# ON Semiconductor

# Is Now

onsemi™

**To learn more about onsemi™, please visit our website at**
**www.onsemi.com**

**onsemi** and onsemi and other names, marks, and brands are registered and/or common law trademarks of Semiconductor Components Industries, LLC dba "**onsemi**" or its affiliates and/or subsidiaries in the United States and/or other countries. **onsemi** owns the rights to a number of patents, trademarks, copyrights, trade secrets, and other intellectual property. A listing of **onsemi** product/patent coverage may be accessed at www.onsemi.com/site/pdf/Patent-Marking.pdf. **onsemi** reserves the right to make changes at any time to any products or information herein, without notice. The information herein is provided "as-is" and **onsemi** makes no warranty, representation or guarantee regarding the accuracy of the information, product features, availability, functionality, or suitability of its products for any particular purpose, nor does **onsemi** assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation special, consequential or incidental damages. Buyer is responsible for its products and applications using **onsemi** products, including compliance with all laws, regulations and safety requirements or standards, regardless of any support or applications information provided by **onsemi**. "Typical" parameters which may be provided in **onsemi** data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. **onsemi** does not convey any license under any of its intellectual property rights nor the rights of others. **onsemi** products are not designed, intended, or authorized for use as a critical component in life support systems or any FDA Class 3 medical devices or medical devices with a same or similar classification in a foreign jurisdiction or any devices intended for implantation in the human body. Should Buyer purchase or use **onsemi** products for any such unintended or unauthorized application, Buyer shall indemnify and hold **onsemi** and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that **onsemi** was negligent regarding the design or manufacture of the part. **onsemi** is an Equal Opportunity/Affirmative Action Employer. This literature is subject to all applicable copyright laws and is not for resale in any manner. Other names and brands may be claimed as the property of others.

# AND9735/D

# FUSB307B Integration Guide

ON Semiconductor®

www.onsemi.com

## APPLICATION NOTE

This document, and the information contained herein, is CONFIDENTIAL AND PROPRIETARY and the property of Semiconductor Components Industries, LLC., dba ON Semiconductor. It shall not be used, published, disclosed or disseminated outside of the Company, in whole or in part, without the written permission of ON Semiconductor. Reverse engineering of any or all of the information contained herein is strictly prohibited.

## INTRODUCTION

The FUSB307 Type−C/PD core is a platform−agnostic code base which, when supplied platform information, can be rapidly integrated into any platform. The core code is contained within the directory, called `core/`. The core exposes its functionality through `core/core.h`, and the platform must define its functionality as declared in `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 − included as examples. 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 |
|---|---|---|---|
| | | **ARM M0** | **QCOM** |
| Type−C | SNK | ✓ | ✓ |
| | SNK + ACC | ✓* | ✓* |
| | SRC | ✓ | ✓ |
| | SRC + ACC | ✓* | ✓* |
| | DRP | ✓ | ✓ |
| | DRP + ACC | ✓* | ✓* |
| | Try. SNK | ✓ | ✓ |
| | Try. SRC | ✓ | ✓ |
| PD | PD | ✓ | ✓ |
| | VDM | ✓ | ✓ |
| | DP | ✓* | ✓* |

NOTE: ✓* Items may not be fully tested / validated.

## 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
2. Choose a platform − if unsure, choose “`PLATFORM_NONE`”, which has the bare minimum set up. See the Feature Selection 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. At run time, initialize the Platform (PlatformInitialize()), the core variables (InitializeVars(…)) , and once the device is powered up and communicating, the port behavior (InitializePort(…)). See Platform_None/main.c for an example.

## 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.

### 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) |

### Platform Functions

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

#### I2C

I2C should 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 RegisterAddress,
                            FSC_U8 DataLength,
                            FSC_U8 *Data);
```

Return TRUE if write is successful, FALSE otherwise.

```
FSC_BOOL platform_i2c_read(FSC_U8 SlaveAddress,
                           FSC_U8 RegisterAddress,
                           FSC_U8 DataLength,
                           FSC_U8 *Data);
```

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

#### Interrupts

```
FSC_BOOL platform_get_device_irq_state(FSC_U8 portID)
```

Returns TRUE if the device interrupt line is active. The value of portID is the index of the interrupt line associated with the struct port (for multiport systems). Note that the FUSB307 features an active−low interrupt pin.

#### Timers

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

(Optional) Enables platform timers if enable is set to TRUE, disables timers otherwise. This allows the system to save power during periods of inactivity.

`void platform_delay(FSC_U32 delay)`

Causes the platform to delay for `delay` microseconds. This function may either sleep or block, but no device interrupts should be serviced during the delay. Used rarely, so shouldn't cause issues of wasted CPU time.

`FSC_U32 platform_current_time(void)`

A platform specific implementation that returns a running global unsigned 32–bit time in microseconds. This value should preferably be implemented using a system or hardware timer. It is possible to reduce the resolution of this time value as low as 1ms but this may cause unreliable behavior depending on interrupt latency, etc. and should be tested. System time constants are defined in milliseconds so the constant kMSTimeFactor should be defined in core/platform.h as shown below. The global timer may sleep when `platform_enable_timer(FALSE)` is called.

| Timer Resolution | KMSTimeFactor Value |
|---|---|
| 1 ms | 1 |
| 0.1 ms | 10 |
| 0.01 ms | 100 |
| 1 μs | 1000 |

`FSC_U32 platform timestamp(void)`

Returns a system timestamp value for logging in the format 0xSSSSTTTT where 0xSSSS represents seconds and 0xTTTT represents tenths of milliseconds. This is an optional function and may be implemented with a lower resolution as long as the LSB of the value remains at 0.1 ms.

### Platform EVENT Notifications

Platform event notifications are called from within the core state machine functions and used to communicate a status or event with the embedded or application processor.

*Defining event handlers*

An event handler is a function of the following prototype `void handle_core_event(int event, int portId, void *usr_ctx, void *app_ctx)`. The event and port ID identify the port which has signaled the event in a multi–port system. The default implementation passes the numerical value starting at 1 for the first port. The user_ctx is a pointer to data that is passed during registration. The `app_ctx` data is passed by the core. For example a `PD_NEW_CONTRACT` event passes the current contract PDO to the event handler as an `app_ctx` data.

```
void PlatformEventHandler(Event_t event, FSC_U16 port_id, void *usr_ctx, void *app_ctx)
{
        struct Port *port = &g_ports[port_id - 1];
        /* Process all events */
        /* Check for attach events */
        if (CHECK_EVENT(event, EVENT_TYPEC_ATTACH))
        {
            platform_printf(port_ID, "EVENT:Attach", -1);
        }
}
```

*Registering event handlers*

An event can be registered using the function `register_observer`.

```
register_observer(EVENT_ALL, handle_core_event, 0)
```

It is also possible to register handler for only selected events.

```
register_observer (EVENT_TYPEC_ATTACH | EVENT_TYPEC_ATTACH, handle core event, 0)
```

Following events are defined in the core code. A total of 32 events can exist in the system.

**Table 3. CORE DEFINED EVENTS**

| Event ID | Description |
|---|---|
| EVENT_TYPEC_ATTACH | New plug attached. |
| EVENT_TYPEC_DETACH | Plug detached. |
| EVENT_CC1_ORIENT | Orientation of plug is CC1. |
| EVENT_CC2_ORIENT | Orientation of plug is CC2. |
| EVENT_CC_NO_ORIENT | Orientation of plug could not be determined. |
| EVENT_PD_NEW_CONTRACT | PD contract has been negotiated. |
| EVENT_PD_CONTRACT_FAILED | PD contract negotiation has failed. |
| EVENT_SRC_CAPS_UPDATED | Source capability of port partner has been updated, |
| EVENT_DATA_ROLE_DFP | Current port data role is DFP. |
| EVENT_DATA_ROLE_UFP | Current port data role is UFP. |
| EVENT_BIST_ENABLED | BIST mode has been enabled for port. |
| EVENT_BIST_DISABLED | BIST mode has been disabled for port. |
| EVENT_ALERT_RECEIVED | Alert PD message received from port partner. |
| EVENT_PPS_STATUS_RECEIVED | PPS Status message received from port partner. |
| EVENT_IDENTITY_RECEIVED | VDM Identity received from port partner. |
| EVENT_CBL_IDENTITY_RECEIVED | VDM Identity for cable received. |
| EVENT_SVID_RECEIVED | SVID for port partner received. |
| EVENT_MODES_RECEIVED | Modes for port partner received. |
| EVENT_MODE_ENTER_SUCCESS | Enter mode request from port on port partner succeeded. |
| EVENT_MODE_EXIT_SUCCESS | Exit mode request from port on port partner succeeded. |
| EVENT_MODE_VDM_ATTENTION | VDM attention received from port partner. |
| EVENT_HARD_RESET | Hardreset sent by port on port partner. |
| EVENT_UNSUPPORTED_ACCESSORY | Unsupported accessory plugged into port receptacle. |
| EVENT_DEBUG_ACCESSORY | Debug accessory plugged into port receptacle. |
| EVENT_AUDIO_ACCESSORY | Audio accessory plugged into port receptacle. |
| EVENT_ILLEGAL_CBL | Illegal cable plugged into port receptacle. |
| EVENT_ALL | Notify on all events. |

*Removing observer*

The observer handler can be unregistered using the function `remove_observer(EventHandler handler).`

*Adding new events*

While the events that are already defined may be adequate in some platforms, others will require more events notification from the firmware. To add a new event, create a `#define` in the platform file.

```
#define USER_EVENT1        PLATFORM_EVENT_ID(USER_EVENT_ID)
#define USER_EVENT2        PLATFORM_EVENT_ID(USER_EVENT_ID + 1)
#define USER_EVENT_ALL     USER_EVENT1 | USER_EVENT2
```

## 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(struct Port *port)`

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

`void core_enable_typec(struct Port *port, FSC_BOOL enable)`

Enables/Disables the core Type−C state machine of the port. `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(struct Port *port)`

Runs the core state machine. In polling mode, call at least once every 4 ms. In interrupt mode, call when the FUSB307B interrupt line is active or when a timer interrupt occurs. The platform should not handle any FUSB307B interrupts until this function returns. The core must first be initialized by calling `core_initialize()`.

`FSC_U32 core_get_next_timeout(struct Port *port)`

Returns the time until  the next active timer expires which can be used to set a match timer interrupt and allow the system to idle. A return of 0 indicates no waiting timers. A return of 1 indicates a timer has expired so continue running `core_state_machine` rather than set a timer interrupt.

`FSC_U8 core_get_rev_lower(void)`

Returns the lower 8 bits of the core version number (prerelease or patch).

`FSC_U8 core_get_rev_middle(void)`

Returns the middle 8 bits of the core version number (minor).

`FSC_U8 core_get_rev_upper(void)`

Returns the upper 8 bits of the core version number (major).

`void core_set_state_unattached(Port *port)`

Force state machine to detach.

## POLLING VS. INTERRUPT

The FUSB307 communicates with the embedded controller (EC). It does this using the I2C bus and the INT_N signal. When the FUSB307 needs to report to the EC that something (like a device attach) is happening, it sets the INT_N pin low. It is up to the EC to monitor the INT_N pin and perform the needed I2C reads at the appropriate time.

It is possible to run the state machines in a pseudo polling mode, where `core_state_machine()` is assumed to be called repeatedly and consistently and the I2C alert and status registers are read with each call to look for new events.

To save on power and prevent excessive traffic on the I2C bus, an interrupt−driven methodology is recommended.

The core assumes the interrupt handler is **falling−edge−sensitive** to the FUSB307 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()`.

There is an idle_ flag in the port structure that will indicate when it is safe and appropriate to stop repeated calls of core_state_machine(). When idle_ is TRUE, and no timers are active, the EC can be put into a low power mode or be allowed to service other tasks.  In this state, however, it is important that the INT_N pin interrupt and system timer interrupt be enabled and ready to process new events.  For examples of how this is implemented, please see main() in Platform_ARM/main.c or work_function() in Platform_Linux/platform_helpers.c

## FEATURE SELECTION

The different features of the FUSB307 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 Table 4 – Valid Feature Configurations.

Table 4. 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 + FSC_HAVE_SRC | DRP capable source or sink |
| FSC_HAVE_VDM | | Enable VDM support |
| FSC_HAVE_DP | FSC_HAVE_VDM | Enable DP support |
| FSC_HAVE_EXTENDED | Any valid config | Enable extended messaging for PD 3.0 |
| FSC_HAVE_ACCMODE | Any valid config | Enable accessory mode |
| PLATFORM_NONE | Any valid config | Build example stub driver* |
| PLATFORM_ARM | Any valid config | Build ARM driver* |
| FSC_PLATFORM_LINUX | Any valid config | Build Linux driver* |
| FSC_HAVE_USBHID | PLATFORM_ARM | Enable debug support, including HostComm, GUI, USB−to−Host, sysfs, Type−C/PD state logs, etc |
| FSC_LOGGING | Any valid config | Enable timestamped logging of state and message logs. |
| FSC_HAVE_UART | PLATFORM_ARM | Enable UART based debug terminal. |

NOTE: See platform <Platform>/README.txt for details.

## LIMITATIONS

- TBD

## SUPPORTING MULTIPLE VBUS SOURCE LEVELS

The FUSB307B is able to support multiple Vbus source voltage levels but requires additional load switches controlled by the EC. For implementation details, see PolicySourceTransitionSupply (policy.c), SendCommand (port.c) and the Platform_ARM switch and PPS controls (platform.c).

**PUBLICATION ORDERING INFORMATION**

**LITERATURE FULFILLMENT**:
Literature Distribution Center for ON Semiconductor
19521 E. 32nd Pkwy, Aurora, Colorado 80011 USA
**Phone**: 303−675−2175 or 800−344−3860 Toll Free USA/Canada
**Fax**: 303−675−2176 or 800−344−3867 Toll Free USA/Canada
**Email**: orderlit@onsemi.com

**N. American Technical Support**: 800−282−9855 Toll Free USA/Canada
**Europe, Middle East and Africa Technical Support:**
Phone: 421 33 790 2910

**ON Semiconductor Website**: **www.onsemi.com**

**Order Literature**: http://www.onsemi.com/orderlit

For additional information, please contact your local Sales Representative