FUSB302 Integration Guide

Table of Contents

1. Introduction

The FUSB302 Type-C/PD core is a platform-agnostic codebase which, with supplied platform information, can be rapidly integrated into any platform. The core code is all contained within a directory, called `core/`. The core exposes its functionality through `core/core.h`, and the platform must expose its functionality through `core/platform.h`.

The core supports the Type-C and PD features listed in *Table 1 Supported Features and Platforms* and has been tested against the indicated platforms. As of version 3.3.0, the core can be customized for specific subsets of the total functionality, which is often desirable for platform-specific optimization. See the *Feature Selection* section for details on selecting which core features to include at compile time.

Table 1. Supported Features and Platforms

Features \ Platforms		Firmware		Linux / Android
		PIC32	***ARM Cortex-M0 & M7***	***QCOM***
Type-C	SNK	✓	✓	✓
	SNK + ACC	✓	✓	✓
	SRC	✓	✓	✓
	SRC + ACC	✓	✓	✓
	DRP	✓	✓	✓
	DRP + ACC	✓	✓	✓
	Try.SNK	✓	✓	✓
	Try.SRC	✓	✓	✓
PD	PD	✓	✓	✓
	VDM	✓	✓	✓
	DP	✓	✓	✓

2. Quick Start

For the fastest integration, take the following steps.

1. Get the latest code release, which includes a `core/` directory and platform-specific directories. The latest releases can always be downloaded through the link below.
    a. https://info.fairchildsemi.com/FUSB302_Reference.html
2. Choose a platform - if unsure, choose "`PLATFORM_NONE`", which has the bare minimum set up. See the *Compilation Options* section for build configuration details.
3. Copy your selected platform code into your own platform-specific directory.
4. Fill in the stubs located in `platform.c`. See the *Platform Functions* section for details.
5. After initializing platform code, call `core_initialize()` and `core_enable_typec(TRUE)`.
6. Call `core_tick()` every 1 ms, and call `core_state_machine()` to update the state machine according to your build configuration. See the *Polling vs. Interrupt* section for details.

3. Platform Requirements

There are two things to be done with `core/platform.h`. The first is to supply a definition for the types that the core uses. The second is to implement the platform-specific functions.

3.1. Platform Type Definition

Some core functions require precise bit-widths. The core uses abstracted types which must be defined by the platform. The reference platforms define their types in a file in their own directory, which is included in `core/platform.h*`. Select the desired platform by defining the platform's preprocessor symbol at build time. See the *Compilation Options* section for details. The types that must be defined in `core/platform.h` are described in *Table 2 Platform Type Definitions*.

Table 2. Platform Type Definitions

Type	Description
FSC_S8	Signed 8-bit integer
FSC_U8	Unsigned 8-bit integer
FSC_S16	Signed 16-bit integer
FSC_U16	Unsigned 16-bit integer
FSC_S32	Signed 32-bit integer
FSC_U32	Unsigned 32-bit integer
FSC_BOOL	Boolean
TRUE	Used with FSC_BOOL data type (must be non-zero)
FALSE	Used with FSC_BOOL data type (must be zero)

3.2. Platform Functions

The platform must implement the following functions, as defined in `core/platform.h`.

3.2.1. I2C

I2C must be run at a minimum of 400 kHz. It is recommended to issue multi-byte I2C reads and writes when possible, where the start register address is `RegisterAddress` and the total number of addresses to read/write is `DataLength`. Multi-byte reads/writes must be to contiguous, valid address ranges.

```
FSC_BOOL platform_i2c_write(FSC_U8 SlaveAddress,
                            FSC_U8 RegAddrLength,
                            FSC_U8 DataLength,
                            FSC_U8 PacketSize,
                            FSC_U8 IncSize,
                            FSC_U32 RegisterAddress,
                            FSC_U8* Data);
```

Return `FALSE` if write is successful, `TRUE` otherwise.

```
FSC_BOOL platform_i2c_read(FSC_U8 SlaveAddress,
                           FSC_U8 RegAddrLength,
                           FSC_U8 DataLength,
```

```
                                    FSC_U8 PacketSize,
                                    FSC_U8 IncSize,
                                    FSC_U32 RegisterAddress,
                                    FSC_U8* Data);
```

Return FALSE if read is successful, TRUE otherwise. If successful, then DataLength bytes of read data will be stored in Data.

3.2.2. VBUS

```
void platform_set_vbus_lvl_enable(VBUS_LVL level,
                                  FSC_BOOL blnEnable,
                                  FSC_BOOL blnDisableOthers)
```

Sets the VBUS enable specified by level to the value specified by blnEnable, where TRUE should enable the specified VBUS and FALSE should disable it. Set blnDisableOthers to TRUE to turn off all other VBUS supplies.

```
FSC_BOOL platform_get_vbus_lvl_enable(VBUS_LVL level)
```

Gets the state of the VBUS enable specified by level. Returns TRUE if VBUS is enabled, FALSE otherwise.

```
void platform_set_vbus_discharge(FSC_BOOL blnEnable)
```

Enables the VBUS discharge path if blnEnable is set to TRUE, disables it otherwise.

3.2.3. Interrupts

```
FSC_BOOL platform_get_device_irq_state(void)
```

Returns TRUE if the FUSB302 interrupt line is active. Note that the FUSB302 features an active-low interrupt pin.

3.2.4. Timers

```
FSC_BOOL platform_enable_timer(FSC_BOOL enable)
```

Enables platform timers if enable is set to TRUE, disables timers otherwise. This allows the system to save power during quiet times.

```
void platform_delay_10_µs(FSC_U32 delayCount)
```

Causes the platform to delay for 10 µs * delayCount. This function may either sleep or block, but no FUSB302 interrupts should be serviced during the delay.

```
FSC_U16 get_system_time(void)
```

A platform specific implementation that must return a running global unsigned 16 bit time in ms. The global timer may sleep when platform_enable_timer(FALSE) is called.

```
FSC_U16 platform_get_system_time(void)
```

Returns the get_system_time(void) value.

```
void platform_set_timer(TIMER *timer, FSC_U16 timeout)
```

Starts a particular timer with a given timeout value in ms. Using a timeout value of 0xFFFF disables that particular timer.

```
FSC_BOOL platform_check_timer(TIMER *timer)
```

Returns TRUE if a particular timer has expired. Returns FALSE if a particular timer has not expired or is disabled.

3.2.1. Callbacks

`FSC_BOOL platform_notify_cc_orientation(CC_ORIENTATION orientation)`

A callback used by the core to report to the platform that a Type-C connection has been established or disestablished and report the CC orientation.

`void platform_notify_pd_contract(FSC_BOOL contract)`

A callback used by the core to report to the platform that a PD contract has been established or disestablished.

`void platform_notify_unsupported_accessory(void)`

A callback used by the core to report entry to the Unsupported Accessory state. The platform may implement a USB Billboard.

`void platform_set_data_role(FSC_BOOL PolicyIsDFP)`

A callback used by the core to report the new data role after a data role swap.

`void platform_notify_bist(FSC_BOOL bistEnabled)`

A callback that may be used to limit current sinking during BIST.

4. Core Versions

The versioning scheme was changed in 3.2.0. The `core_get_rev_*` functions should be interpreted as shown in *Table 3 Version Number Decodes*. Versions 3.2.0 and beyond will continue using the straightforward versioning system.

Table 3. Version Number Decodes

Upper	Middle	Lower	Results
0x46 (ASCII 'F')	-	0x30 (ASCII '0')	Version 3.0
0x46	-	0x31 (ASCII '1')	Version 3.1
0x03	0x02	0x00	Version 3.2.0
0x03	0x03	0x00	Version 3.3.0
0x03	0x03	0x01	Version 3.3.1

5. Core Function

All functions available to the platform are declared in `core/core.h`. Some functions may only be available if the symbol FSC_DEBUG is defined in the build process.

`void core_initialize(void)`

Initializes the core. This function must be called before calling `core_state_machine()`.

`void core_enable_typec(FSC_BOOL enable)`

Enables/Disables the core Type-C state machine. `TRUE` to enable and `FALSE` to disable. Enable after calling `core_initialize()`, but before calling `core_state_machine()` for the first time.

`void core_state_machine(void)`

Runs the core state machine. In polling mode, call at least once every 4 ms. In interrupt mode, call when the FUSB302 interrupt line is active. The platform should not handle any FUSB302 interrupts until this function returns. The core must first be initialized by calling `core_initialize()`.

`void core_tick(void)`

Advances the core timers in 1 ms increments. Must be called by the platform every 1 ms (±10%).

`FSC_U8 core_get_rev_lower(void)`

Returns the lower 8 bits of the core version number.

`FSC_U8 core_get_rev_middle(void)`

(v3.2.0+) Returns the middle 8 bits of the core version number.

`FSC_U8 core_get_rev_upper(void)`

Returns the lower 8 bits of the core version number.

`void core_set_vbus_transition_time(FSC_U32 time_ms)`

Tells the core what the maximum VBUS switch transition time is for situations where the core must wait for VBUS to settle. Defaults to 20 ms.

`void core_configure_port_type(FSC_U8 config)`

Configures the core port type according to *Table 4 Port Type Configuration Bits*. This may be called at any time to update the port type, which causes the core to return to the Unattached state.

Table 4. Port Type Configuration Bits

Bit(s)	Meaning
0..1	00: Sink 01: Source 10: DRP 11: Invalid
2	0: Does not support accessories 1: Supports accessories
3	0: (DRP) No Source Preference 1: (DRP) Source Preferred
4..5	00: No Source Current 01: USB Default Current 10: 1.5A 11: 3.0A
6	0: (DRP) No Sink Preference 1: (DRP) Sink Preferred
7	0: No action 1: Type-C State Machine enabled

```
void core_enable_pd(FSC_BOOL enable)
```

Enables the PD state machine. `TRUE` to enable, `FALSE` to disable. May be called at any time, but use caution because an unexpected disable could cause a state mismatch with the attached device, and will prevent the PD state machine from capturing incoming PD traffic.

```
void core_set_source_caps(FSC_U8* buf)
```

Sets the core source capabilities according to *Table 5 Source Capability Configuration Format*. If in Source mode, core will automatically send the new capabilities to the Sink.

Table 5. Source Capability Configuration Format

Byte(s)	Use
0..1	Source Capability Header
2..5	Source Capability 1
6..9	Source Capability 2
(N*4-2)..(N*4+1)	Source Capability N

Please refer to the PD specification for Source capability header/object bitfield breakdowns.

```
void core_get_source_caps(FSC_U8* buf)
```

Gets the core source capabilities, and places them in buf. Follows format described in *Table 5 Source Capability Configuration Format*.

```
void core_set_sink_caps(FSC_U8* buf)
```

Sets the core Sink capabilities according to *Table 6 Sink Capability Configuration Format*.

Table 6. Sink Capability Configuration Format

Byte(s)	Use
0..1	Sink Capability Header
2..5	Sink Capability 1
6..9	Sink Capability 2
(N*4-2)..(N*4+1)	Sink Capability N

Please refer to the PD spec for Sink capability header/object bitfield breakdowns.

`void core_get_sink_caps(FSC_U8* buf)`

Gets the core Sink capabilities, and places them in buf. Follows format described in Table 6 *Sink Capability Configuration Format*.

`void core_set_sink_req(FSC_U8* buf)`

Sets Sink Request settings according to *Sink Request Configuration Format*.

Table 7. Sink Request Configuration Format

Byte(s)	Use
0	Bit 0: Go To Min Compatible Bit 1: USB Suspend Operation Bit 2: USB Comm. Capable Bit 3..7: Reserved
1..2	Maximum Voltage (50 mV steps)
3..6	Operating Power (0.5 mW steps)
7..10	Maximum Power (0.5 mW steps)

`void core_get_sink_req(FSC_U8* buf)`

Gets the core Sink Request configuration, and places them in buf. Follows format described in *Sink Request Configuration Format*.

`void core_send_hard_reset(void)`

Tells the core to send a hard reset.

`void core_set_state_unattached(void)`

Force state machine to detach.

`void core_reset_pd(void)`

Enables and starts the PD state machine. Used when device must power-up without PD.

`FSC_U16 core_get_advertised_current(void)`

Reports the present current advertisement of Source or Sink while in Type-C or PD contract.

`FSC_U8 core_get_cc_orientation(void)`

Reports the current CC pin orientation: 0 = no CC pin, 1 = CC1 is the CC pin, 2 = CC2 is the CC pin.

6. Compilation Options

6.1. Polling vs. Interrupt

6.1.1. Background

The FUSB302 communicates with the embedded controller (EC). It does this using the I2C bus and the INT_N signal. When the FUSB302 needs to report to the EC that something (like a device attach) is happening, it sets the INT_N pin low. The EC has two ways to see the INT_N signal go low:

1) In polling mode, the INT_N signal is connected to the EC as a regular input signal. The EC has to constantly check the INT_N signal to see if it has gone low. If the INT_N signal is low, the EC will start running the Type-C state machine firmware. In our reference code we check every 1ms using core_tick(), so in this mode the EC is always running at least every 1 ms.

2) In interrupt mode, the INT_N signal to the EC is set up as an active low, edge triggered. When the INT_N signal transitions from high to low, the EC will run an interrupt service routine and call the Type-C state machine firmware. In this mode the EC can go into a low power or sleep mode while it waits for the INT_N signal to fall.

6.1.2. Setup

By default, the core runs in polling mode, where `core_state_machine()` is assumed to be called repeatedly and consistently. To save on power, an interrupt-driven option is supplied.

In order to use the interrupt-driven option, define `FSC_INTERRUPT_TRIGGERED` in your build process.

The core assumes the interrupt handler is **falling-edge-sensitive** to the FUSB302 INT_N pin. The platform is responsible for calling `core_state_machine()` again if the INT_N pin remains low after returning from a previous call into `core_state_machine()`. Note - it is not safe to make concurrent calls into `core_state_machine()`.

In polling mode, `core_state_machine()` returns regardless of the core's active/idle state, because it is expected to be called again. In interrupt mode, `core_state_machine()` returns only once the core reaches an idle state, in which it waits for an interrupt from the FUSB302.

In interrupt mode, the core calls itself repeatedly until it can idle. To allow the system to process other tasks, `SLEEP_DELAY` is defined. This delay, in units of 10 µs, is used as an argument to `platform_delay_10 µs()` between internal passes of the state machine. It can be configured by editing the value of `SLEEP_DELAY` in `core/TypeC.h`. A delay of up to 1 ms has been tested with the platforms noted in Table 1. It is recommended to experiment with the sleep delay to optimize the core for the desired platform or application.

In interrupt mode, once `core_state_machine()` returns, it is up to the platform code to call `core_state_machine()` when the FUSB302 asserts its interrupt. It is not safe to concurrently call `core_state_machine()` while already handling a FUSB302 interrupt - wait until the function returns before calling it again.

6.2. Feature Selection

The different features of the FUSB302 can be optionally compiled in order to conserve memory on devices that only need a subset of the total functionality. This is configured by defining preprocessor symbols in the build system as described in *Valid Feature Configurations*.

Table 8. Valid Feature Configurations

Build Configuration	Requirements	Description
FSC_HAVE_SRC		Source only
FSC_HAVE_SNK		Sink only
FSC_HAVE_SNK + FSC_HAVE_SRC		Source or sink (not DRP)
FSC_HAVE_DRP	FSC_HAVE_SNK and FSC_HAVE_SRC	DRP capable source or sink
FSC_HAVE_VDM	FSC_HAVE_DP	Enable VDM support
FSC_HAVE_DP	FSC_HAVE_VDM	Enable DP support
FSC_HAVE_ACCMODE	Any valid config	Enable accessory mode
FSC_INTERRUPT_TRIGGERED	Any valid config	Enable interrupt mode. See the *Polling vs. Interrupt* section.
PLATFORM_NONE	Any valid config	Build example stub driver*
PLATFORM_PIC32	Any valid config	Build PIC32 driver*
PLATFORM_ARM	Any valid config	Build ARM driver*
FSC_PLATFORM_LINUX	Any valid config	Build Linux driver*
FSC_DEBUG + HOST_COM_USB	Any valid config	Enable debug support, including HostComm, GUI, USB-to-Host, sysfs, Type-C/PD state logs, etc

** See platform <Platform>/README.txt for details*

7. Limitations

- In 3.0 – 3.3.0, the core only supports two voltage levels – 5 V and another voltage of the platform's choosing (ex. 9 V or 12 V).
- Interrupt mode was introduced in 3.2.0. Therefore, any earlier versions are polling mode only.

8. Auto Alternate Mode Entry

- Set AutoModeEntryEnabled = TRUE in dp.c to enabled automatic alternate mode entry. Set to FALSE to disable.
- Set SVID_AUTO_ENTRY in vdm.h based on the desired SVID for automatic alternate mode entry.

9. Supporting Multiple VBUS

- Versions 3.3.1+ support the ability to implement any number of VBUS voltages.
- Add the desired voltages to the 'VBUS_LVL' enum in platform.h.
- Implement the system's mechanism for enabling and disabling VBUS in the function 'void platform_set_vbus_lvl_enable(VBUS_LVL level, FSC_BOOL blnEnable, FSC_BOOL blnDisableOthers)'
- The customer is responsible for implementing the VBUS transitions in the function 'void PolicySourceTransitionSupply(void)' in 'PDPolicy.c."
- The customer is responsible for conforming to the power rules established Section 10 of the PD Specification

10. Vendor Info File

- Complete configuration of the device, except for Auto Alternate Mode Entry, is set in core/vendor_info.h. The list of defines in this file matches exactly the list generated by the USB-IF "VendorInfoFileGenerator.exe," available from the USB-IF website. You may either alter the values in vendor_info.h directly, or run the VendorInfoFileGenerator and copy the values from the generated .txt file. Any define that is unused or not present in the generated .txt file should be left at its default value.
- Some features are marked as "Not Currently Implemented." This means that the device is able to support these features, but there has been no request for example code. Example implementation of the features can be added on request.