cadence

USB Super Speed software driver quick start guide

Product Version 2.0.7

August 2020

1. Terms and abbreviations

The driver	Cadence USB Super Speed software driver.
The hardware	Cadence USB Super Speed hardware controller.
The user	Entity that is using the driver.
The application, user's application	A piece of software that is using the driver to carry out a particular task. This piece of software is developed and maintained by the user.
The software	A combination of the driver and the application.
The platform, user's platform	A piece of hardware that the user is running the software on.

2. Overview

This document is a quick start guide for Cadence USB Super Speed software driver intended to be run with Cadence USB Super Speed hardware controller.

This document is to provide the user with information required for integrating the driver with the application.

This document also describes the means of configuration of the driver.

3. Composition

The driver is composed of three modules:

- host - *cdn_xhci.c*,
- device - *cusbd.c*,
- dual role device - *usb_ssp_drd.c*.

Each of those modules supports different part of the hardware controller.

Host and device modules can be used separately or together with dual role device module.

4. Integration

This section describes how the driver should be integrated with user's application.

4.1. Hardware abstraction layer required by the driver

The driver operates using hardware abstraction layer declared in *cps.h*. A definition of functions declared in *cps.h* is delivered with the driver as a reference code in file *cps.c*. Hardware abstraction layer code is platform specific. The user can use definitions from *cps.c* but must make sure that those definitions are suitable for user's platform. The user is required to provide uncached access to the memory used by the hardware. Hardware abstraction layer code should be then complied and linked with user's application.

The driver requires at least the following hardware abstraction layer functions to be implemented:

- CPS_CacheFlush,
- CPS_CacheInvalidate,
- CPS_DelayNs,
- CPS_MemoryBarrier,
- CPS_UncachedRead64,
- CPS_UncachedWrite64,
- CPS_UncachedRead32,
- CPS_UncachedWrite32,
- CPS_WriteReg32,
- CPS_ReadReg32.

4.2. Callbacks of dual role device module of the driver

Dual role device module of the driver does not call directly any functions of host or device modules of the driver. Instead, dual role device module of the driver is using callbacks to user application code. This mechanism makes dual role device module of the driver flexible and allows the user to control host and device modules of the driver according to the use case of the application.

Dual role device module of the driver executes callbacks whenever it requires to interact with host or device module of the hardware. Those callbacks are supposed to call API functions of host or device modules of the driver. Dual role device module of the driver will return an error in case a required callback is missing. The user is responsible for defining those callbacks. The user is responsible for providing those callbacks to the driver during driver initialization.

The driver is delivered with a reference application code. The user should use this code as a reference for writing user's own callbacks.

4.3. Driver’s interrupt handler for the hardware interrupt

Each module of the driver supplies user with user accessible interrupt handler. USBSSP_Isr for host module of the drvier, CUSBD_Isr for device module of the driver and USB_SSP_DRD_Isr for dual role device module of the driver. Those functions are designed to be an interrupt service routines for the hardware.

4.3.1. Host and device modules of the driver used separately

When using host and device modules of the driver separately, function USBSSP_Isr or CUSBD_Isr should be called in user's application on hardware interrupt.

4.3.2. Host and device modules of the driver used with dual role device module of the driver

When using host and device modules of the driver together with dual role device module of the driver, only USB_SSP_DRD_Isr function should be called in user's application on hardware interrupt.

Function USB_SSP_DRD_Isr detects if an interrupt is caused by dual role device module of the hardware. If the interrupt is caused by dual role device module of the hardware then dual role device module of the driver will handle this interrupt. If the interrupt is caused by host or device module of the hardware then appropriate callback will be called by dual role device module of the driver based on current mode of operation of dual role device module of the driver. The user is required to define an interrupt handling procedures for host and device as callbacks in user application.

4.4. The driver build process

The driver is delivered with a makefile that can be used to build the driver into a statically linked library. This statically linked library should be then linked to user's application by the user. The user should customize delivered makefile to meet the requirements of user's platform.

4.5. Driver's initialization procedure

The driver is delivered with a reference application code. The user should use this code as a reference for writing user's own initialization procedure of the driver.

4.6. Using the driver in user's application

The user should reference *usb_ss_drd_driver_guide.pdf* for a list of user accessible functions of the driver. Only those functions should be called by user's application.

Some of user defined functions are blocking ie. they will not return until a certain transfer is performed. The user should not call those functions in an interrupt handler. The user should reference *usb_ss_drd_driver_guide.pdf* where each blocking function is documented.

The driver is delivered with a reference application code. The user should use this code as a reference for a call sequence of user accessible functions.

5. Memory size and allocation options in the driver

The driver requires four specific memory regions for operation:

- *USBSSP_DriverResourcesT* - Private data of host module of the driver. This memory has to be allocated and set to zero by the user.
- *USBSSP_XhciResourcesT* - Memory shared by the driver and by the hardware. This can be allocated by the driver or by the user based on configuration of the driver.
- *CUSBD_PrivateData* - Private data of device module of the driver. This memory has to be allocated and set to zero by the user.
- *USB_SSP_DRD_PrivData* - Private data of dual role device module of the driver. This memory has to be allocated and set to zero by the user.

Details about memory configuration options in the driver and memory allocation are described in following sections.

5.1. USBSSP_DriverResourcesT

The driver provides options for limiting driver's memory size. Following table holds driver's parameters that influence the size of memory required by the driver. Next to those parameters the table lists non-default values of the parameters for minimal memory size. Setting macros listed in the table below to values listed in the table below will limit the scope of hardware features supported by the driver.

All presented macros are defined in file *cdn_xhci_if.h*.

Table 1. Driver's configuration macros values for minimal memory footprint.

`#define`	value	description
`USBSSP_INTERRUPTER_COUNT`	1U	Disable all interrupters except one.
`USBSSP_DEMO_TB`	*undefined*	Disable memory allocation for the hardware in the driver.
`USBSSP_SCRATCHPAD_BUFF_NUM`	32U	Lower numbers of scratchpads.
`USBSSP_MAX_EP_CONTEXT_NUM`	4U	Lower number of contexts per endpoint.
`USBSSP_MAX_EP_NUM_STRM_EN`	1U	Lower number of streams per endpoint.
`USBSSP_MAX_DEVICE_SLOT_NUM`	1U	Lower number of device slots to minimum.

`#define`	value	description
`USBSSP_MAX_STREMS_PER_EP`	1U	Lower number of maximum streams.
`USBSSP_STREAM_ARRAY_SIZE`	1U	Lower size of stream array.
`DEBUG`	*undefined*	Disable debug aid.

Host module of the driver memory footprint is 8780 B when driver's memory configuration is set to above values (compiled with armv7m-none-eabi-gcc).

5.2. USBSSP_XhciResourcesT

5.2.1. Enable allocation of memory for the host module of hardware by the driver

By default host module of the driver will not allocate the memory for the host module of the hardware.

To make host module of the driver allocate memory for host module of the hardware define *USBSSP_DEMO_TB*. Host module of the driver will allocate memory for the host module of the hardware by calling a function *USBSSP_SetMemResCallback*. This option is useful for testing.

If *USBSSP_DEMO_TB* is defined, host module of the driver will allocate memory for *a single instance of host module of the hardware* corresponding to a device slot.

Host module of the driver memory footprint is 32116 B when memory for host module of the hardware is allocated by host module of the driver and the rest of memory configuration parameters are set to their default values (compiled with armv7m-none-eabi-gcc).

5.2.1.1. Force the driver to use hardware memory allocated externally

Even if memory for host module of the hardware is allocated by the host module of the driver (i.e. *USBSSP_DEMO_TB* is defined), user's application can make host module of the driver use a different memory region. This is done be setting xhciMemRes to the appropriate memory pointer by user's application.

Memory for host module of the hardware needs to be initialized to zero prior to first use. In case the driver-based memory allocation is used, this clean up is performed by the driver, using function *cleanMemRes()* defined in file *xhci_mem_alloc.c* delivered with the driver. User is required to set memory for the host module of the hardware to zero if the user forced the driver to use memory for host module of the hardware allocated outside of the driver.

5.2.2. Disable allocation of the hardware memory by the driver

Host module of the driver will not allocate memory for host module of the hardware if *USBSSP_DEMO_TB* is undefined. It can be achieved with the following code.

```
        #undef USBSSP_DEMO_TB
```

Host module of the driver memory footprint is 9352 B when allocation of memory for host module of the hardware is disabled and the rest of memory configuration parameters are set to their default values (compiled with armv7m-none-eabi-gcc).

If host module of the driver is configured not to allocate the memory for host module of the hardware, then the user is responsible for allocating the memory for host module of the hardware.

Memory for the hardware needs to be initialized to zero prior to first use. User is responsible for doing that.

5.3. USB_SSP_DRD_PrivData

Memory for this structure should only be allocated by the user only if the user is using dual role device module of the driver.

USB_SSP_DRD_PrivData structure holds pointers to *USBSSP_DriverResourcesT* and *CUSBD_PrivateData*. The user should allocate memory for *USBSSP_DriverResourcesT* and *CUSBD_PrivateData* separately. User should then feed pointers to allocated host and device structures by setting corresponding pointers in structure *USB_SSP_DRD_Config* which has to be passed as an argument to dual role device initialization procedure USB_SSP_DRD_Init.
Setup of structure *USBSSP_XhciResourcesT* is independent of dual role device module of the driver. Refer to Section 5.2 for details about memory allocation for host module of the hardware.

When dual role device module of the driver executes a callback it will pass a pointer to its *USB_SSP_DRD_PrivData* structure. This way, code of a callback has access to private data of host and device module of the driver.

6. Safety mechanisms implemented in the driver

6.1. Parameter pointer check

User accessible functions of the driver will return a non-zero error value if passed pointer is not valid.

6.2. Version of hardware

During initialization the driver (function USB_SSP_DRD_Start) will read values of hardware identification registers. If values of those registers match versions of the hardware supported by the driver, then the driver will proceed with further initialization of the hardware. Otherwise the driver will return a non-zero error code.

6.3. Memory alignment check

During initialization the driver will check if memory supplied by the user is aligned correctly. The driver will return a non-zero error code if the memory is misaligned.

Following array holds alignment requirements for memory used by the hardware. The alignment requirements are checked for all the items in the below array by the driver except for DMA.

Table 2. Alignment requirements for memory used by the hardware

Name of memory region	alignment
Input context	64 bytes
Output context	64 bytes
Scratchpad buffer	64 bytes
Command ring	64 bytes
Event ring	64 bytes
Event ring segment entry	64 bytes
DMA	4 bytes

6.4. Timeout for waiting for a register value to change

Due to hardware runtime errors, values of the registers might not change as expected.

The driver has been protected against hanging in an infinite loop while waiting for a register value to change. Number of loop iterations is limited to a value of define *USBSSP_DEFAULT_TIMEOUT* for host module of the driver and to *CUSBD_DEFAULT_TIMEOUT* for device module. Driver’s function will return non-zero error code if number of iterations will reach *USBSSP_DEFAULT_TIMEOUT* for host module and *CUSBD_DEFAULT_TIMEOUT* for device module.

Values of *USBSSP_DEFAULT_TIMEOUT* and *CUSBD_DEFAULT_TIMEOUT* are platform dependent. The user should tune them to make the driver operate in the user’s system.

6.5. Limit for the number of consecutive events handled by the driver

Due to hardware runtime error, an endless stream of incoming hardware events might occur. The driver was designed to work in a single threaded environment. This makes it vulnerable to being stuck in the event handler if hardware constantly indicates the occurrence of an unprocessed hardware event.

The driver has been protected against such hardware error. Driver’s interrupt handler will return a non-zero error code if a number of consecutive events reaches the value of *USBSSP_CONSECUTIVE_EVENTS* define.