hardware peripherals like the UCPD, DMA, Timers, and ADC are utilized to realize the protocol's requirements.

5.1. Layered Architecture

The library is structured into three distinct layers: Device Policy Manager, Policy Engine/Protocol/Cable Detection module and Physical layer, as illustrated in Figure 5.1. This layered design isolates hardware dependencies, separates protocol logic from policy decisions, and provides a clear API for the end application. The separation of concerns is primarily functional, focused on the state machines. For efficiency, data structures such as message buffers are often accessed across layers, avoiding redundant data copying at the cost of strict encapsulation.

Application and Device Policy Manager (DPM) The top layer consists of the user's application and the Device Policy Manager (DPM). The DPM is the main entry point into the stack and is responsible for managing all configured USB-PD ports on the device. It receives notifications from the Policy Engine of each port and, in turn, invokes a user-provided policy callback function.

This policy callback is the primary mechanism for controlling the library's behavior. It is here that the application implements all its decision-making logic. For example, when the stack receives a Source_Capabilities message, the DPM is notified, and it calls

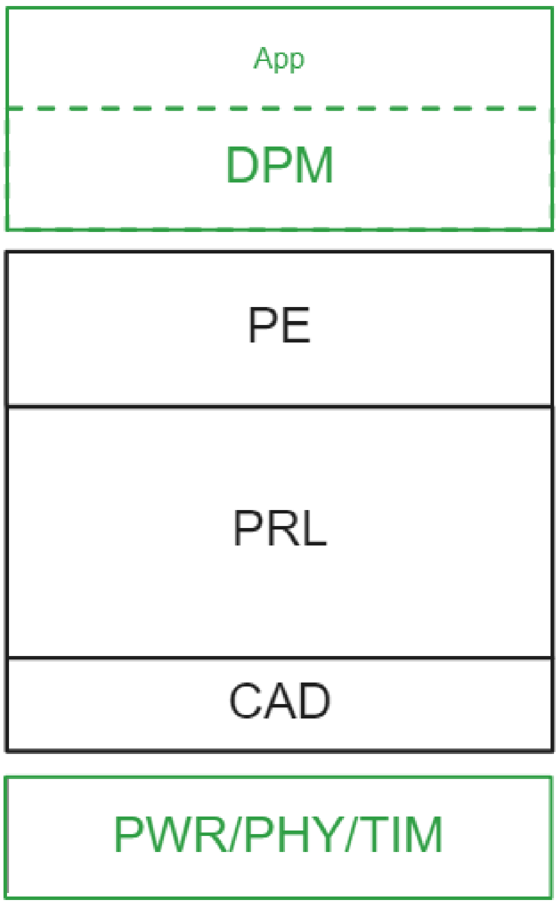


Figure 5.1. Layered architecture. Green parts contain adopter's code. In black there is only library code.

the policy function with the UCPD_POLICY_EVENT_EVALUATE_CAPABILITIES event. The policy code can then inspect the received Power Data Objects (PDOs) directly in the port's receive buffer and, if a suitable power option is found, construct a Request message in the port's transmit buffer and send it. All these operations happen in three runs of the worker interrupt. First one happens when physical layer receives the message: Policy evaluates capabilities, constructs a request and Policy Engine passes it to Protocol, which uses Physical layer procedures to send it. The next run of the worker happens when hardware notifies that the message has been sent by the Physical layer. The worker then starts waiting for a GoodCRC message and starts the corresponding timer. The third run is pended by the physical layer on message receive interrupt. Then the code checks if the GoodCRC message is correct, indicating that the Request message has been received by the Source and operation continues. This approach makes the library itself policy-agnostic; it orchestrates the protocol but delegates all decisions to the application, providing maximum flexibility.

While the policy callback is the main control path, the library also exposes a direct API for actions like requesting a specific power contract, which can be called from the main application context. In this case the exposed API procedures set a separate DPM event and pend the worker interrupt.

Core Protocol Stack (CAD, PRL, PE) The middle layer contains the core of the USB Power Delivery protocol logic. To ensure coherent and serialized processing, it is implemented as a set of tightly integrated state machines that are executed synchronously within the deferred worker task. Since the operation of Power Delivery is done through Atomic Message Sequences and each message in a sequence is coordinated by GoodCRC transmission, the general operation of message transmission and reception is quite deterministic. Thus, it can be processed in a single worker interrupt run. This composite module includes:

- **Type-C Connection Detection (CAD)**: This state machine is the first to become active, regardless of event origin (timer, ADC, UCPD peripheral or DPM). It governs the physical attach and detach sequencing according to the Type-C specification. Its primary responsibility is to manage the tCCDebounce/tPDDebounce timers and react to VBUS voltage level change (via notifications from ADC) to reliably determine when a sink is attached. Once a stable connection is confirmed, it configures the UCPD PHY for PD communication and signals the other state machines to self-initialize and startup the operation. It is also responsible for identifying a cable detach event and resetting the stack to its initial state. In the case of Hard Reset VBUS dissapears and appears as intended part of the procedure. Thus CAD must be aware when the Hard Reset is happening to ignore these transitions and not go through detach/attach.
- **Protocol Layer (PRL)**: The PRL is responsible for the reliable, transaction-level exchange of PD messages. Its state machine handles the low-level details of the protocol, such as managing the MessageID counter to identify and discard duplicate messages, handling the GoodCRC handshake to acknowledge received messages, and implementing the automatic retry mechanism (nRetryCount) in case of a failed transmission. It consists of six state machines responsible for regular message transmission and reception, chunking of messages in case of extended message transmission and reception, hard reset and a separate state machine for routing of messages to appropriate chunking state machine. It interfaces directly with the hardware abstraction to send and receive the message payloads that are prepared or consumed by the Policy Engine.
- **Policy Engine (PE)**: The PE implements the high-level logic of a PD contract negotiation. Following the sequence defined in the specification, it transitions from a startup state to discovering source capabilities in order to establish initial mandatory explicit contract. Upon receiving a Source_Capabilities message, it enters the PE_EVALUATE_CAPABILITY state, which triggers the DPM to call the user's policy. Based on the policy's decision of the capability, the PE transitions to PE_SELECT_CAPABILITY, transmits the request, and awaits the source's response (Accept or Reject), guarded

by the tSenderResponse timer. A successful negotiation concludes with the PS_RDY message, at which point an explicit contract is active. After this the operation can proceed in different ways: API emits an event for new power request or entrance to EPR, DPM requests renegotiation with new power level, Hard Reset is received or an error happens at Protocol layer that causes Soft Reset, a message is received that is not supported, etc. The PE also manages error recovery states like Soft and Hard Resets and EPR mode operation: exit, enter, requests and keeping it alive.

5.2. State Machine Implementation

The core logic of the library is implemented as a collection of finite-state machines (FSMs). This approach directly follows from general specification of the protocol behavior defined in the Power Delivery specification. The library contains several state machines, each responsible for a specific aspect of the USB-PD protocol, from physical connection to high-level policy negotiation. The worker invokes a "_Delta" function for each machine, which is responsible for consuming the latest event and orchestrating state transitions. These transitions are performed by dedicated "_Enter" and "_Exit" functions that execute the actions associated with entering or exiting each state. These machines are not independent; they interact by calling the enter functions on each other or posting events to one another through delta functions (if the machines are inside different layers), which are processed in the same worker run sequentially.

Anatomy of a State Transition The state machines are advanced by a central handler within the worker, UCPD_PE_PRL_Handler, which processes events in a specific order of precedence. A simplified view of this logic is shown in Listing 5.2. This function first checks for and processes events related to VBUS and connection removal, as these can immediately terminate any ongoing PD activity. It then gives priority to Hard Reset events, followed by timer expirations and normal PHY events like message reception; events that may be posted by DPM are processed last. This ensures that critical, time-sensitive, or connection-breaking events are handled before standard protocol messages.

Listing 1. Event processing flow in the worker handler.

```
void UCPD_PE_PRL_Handler(UCPD_PORT_Number port_number) {
    // ... obtain port instance ...

    // Block lower-priority interrupts for atomic operation
    BLOCK_EVENT(UCPD_TIMER_EVENT);
    BLOCK_EVENT(UCPD_ADC_EVENT);

    // Check events in order of priority (VBUS, detach, HRD, timers, etc.)
    if (pe_prl_cad->pwr_event != PE_PRL_CAD_EVENT_NONE) {
        UCPD_CAD_SM_Delta(pe_prl_cad, pe_prl_cad->pwr_event);
        pe_prl_cad->pwr_event = PE_PRL_CAD_EVENT_NONE;
    }
    if(pe_prl_cad->phy_event == CAD_EVENT_CONN_REMOVED) {
        UCPD_CAD_SM_Delta(pe_prl_cad, pe_prl_cad->phy_event);
        pe_prl_cad->phy_event = PE_PRL_CAD_EVENT_NONE;
    }

    if(pe_prl_cad->phy_event == PE_PRL_EVENT_HRD_RECEIVED)
    {
        UCPD_PRL_SM_Delta(pe_prl_cad, pe_prl_cad->phy_event);
        pe_prl_cad->phy_event = PE_PRL_CAD_EVENT_NONE;
        pe_prl_cad->timer_event = PE_PRL_CAD_EVENT_NONE;
    }

    // same with cc detach debounce - may exit earlier if detached
    if (pe_prl_cad->timer_event != PE_PRL_CAD_EVENT_NONE) {
        UCPD_CAD_SM_Delta(pe_prl_cad, pe_prl_cad->timer_event);
        UCPD_PRL_SM_Delta(pe_prl_cad, pe_prl_cad->timer_event);
        UCPD_PE_SM_Delta(pe_prl_cad, pe_prl_cad->timer_event);
        pe_prl_cad->timer_event = PE_PRL_CAD_EVENT_NONE;
    }

    // timers take priority over received messages
    // i.e. if we get goodcrc message and corresponding timer expires, we w
    // process the timer first
    if (pe_prl_cad->phy_event != PE_PRL_CAD_EVENT_NONE) {
        UCPD_CAD_SM_Delta(pe_prl_cad, pe_prl_cad->phy_event);
        UCPD_PRL_SM_Delta(pe_prl_cad, pe_prl_cad->phy_event);

        pe_prl_cad->phy_event = PE_PRL_CAD_EVENT_NONE;
    }

    // dpm queries have lowest priority
    if (pe_prl_cad->dpm_event != PE_PRL_CAD_EVENT_NONE) {
        UCPD_PE_SM_Delta(pe_prl_cad, pe_prl_cad->dpm_event);

        pe_prl_cad->dpm_event = PE_PRL_CAD_EVENT_NONE;
    }



    // ... call to hardware procedure f ...

    // Re-enable interrupts
    UNBLOCK_EVENT(UCPD_TIMER_EVENT);
    UNBLOCK_EVENT(UCPD_ADC_EVENT);
}
```

Type-C Connection Detection (CAD) The CAD state machine is responsible for the physical layer connection management, ensuring a stable and valid Type-C connection is established before any Power Delivery communication begins. It handles the initial attach detection, debouncing, VBUS monitoring, and the eventual detach detection. Its operation is a direct implementation of the sink-side connection state diagram from the Type-C specification.

A typical attach sequence proceeds as follows:

1. The state machine starts in the CAD_STATE_UNATTACHED state. In this state, the UCPD peripheral is configured to only generate interrupts for Type-C level events on the CC pins, and the rest of the PD stack is idle.
2. When a source is connected, a voltage appears on one of the CC pins, which triggers a PHY interrupt. The ISR posts a CAD_EVENT_TYPEC_EVENT to the worker.
3. The worker processes this event, causing the CAD machine to enter CAD_STATE_ATTACH_WAIT. Upon entering this state, the firmware starts the tCCDebounce timer (nominally 100-200ms) to filter out spurious connections or noise.
4. During this debounce period, the source must also apply VBUS. This is detected by the ADC's analog watchdog, which triggers an ISR that posts a CAD_EVENT_VBUS_DETECTED event. The CAD machine records that VBUS is now present.
5. When the tCCDebounce timer expires, its ISR posts a CAD_EVENT_TCDEBOUNCE_TIMEOUT. The worker processes this timeout, and because VBUS has already been detected, it confirms a valid connection and transitions the machine to the CAD_STATE_ATTACHED state. The VBUS appearance and CC appearance can happen in any order, but both conditions are neccessary for attach.
6. Upon entering CAD_STATE_ATTACHED, the CAD's primary job is complete. It fully enables the UCPD PHY for PD message reception and transmission and posts an internal event to init the PE and PRL state machines with initial states so that they can begin the actual PD negotiation.

Detach handling is similarly robust. If, while in the CAD_STATE_ATTACHED state, VBUS is lost(during Hard Reset this as well as VBUS appearance are ignored), the ADC watchdog will fire, posting a CAD_EVENT_VBUS_REMOVED event. This causes a transition to CAD_STATE_UNATTACHED directly. If on the other hand the CC connection is lost, the state transitions to CAD_STATE_ATTACH_WAIT and the shorter tPDDebounce timer is started. If CC does not reappear before this timer expires, the machine transitions back to CAD_STATE_UNATTACHED, and all PD activity is terminated. This two-stage process prevents CC dips from being misinterpreted as a full disconnection.

Figure 5.2. Type-C Connection Detection (CAD) State Machine.

Protocol Layer (PRL) The Protocol Layer (PRL) serves as the backbone for reliable message delivery over the CC line. It abstracts the complexities of the physical layer, providing the Policy Engine with a straightforward, transaction-based interface. Its primary duties include managing the MessageID to prevent duplicate message processing, handling the GoodCRC acknowledgment handshake, executing the automatic retry mechanism, and managing protocol-level resets. To handle these distinct but related tasks, the PRL's logic is partitioned into a set of interacting sub-state machines, each dedicated to a specific part of the protocol. Unlike the PE and CAD, the PRL does not have a single top-level state; instead, its sub-machines transition based on events processed by the central UCPD_PRL_SM_Delta function.

Listing 2. Example of a _Delta function calling an _Enter function in the PRL. If we received a message, we must send GoodCRC.

```
/* From ucpd_prl_sm.c */
void UCPD_PRL_SM_Delta(UCPD_PE_PRL_CAD_Module *pe_prl_cad,
                       UCPD_PE_PRL_CAD_Event event) {
    switch (event) {
    // ... other cases
    case PE_PRL_EVENT_MSG_RX_COMPLETE: {
        UCPD_MSG_HEADER *header =
                &pe_prl_cad->buffers[UCPD_RX_BUFFER_INDEX].header;

        if (header->message_type == UCPD_SOFT_RESET_MSG_ID) {
            UCPD_PRL_SM_RX_Enter(pe_prl_cad,
                        PRL_RX_STATE_LAYER_RESET_FOR_RECEIVE);
        } else {
            UCPD_PRL_SM_RX_Enter(pe_prl_cad,
                            PRL_RX_STATE_SEND_GOODCRC);
        }
        break;
    }
```

```
        // ... other cases
    }
}

void UCPD_PRL_SM_RX_Enter(UCPD_PE_PRL_CAD_Module *pe_prl_cad,
                          UCPD_PRL_SM_RX_State state) {
    pe_prl_cad->rx_state = state;
    switch (state) {
    case PRL_RX_STATE_SEND_GOODCRC: {
        // ... logic to construct and send a GoodCRC message ...
        UCPD_PHY_SendMessage(pe_prl_cad->port_number);
        break;
    }
    // ... other states
    }
}
```

- **Message Reception (UCPD_PRL_SM_RX_Enter)**: This machine handles all incoming messages. When the PHY ISR signals a MSG_RECEIVED event, the RX machine is activated. It transitions to the PRL_RX_STATE_SEND_GOODCRC state(unless the message is Soft Reset, in this case it first has to go through additional state but eventually will enter the state for sending GoodCRC), where it instructs the PHY to transmit a GoodCRC response containing the MessageID of the received message. It then proceeds to check the MessageID against its stored value to detect and discard any duplicate transmissions. If the message is new, it is passed up to the PE for processing; otherwise, it is ignored.
- **Message Transmission (UCPD_PRL_SM_TX_Enter)**: This machine manages the reliable transmission of outgoing messages. When the PE needs to send a message (e.g., a Request), it places the data in the transmit buffer and posts an event to the PRL. The TX machine then enters the PRL_TX_STATE_CONSTRUCT_MESSAGE state, commanding the PHY to send the message over the wire. After transmission, it starts the tCRCReceive timer and transitions to PRL_TX_STATE_WAIT_FOR_PHY_RESPONSE. If the port partner responds with a valid GoodCRC before the timer expires, the transmission is considered successful. The machine increments the MessageID counter, notifies the PE, and returns to idle. If the timer expires, it transitions to PRL_TX_STATE_CHECK_RETRY_COUNTER, where it will attempt to resend the message up to the nRetryCount limit before declaring a failure.
- **Hard Reset (UCPD_PRL_SM_HRD_Enter)**: This machine handles the sequencing of a Hard Reset. Upon receiving a request from the PE or an event from the PHY, this machine's primary role is to bring the entire protocol stack to a clean, default state.

It immediately notifies the PE to abort any ongoing negotiations, stops all protocol timers, and resets all counters, including the MessageID. It ensures the stack is ready to restart negotiation from scratch once VBUS makes switch to vSafe0V, followed by switch to vSafe5V.

- **Chunking (Router, RX, and TX)**: A set of three cooperative state machines (UCPD_PRL_SM_CHUNK UCPD_PRL_SM_CHUNKED_RX_Enter, UCPD_PRL_SM_CHUNKED_TX_Enter) is responsible for handling extended messages, which are transmitted in several parts.
  - — The **Router** FSM determines whether the message incoming from PRL is a regular message or chunk in a chunked message and passes it to Chunked RX machine, from which it will be passes to PE. Otherwise, if chunked transmission is ongoing, it will pass the message to Chunked Tx as the received message is assumed to be a request for the next chunk to be transmitted.
  - — The **RX Chunking** FSM reassembles incoming chunks. It receives each part, sends a GoodCRC for each one, validates that they are arriving in the correct order, and copies their payloads into a larger buffer until the entire message is received. Once complete, the fully reassembled message is passed to the Policy Engine. The Policy Engine is completely oblivious to chunking mechanism, it just constructs what it needs to be sent.
  - — The **TX Chunking** FSM is responsible for breaking down a large extended message from the PE into multiple chunks and sending them sequentially, waiting for a GoodCRC for each part. Each chunk sent is followed by reception of request for the next chunk. This request is also confirmed with transmission of GoodCRC.

These sub-machines work together to provide a robust and compliant messaging service, allowing the PE to focus on the high-level logic of power negotiation.

Figure 5.3. PRL Message Reception State Machine.

Figure 5.4. PRL Message Transmission State Machine.

Figure 5.5. PRL Soft Reset State Machine.

Figure 5.6. PRL Hard Reset State Machine.

Figure 5.7. PRL Chunking State Machines.

Policy Engine (PE) The PE implements the high-level logic of a PD contract negotiation and contains two distinct state machines: one for the main protocol flow and a smaller one for the Sender Response Timer (SRT). The main FSM follows the sequence defined in the specification, transitioning from a startup state to discovering capabilities. A crucial design choice is the role of the PE_SNK_STATE_EVALUATE_CAPABILITY state. Upon expiration of the PPS periodic timer (tPPSPeriodic) that signals PE to send new power request as a keep-alive measure, the machine enters the PE_SNK_STATE_EVALUATE_CAPABILITY state. This differs from the specification, where upon this timer's expiration the state entered is PE_SNK_STATE_SELECT_CAPABILITY, which sends request without reevaluation of requirements and thus requests the same contract again and again. But in this implementation the transition immediately invokes the DPM to call the user's policy callback, which may or may not request new capability. This allows the application's policy to decide whether to continue with the current PPS contract, request a different voltage, or even select a different fixed PDO entirely, all within a single, consistent mechanism. Such approach increases flexibility of Policy operation.

After the policy has made its decision (e.g., by populating a Request message in the TX buffer), the PE transitions to PE_SNK_SELECT_CAPABILITY to transmit the message. It then enters a state to await a response, while the SRT sub-machine manages the tSender-Response timeout. This timer bounds the window in which the response may come. If the timing constraint is violated and the message is not received in the designated timespan, a Hard Reset is issued in most cases(exceptions include entry to EPR mode). A successful negotiation concludes with the PS_RDY message, at which point an explicit contract is active.

Figure 5.8. Policy Engine (PE) Main State Machine.

Figure 5.9. Policy Engine (PE) Sender Response Timer State Machine.

References

[1] USB Implementers Forum, Inc. (USB-IF), *Usb type-c® cable and connector specification, release 2.4*, https://usb.org/document-library/usb-type-cr-cable-and-connector-specification-release-24, Accessed 2025-09-01, Oct. 2024.

[2] *Usb power delivery specification revision 3.1*, Initial public release (Version 1.0, May 2021). Later maintenance versions exist., USB Implementers Forum (USB-IF), 2021. [Online]. Available: https://www.usb.org/document-library/usb-power-delivery (visited on 09/15/2025).

[3] *Usb battery charging specification, revision 1.2*, Accessed 2025-09-14, USB Implementers Forum, Dec. 2010. [Online]. Available: https://www.usb.org/document-library/battery-charging-specification-revision-12.

[4] Qualcomm Technologies, Inc., *Fast charging technology | quick charge*, Accessed 2025-08-31, 2025. [Online]. Available: https://www.qualcomm.com/products/features/quick-charge.

[5] Qualcomm Technologies, Inc., *Qualcomm announces world's fastest commercial charging solution—quick charge 5*, Accessed 2025-08-31, 2020. [Online]. Available: https://www.qualcomm.com/news/releases/2020/07/qualcomm-announces-worlds-fastest-commercial-charging-solution-quick-charge.

[6] Samsung Electronics, *Adaptive fast charging wall charger (ep-ta20)*, Accessed 2025-08-31, 2025. [Online]. Available: https://www.samsung.com/us/mobile/mobile-accessories/phones/adaptive-fast-charging-wall-charger-detachable-microusb-usb-cable-white-ep-ta20jweusta/.

[7] Samsung Electronics, *Wall chargers and charging your samsung phone or tablet*, Covers Adaptive Fast Charging (AFC) and newer USB PD/PPS options; Accessed 2025-08-31, 2025. [Online]. Available: https://www.samsung.com/us/support/answer/ANS10001612/.

[8] OPPO, *Flash charging technology—technical paper*, Accessed 2025-08-31, 2018. [Online]. Available: https://www.oppo.com/content/dam/oppo/en/mkt/newsroom/story/flash-charging-technical-pape/OPPO%20Flash%20Charging%20Technical%20Paper.pdf.

[9] OPPO, *Oppo flash charging technical paper*, Overview of VOOC/SuperVOOC; Accessed 2025-08-31, 2021. [Online]. Available: https://www.oppo.com/en/newsroom/stories/flash-charging-technical-paper/.

[10] Huawei Consumer Business Group, *Huawei superpower wall charger (max 88 w)*, Lists supported protocols incl. HUAWEI SuperCharge; Accessed 2025-08-31, 2025. [Online]. Available: https://consumer.huawei.com/en/accessories/superpower-wall-charger-max-88w/.

[11] Huawei Consumer Support, *Charging modes*, Mentions SuperCharge; Accessed 2025-08-31, 2025. [Online]. Available: https://consumer.huawei.com/en/support/content/en-us00409956/.

[12] *Universal serial bus specification, revision 2.0,* Accessed 2025-09-14, USB Implementers Forum, 2000. [Online]. Available: `https://www.usb.org/document-library/usb-20-specification`.

[13] *Um2552: Managing usb power delivery systems with stm32 microcontrollers,* STMicroelectronics, 2019. [Online]. Available: `https://www.st.com/resource/en/user_manual/um2552-managing-usb-power-delivery-systems-with-stm32-microcontrollers-stmicroelectronics.pdf` (visited on 09/09/2025).

[14] "Stm32cubeide — all-in-one stm32 development tool", STMicroelectronics. (), [Online]. Available: `https://www.st.com/en/development-tools/stm32cubeide.html` (visited on 09/09/2025).

[15] "Ec implementation of usb-c power delivery and alternate modes", ChromiumOS EC Documentation. (), [Online]. Available: `https://chromium.googlesource.com/chromiumos/platform/ec/+/HEAD/docs/usb-c.md` (visited on 09/09/2025).

[16] "Ec usb-c power delivery tcpmv2 overview", ChromiumOS EC Documentation. (), [Online]. Available: `https://chromium.googlesource.com/chromiumos/platform/ec/+/HEAD/docs/usb-tcpmv2.md` (visited on 09/09/2025).

[17] *Stm32g0x1 advanced arm®-based 32-bit mcus: Reference manual (rm0444),* Latest rev as of Dec 2024, STMicroelectronics, 2024. [Online]. Available: `https://www.st.com/resource/en/reference_manual/rm0444-stm32g0x1-advanced-armbased-32bit-mcus-stmicroelectronics.pdf`.

[18] "Stm32g0b1re — mainstream arm® cortex®-m0+ mcu", STMicroelectronics. (), [Online]. Available: `https://www.st.com/en/microcontrollers-microprocessors/stm32g0b1re.html` (visited on 09/09/2025).

[19] "Stm32cubeg0 mcu package: Hal/ll drivers and middleware", STMicroelectronics. (), [Online]. Available: `https://www.st.com/en/embedded-software/stm32cubeg0.html` (visited on 09/14/2025).

[20] *Um2319: Description of stm32g0 hal and low-layer (ll) drivers,* STMicroelectronics. [Online]. Available: `https://www.st.com/resource/en/user_manual/um2319-description-of-stm32g0-hal-and-lowlayer-drivers-stmicroelectronics.pdf` (visited on 09/09/2025).

[21] "Cmsis-core (cortex-m) documentation: Nvic access using cmsis", Arm. (), [Online]. Available: `https://developer.arm.com/documentation/100235/` (visited on 09/14/2025).

[22] *Um2609: Stm32cubeide user guide,* STMicroelectronics, 2020. [Online]. Available: `https://www.st.com/resource/en/user_manual/um2609-stm32cubeide-user-guide-stmicroelectronics.pdf` (visited on 09/14/2025).

[23] *Programming manual: Cortex®-m0/m0+ core for stm32 (nvic, exceptions, registers),* Covers NVIC pending and system exception control on M0/M0+, STMicroelectronics. [Online]. Available: `https://www.mouser.com/pdfDocs/dm00104451-cortexm0-programming-manual-for-stm32l0-stm32g0-stm32wl-and-stm32wb-series-stmicroelectronics-2.pdf` (visited on 09/14/2025).

[24] *X-nucleo-snk1m1: Usb type-c™ power delivery sink expansion board based on tcpp01-m12 — data brief*, Document code: DB4427 (Rev. 3), STMicroelectronics. [Online]. Available: `https://www.st.com/resource/en/data_brief/x-nucleo-snk1m1.pdf` (visited on 09/09/2025).

[25] “X-nucleo-snk1m1 — usb type-c® pd sink expansion board based on tcpp01-m12”, STMicroelectronics. (), [Online]. Available: `https://www.st.com/en/evaluation-tools/x-nucleo-snk1m1.html` (visited on 09/09/2025).

[26] *Stm32cubeide release notes*, STMicroelectronics. [Online]. Available: `https://www.st.com/resource/en/release_note/rn0114-stm32cubeide-release-v1180-stmicroelectronics.pdf` (visited on 09/14/2025).

[27] G. Mazur, “Event-driven firmware design with hardware handler scheduling on cortex-m-based microcontroller”, *Measurement Automation Monitoring*, vol. 64, no. 1, pp. 20–22, 2018, ISSN: 2450-2855.

List of Symbols and Abbreviations

PD – Power Delivery
PDO – Power Data Object
PPS – Programmable Power Supply
APDO – Augmented PDO
SPR – Standard Power Range
EPR – Extended Power Range
EMC – Electronically Marked Cable
DRP – Dual Role Power
VDM – Vendor Defined Messages
CC – Communication Channel
SOP – Start Of Packet
RDO – Request Data Object
AMS – Atomic Message Sequence
DPM – Device Policy Manager
PE – Policy Engine
PRL – Protocol Layer
PHY – Physical Layer
UCPD – USB Type-C Power Delivery
CRC – Cyclic Redundancy Code
OVP – Over-Voltage Protection
ESD – Electrostatic Discharge
ADC – Analog to Digital Converter
AWD – Analog Watchdog
ISR – Interrupt Service Routine
NVIC – Nested Vector Interrupt Controller

List of Figures

List of Tables

List of Appendices

Appendix 1. Main loop in STM USBPD software stack

Listing 3. Main loop approach in STM's software solution

```
do
  {
    if ((HAL_GetTick() - DPM_Sleep_start[USBPD_PORT_COUNT])
        >= DPM_Sleep_time[USBPD_PORT_COUNT])
    {
      DPM_Sleep_time[USBPD_PORT_COUNT] = USBPD_CAD_Process();
      DPM_Sleep_start[USBPD_PORT_COUNT] = HAL_GetTick();
    }

    uint32_t port = 0;

    for (port = 0; port < USBPD_PORT_COUNT; port++)
    {
      if ((HAL_GetTick() - DPM_Sleep_start[port]) >= DPM_Sleep_time[port])
      {
        DPM_Sleep_time[port] =
          USBPD_PE_StateMachine_SNK(port);
        DPM_Sleep_start[port] = HAL_GetTick();
      }
    }
    //allows user to execute code once per iteration
    //limits flexibility and performance
    USBPD_DPM_UserExecute(NULL);

  } while (1u == 1u);
```