USB 3.1 Device Class Specification for Debug Devices

Revision 1.0 – July 14, 2015

Please send comments via electronic mail to techsup@usb.org.

For industry information, refer to the USB Implementers Forum web page at http://www.usb.org.

Contributors

Intel (chair)	Rolf Kühnis
Intel	Sri Ranganathan
Intel	Sankaran Menon
Intel (chair)	John Zurawski
ST-Ericsson	Andrew Ellis
ST-Ericsson	Tomi Junnila
ST-Ericsson	Rowan Naylor
ST-Ericsson	Graham Wells
ST Microelectronics	Jean-Francis Duret
Texas Instruments	Gary Cooper
Texas Instruments	Jason Peck
Texas Instruments	Gary Swoboda
AMD	Will Harris
Lauterbach	Stephan Lauterbach
Lauterbach	Ingo Rohloff
Nokia	Henning Carlsen
Nokia	Eugene Gryazin
Nokia	Leon Jørgensen
Qualcomm	Miguel Barasch
Qualcomm	Terry Remple
Qualcomm	Yoram Rimoni

TABLE OF CONTENTS

TABLE OF FIGURES

1 Terms and Abbreviations

1.1 USB & Debug Terms and Abbreviations

Term	Description
adb	Android Debug Bridge
APE	Application Processor Engine
BELT	Best Effort Latency Tolerance. Please see USB 3.1 Architecture specification.
BOS	Binary Object Store Descriptor. See USB 3.1 Architecture specification
Capabilities	Those attributes of a USB device that are administrated by the host.
Configuration	A collection of one or more interfaces that may be selected on a USB device.
Control	A logical object within an Entity that is used to manipulate a specific property of that Entity.
Composite Device	A device that contains more than one interface descriptor is known as a *composite USB device*.
Debugger	The debug application running on the USB host that controls the debug session and receives the debug traces.
Descriptor	Data structure used to describe a USB device capability or characteristic.
DUD	Debug Unit Descriptor.
DbC	Debug Capability on the Extended Host Controller Interface
Dfx	Design for Debug or Test. This refers to a logic block that provides debug or test support.
DvC	Debug Capability on the USB device (Device Capability)
DxC	Refers to DbC or DvC interchangeably
DTS	Debug and Test System. This is the debugger application running on the host together with any probes connecting it to the Target System (i.e., device under test). For USB 3.1 Debug, this refers to the host laptop or PC running the debugger. There is typically no probe involved, but vendors may provide a Probe for enhanced capability.
DTT	Debug and Test Target (also called a Target System (TS))
DIC	Debug Interface Collection. This refers to the collection of Debug interfaces within the same

	Debug Function.
Device	USB peripheral.
Endpoint	Source or sink of data on a USB device.
Entity	A Unit, Terminal or Interface within the debug function, each of which may contain Controls.
Function	A set of one or more related interfaces that expose a capability to a software client
GUID	Global Unique Identifier. Also known as a universally unique identifier (UUID). The Guidgen.exe command line program from Microsoft is used to create a GUID. Guidgen.exe never produces the same GUID twice, no matter how many times it is run or how many different machines it runs on. Entities such as video formats that need to be uniquely identified have a GUID. Search www.microsoft.com for more information on GUIDs and Guidgen.exe.
HW	Hardware
Host	Computer system where a Host Controller is installed.
Host Controller	Hardware that connects a Host to the USB.
Host Software	Generic term for a collection of drivers, libraries and/or applications that provide operating system support for a device.
IAD	Interface Association Descriptor. This is used to describe that two or more interfaces are associated to the same function. An ‘association’ includes two or more interfaces and all of their alternate setting interfaces.
Interface	An Entity representing a collection of zero or more endpoints that present functionality to a Host.
IC	Input Connection
IP vendor	Intellectual Property vendor.
JTAG	Join Test Action Group
OC	Output Connection
OS	Operating System
OTG	On-the-Go: Supplement to the USB standard for mobile devices. Amongst other functional enhancements, it allows point-to-point communication and greater power-efficiency.
MIPI	MIPI Alliance. See www.mipi.org.
MIPI STP	MIPI System Trace Protocol [1]
Payload Transfer	In the context of the USB 3.1 Debug Class, a Payload Transfer is a unit of data transfer common to bulk and isochronous endpoints. Each Payload Transfer includes a Payload Data

	followed by Payload Footer. For isochronous endpoints, a Payload Transfer is contained in the data transmitted during a single service interval: up to 1024 bytes for a super-speed endpoint. For bulk endpoints, a Payload Transfer is contained in the data transmitted in a single bulk transfer (which may consist of multiple bulk data transactions).
Payload Data	Format-specific data contained in a Payload Transfer (excluding the Payload Footer).
Payload Footer	A header at the end of each Payload Transfer that provides data framing and encapsulation information.
Run-control	Run-control is a generic term referring to run-mode control or stop-mode control. Stop-mode control use the TAP infrastructure to perform halt, single-step, breakpoint, etc. debug operations. Run-mode uses a kernel debugger or similar software debug capability to perform similar operations.
Run-mode	See Run-control
Stop-mode	See Run-control
SoC	System on a Chip.
STM	Module implementing the MIPI STP protocol
STP	See MIPI STP
TAP	Test-Access Port
TD	Transfer Descriptor
TS	Target System (also called a Debug and Test Target (DTT))
TRB	Transfer Request Block
Trace Transfer	A trace transfer is composed of one or more payload transfer(s) representing a debug trace.
USB	Universal Serial Bus.
USB Transaction	See USB 2.0 Chapter 5.
USB Transfer	See USB 2.0 Chapter 5.
xHCI-Device	The USB 3.1 Device Controller. This is specified as an extended capability to the eXtensible Host Controller [2]
xHC(I)	USB 3.1 eXtensible Host Controller (Interface) [3]

1.2 Terminology

This document has adopted Section 13.1 of the *IEEE Standards Style Manual*, which dictates use of the words “shall”, “should”, “may”, and “can” in the development of documentation, as follows:

- The word *shall* is used to indicate mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*).
- The use of the word *must* is deprecated and shall not be used when stating mandatory requirements; *must* is used only to describe unavoidable situations.
- The use of the word *will* is deprecated and shall not be used when stating mandatory requirements; *will* is only used in statements of fact.

- The word *should* is used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended that*).
- The word *may* is used to indicate a course of action permissible within the limits of the standard (*may* equals *is permitted*).
- The word *can* is used for statements of possibility and capability, whether material, physical, or causal (*can* equals *is able to*).

All sections are normative, unless they are explicitly indicated to be informative.

This specification uses the word “device” to refer to a device-under-test (DUT) or a Target System (TS). For example, we may refer to a “mobile device” as the device being debugged. The USB architecture assigns specific meanings to the words “host” and “device”. The word “device” thus becomes confusing when debugging a “host” device, such as an OTG smartphone. In this case, the host being debugged supports the xHCI Debug Capability (DbC), where the DbC is essentially a barebones device controller. Thus, in this context, it is correct to refer to the “host” device as a device. To avoid confusion, we try to use TS instead of device.

1.3 Abbreviations

e.g.	For example (Latin: exempli gratia)
i.e.	That is (Latin: id est)
aka	also known as

2 Related Documents

[1] "MIPI Alliance Debug Architecture Overview," [Online]. Available: www.mipi.org.

[2] "USB2 Debug Device: A functional Device Specification. Rev 0.9," March 2003.

[3] "xHCI Specification," www.usb.org, May 21, 2010.

[4] "USB2 Specification Rev 2.0," www.usb.org, April 2000.

[5] "USB3 Specification Rev 3.0," www.usb.org, Nov 12, 2008.

[6] "USB Attached SCSI Protocol V1.0," 24 June 2009. [Online]. Available: http://www.usb.org/developers/devclass_docs.

[7] "USB Interface Association Descriptor Device Class Code and use model," usb.org, July 23, 2003.

[8] "Universal Serial Bus Communications Class Subclass Specifications for Network Control Model Devices," 24 Nov 2010. [Online]. Available: http://www.usb.org/developers/devclass_docs.

3 Specification Overview and Scope

3.1 Introduction

The integration of processors and hardware accelerators on a single die and within a single mobile appliance leads to a premium on pins and connectors, such that the cost of implementing a dedicated debug port is very high. In addition, the quest for ever thinner and smaller mobile devices makes it physically difficult to add any connectors, never mind a dedicated debug port. Consequently, debug connectivity is often no longer easily accessible in the late phases of Research and Development, and is often removed in the end product. This severely restricts the debugging capabilities, especially in the case of customer returns.

It is thus very attractive to share an available USB port for debug, especially if this does not preclude normal operation of the port. Enhanced SuperSpeed is ideal because it allows probe-less debug by third-party application developers with trace bandwidth of up to 10Gbps. The USB Type-C port doubles this via two SuperSpeedPlus ports.

Debug spans many usage cases. For example, it could mean accessing registers via the TAP network, or debugging the OS or an application with a GNU-type debugger, or capturing hardware or software traces, and so on. To help address these various scenarios, this specification supports a number of different debug capabilities: there is a capability for accessing a debug unit, such as a TAP controller; a capability to capture traces; and finally, a capability to access debug software. The multiple USB endpoints allow these capabilities to operate concurrently. Thus, it is possible to perform trace capture over one endpoint, and to use another pair of endpoints for run control. We provide more extensive examples later in the document.

An SoC consists of many cores, each of with may require its own debug tool. For example, the audio, video, graphics, modem and primary cores of an SoC may all come from different IP vendors, and each of these vendors may provide their proprietary debug tools. This specification allows multiple such tools with their corresponding USB drivers to all coexist on the host, and for the user to swap between tools/drivers as required to debug the different IPs within the SoC target, albeit in a mutually-exclusively manner.

In addition, the USB Debug Class specification allows extensions to the descriptors by standardization bodies and for vendor-specific use. This flexibility allows the Debug Class to accommodate future developments, similar to how the video class allows the addition of future image-compression standards. Thus, for example, a debug-standards body could develop a specification for a new debug function, and define an associated set of Debug Class specific debug commands to control this function.

The aforementioned flexibility helps ensure that the Debug Class specification accommodates the demanding debug requirements for the SoCs within mobile devices, as well as the debugging needs for laptops, PCs, servers, and other compute devices with a USB port – both, for current and for future devices.

Because most debuggers are not USB experts (and equally, most USB experts are not debug experts), then this document provides numerous examples and use cases to help bridge this knowledge gap.

Finally, debug support is often inconsistent across designs and generations of products. This leads to multiple variants of debug tools, each crafted for a particular design. This specification attempts to address this shortcoming by recommending or mandating how the USB 3.1 debug capabilities should/shall be used, to help drive standardization of debug via USB 3.1 in the industry.

3.2 Purpose

This document describes the minimum capabilities and characteristics that a USB device shall support to comply with the USB 3.1 Device-Class Specification for debug devices.

The debug function running on the USB device can use either the device endpoints provided by the USB 3.1 Device Controller or via the Debug Capability (DbC). The DbC was originally architected in the Extensible Host Controller Interface for the USB (xHCI) [3], but has been extended in this document to support Debug Control and other attributes.

Devices that conform to this specification are referred to as USB 3.1 Debug Class devices.

3.3 Scope

The USB Device-Class definition for debug devices applies to all devices or functions within composite devices that are used to manipulate debug and debug-related functionality using the capabilities defined in this specification. This includes devices such as a smartphone, tablet, laptop, desktop computers, servers, game console, embedded device, and other digital devices that provide a USB 3.1, USB 3.0, or USB 2.0 port and require debug, either in the field or in the lab. It includes the debug and test of devices from initial power-on to the tuning of software applications in a released product. Note that the target system may be a Smartphone, laptop, etc., which provides an OS. An implementation may choose to use this software in support of the debug, but this is not anticipated to be the usual case.

This specification assumes the reader is familiar with the USB 2.x, USB 3.x, and the eXtensible Host Controller Interface specifications [4], [5], [3].

USB 3.1 is a dual-bus architecture that incorporates USB 2.0 and an Enhanced SuperSpeed bus. It is a physical Enhanced SuperSpeed bus combined in parallel with a physical USB 2.0 bus. The USB 3.1 connection model accommodates backward and forward compatibility for connecting USB 3.1, USB 3.0, or USB 2.0 devices into a USB 3.1 bus. Similarly, USB 3.1 devices can be attached to a USB 2.0 bus. USB 3.1 devices accomplish backward compatibility by including both Enhanced SuperSpeed and non-SuperSpeed bus interfaces. USB 3.1 hosts also include both Enhanced SuperSpeed and non-SuperSpeed bus interfaces, which are essentially parallel buses that may be active simultaneously in a host.

A USB 3.1 peripheral device must provide support for both Enhanced SuperSpeed and at least one non-SuperSpeed speed. For a Debug Class device, we recommend HighSpeed. The minimum requirement for non-SuperSpeed is for a device to be detected on a USB 2.0 host and allow system software to direct the user to attach the device to an Enhanced SuperSpeed capable port. A device implementation may choose to provide appropriate full functionality when operating in non-SuperSpeed mode. We recommend providing a basic set of capability as a backup in case the Enhanced SuperSpeed mode fails during early debug.

Note that the USB 3.1 specification does not permit simultaneous operation of Enhanced SuperSpeed and non-SuperSpeed modes for peripheral devices. In contrast, a host or hub may support both interfaces simultaneously – see Figure 3-1.

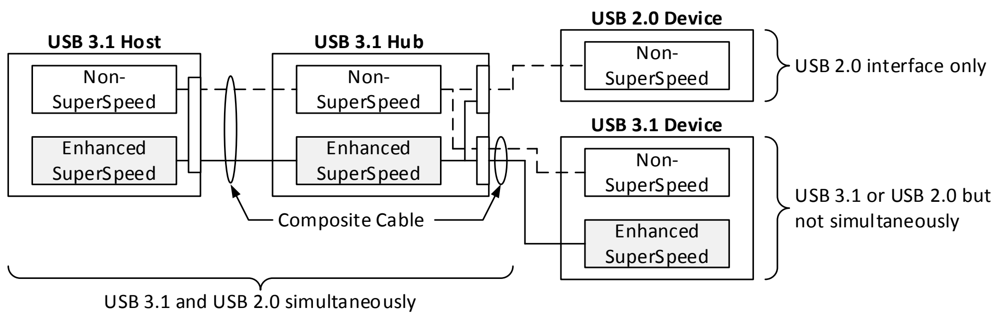


Figure 3-1: USB 3.1 and USB 2.0 interfaces

Because of this dual-bus architecture, this specification addresses debug support of a USB 3.1 or USB 2.0 host debugging a USB 3.1 or USB 2.0 device. The expectation is that debuggers will primarily focus on Enhanced SuperSpeed because of its greater bandwidth. However, this duality is beneficial, since the non-SuperSpeed interface provides a backup in case the Enhanced SuperSpeed interface is buggy and not functioning correctly (e.g., during early-phase debug).

Consequently, this debug support provides a means of connecting two systems where one system is a USB 3.1 or USB 2.0 Debug and Test System (DTS) host (i.e., where the debug tool runs) and the other is a USB 3.1 or USB 2.0 Target System (TS) (e.g., a smartphone). See option 1 in Figure 3-2. It is also possible for a single DTS to debug multiple target systems, as shown in option 2 of Figure 3-2. Appendix E: gives more examples of other debug scenarios.

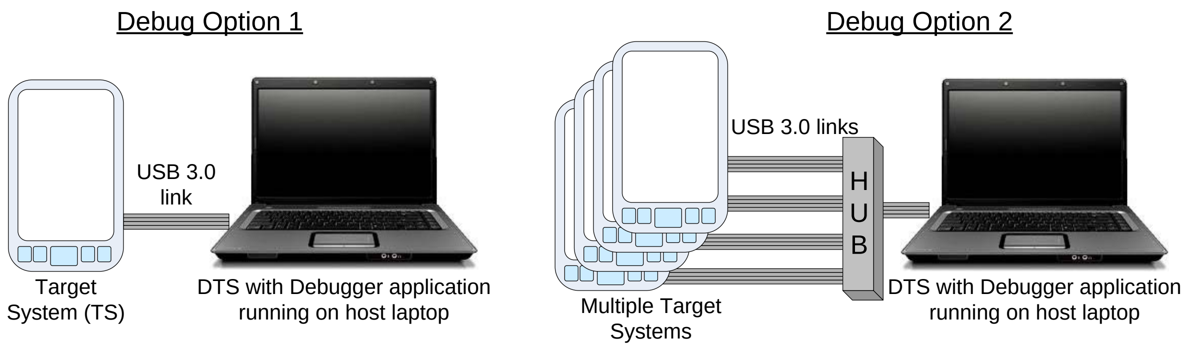

Debug Option 1
USB 3.0 link
Target System (TS)
DTS with Debugger application running on host laptop
Debug Option 2
USB 3.0 links
HUB
Multiple Target Systems
DTS with Debugger application running on host laptop

Figure 3-2: Two possible debug scenarios

The TS requires device capability. Thus, it can be a USB device, or a USB host that supports DbC.

A USB 3.1 Debug device is a standard USB device, in the sense that it supports a default Control endpoint that responds to standard USB requests, e.g. SET ADDRESS, etc. In addition, the Debug Class supports optional Debug Class specific commands on the default Control endpoints, together with an optional interrupt IN endpoint.

The Debug Class commands provide elementary capabilities to configure the debug hooks. This is particularly useful in a low-cost device with only a limited number of endpoints available for debug use. For example, a device may provide a single IN endpoint for debug trace, and use the control endpoint for configuration of the trace capability.

The Debug class predominantly uses the Bulk transfer mode, but it also supports isochronous transfers for streaming debug traces across the debug-trace interface.

The actual mechanisms used to configure and initialize the debug hooks in the TS are out of scope of this document, although elementary, optional debug commands are defined to perform basic configuration operations. The details of the TAP chains and the functionality of the debug hooks are also out of scope.

This Debug Class does not support the USB 2.0 Debug Device [2], but it does not preclude its use. The USB 2.0 debug capabilities are orthogonal to those covered by this document and thus an OTG device may choose to support both (providing the host controller provides the appropriate support).

3.4 Overview

3.4.1 Debug Capabilities

The USB 3.1 Debug Class allows the debug of current and future generations of compute devices, using debug capabilities. In the past, debug was achieved via a trace port, TAP, and a UART port accessing a kernel debugger, etc. Thus, the USB Debug Class provides the following debug *capabilities*:

Dfx: The intent of this capability is to provide access to a hardware debug unit such as a TAP controller. Once this capability is initiated, it is fully self-controlled (e.g., via a hardware state machine and not via an OS USB stack), thus allowing a host to perform stop-mode debug on a running TS.

Trace: This capability supports debug traces.

GP: The intent of this "General-Purpose" debug capability is to provide access to debug software, such as a kernel debugger or software used to configure the debug features. Unlike Dfx, this capability could be controlled by the USB OS Stack, just like a normal device endpoint. Note that an implementation may choose to use this for any general-purpose debug usage, such as accessing memory via a hardware block such as a DMA engine.

Control: The Debug class supports optional Debug Class specific control requests that allow the debugger to perform basic operations on the debug logic via the default endpoint. These include reads and writes of data structures (e.g., configuration registers), the enabling of power-management operating modes, and so on. It is possible to use these class-specific commands for basic debug operations (e.g., read/write memory), although the bandwidth via the default Control endpoint is low.

Each Dfx and GP interface requires a pair of IN/OUT endpoints, while a Trace interface requires a single IN endpoint. Debug Control uses the default Control endpoint, together with an optional Interrupt endpoint.

Certain debug scenarios may require all capabilities (i.e., Dfx, Trace, GP, and Debug Control) during the same debug session, although a typical scenario requires fewer capabilities (e.g., just Trace). In addition, a device may choose to replicate capabilities across multiple interfaces. For example, a device may provide three, independent trace interfaces, one for the modem traces, one for the main core, and one for the graphics traces. Such a device may also provide GP and Dfx interfaces, and will thus support three debug capabilities across five debug interfaces.

Note: An SoC implementation consists of multiple IP blocks, each of which may provide a trace output. An implementation may find it more convenient to treat these as separate, dedicated trace interfaces, rather than attempt to merge the traces together into a single interface. This is especially true if there are legacy debug tools associated with the IP blocks that cannot handle the new protocol necessary for a merged trace stream.

Thus, a TS implementation of the Debug class may provide from zero to many endpoints. For example, an implementation may only use the Debug Control, which uses the default endpoint; or it may use many endpoints, as given in the prior example. That example supported all four debug capabilities across six interfaces spread across seven IN or OUT endpoints together with the Default Control endpoint:

Dfx:		1 Interface using an IN/OUT endpoint pair.
Trace:	Trace 1 (Modem):	1 interface using an IN endpoint.
	Trace 2 (Core):	1 interface using an IN endpoint.
	Trace 3 (Graphics):	1 interface using an IN endpoint.
GP:		1 Interface using an IN/OUT endpoint pair
Debug Control:		1 Default Control interface using default Control endpoint 0

3.4.2 DbC and DvC Overview

3.4.2.1 DbC Overview

The Debug Class requires a USB device controller on the TS. However, a host does not have a USB device controller, and thus needs extra logic so that it behaves as a device (and not a host) during debug over USB. The xHCI specification [3] defines the DbC Debug Capability for this purpose, where the DbC is in essence a simple device controller that provides a pair of Bulk endpoints for the purpose of debug. This allows, for example, a laptop to debug another laptop that supports DbC.

The Debug Class extends upon the xHCI-compliant DbC to include the full complement of debug capabilities, namely GP, Trace, Dfx, and Control. Thus, the Debug Class defines three debug capabilities, DbC.Dfx, DbC.Trace, and DbC.GP, together with Debug Control. Each of these capabilities expands upon the original xHCI DbC. For example, DbC.Dfx, DbC.Trace, and DbC.GP support topology and Debug Control, which the original xHCI DbC did not.

Note that the xHCI DbC specification is not a USB class specification, but an architectural specification (e.g., state machines, registers, etc.). This Debug Class specification describes the capabilities of the DbC as perceived by the host, and not the actual architecture or implementation of the DbC.GP, DbC.Trace, and DbC.Dfx logic. An implementation is free to architect DbC.GP, DbC.Trace, and DbC.Dfx as they prefer, and need not adhere to the original xHCI-Compliant DbC architecture. This is most likely the case for DbC.Trace and DbC.Dfx which interface to debug hardware. DbC.GP interfaces to software, and thus there is some value in using the original xHCI DbC architecture for this particular capability.

3.4.2.2 DvC Overview

A USB host requires DbC to provide the necessary device-controller hardware for debug. A USB device clearly does not need any extra hardware – debug simply uses any of the available device endpoints.

Thus, the Debug Class uses the standard device endpoints for the three debug capabilities, and refers to them as DvC.Dfx, DvC.Trace, and DvC.GP, where DvC denotes Device (debug) Capability. The acronym DvC is purely a convenience, and unlike DbC, does not imply additional hardware support for debug (although a TS may do so if it wishes).

We also use the acronym DxC when we are referring equally to either DbC or DvC.

Thus, a Host TS requires DbC in order to be debug-able, whereas a USB device or OTG does not. Note that the Host controller of an OTG may provide DbC logic, thus allowing it to be debugged via DbC or DvC, depending upon which USB cable is inserted.

In reality, there is no difference between the DbC and DvC capabilities from the perspective of the Debug Class – they are interchangeable. The two categories, DbC and DvC, originally came about because the current implementations physically associate the DbC logic with the xHCI, while the DvC uses the endpoints of the device controller. In future designs, if the device controller and the DbC logic are integrated into one IP, then this distinction becomes immaterial.

Note that the xHCI specification restricts DbC to SuperSpeed only. This Debug class does not impose this restriction. Thus, one may use DbC.Dfx, DbC.GP, or DbC.Trace with HighSpeed.

3.4.3 Example Implementation

Figure 3-3 shows an example of a TS device providing a number of debug interfaces together with a normal interface (e.g., to a mass-storage function). Thus, in this example, a single USB port supports both normal USB 3.1 traffic together with debug traffic.

This example supports all three debug capabilities (i.e., DvC.Trace, DvC.GP, and DvC.Dfx) across four interfaces (i.e., DvC.GP, DvC.Dfx and two Trace interfaces). It also uses the default, endpoint 0 interface for the Debug Class-specific commands. For example, the debugger can use the Debug Class-specific requests to configure the debug capabilities, such as enabling the Trace-Processing unit.

The TS of Figure 3-3 has two independent Trace-Processing units that merge a number of internal traces into a single trace stream, before sending the trace out via two separate DvC.Trace interfaces. The device also provides an interface to the TAP logic via the DvC.Dfx interface. In addition, the device uses the DvC.GP interface for a kernel debugger on the host to communicate with the corresponding Kernel-debug Function on the device.

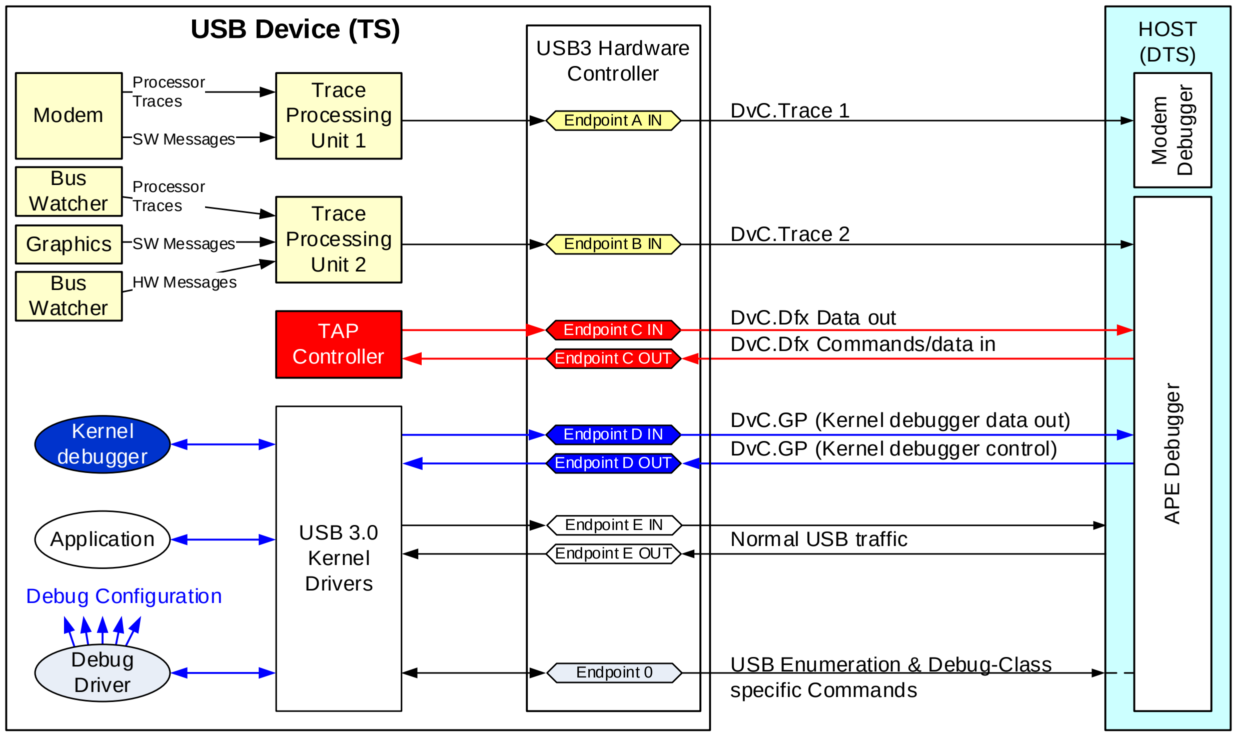


Figure 3-3: Example of a TS device supporting all Debug Capabilities

The Debug Class descriptors provide the optional capability to define the debug topology. For example, the topology could define which sources generate the debug traces and how these traces are merged to form the final trace stream sent on the trace endpoint. Thus, for example in Figure 3-3, the topology descriptors will show that the Modem traces connect to the Trace-Processing unit 1, while the Core, Graphics unit, and the Bus-Watcher unit connect to the Trace-Processing unit 2. In addition, the descriptors could provide the trace format of the output of each of these units. The Debug Class-specific commands can target these individual units for the purpose of configuration, power-gating, etc.

The following examples itemize a number of debug-use cases, and suggest the appropriate debug capability:

- *TAP debug*: DxC.Dfx is the most suitable capability for accessing this debug functionality.
- *Trace Capture*: DxC.Trace in conjunction with DxC.Dfx, DxC.GP, or the Debug Class-specific debug commands to configure and enable the traces.
- *Software debug*: DxC.GP to access the Kernel debugger, and possibly, in addition, DxC.Dfx for TAP debug when the kernel debugger hangs and the debugger needs to access hardware state.
- *System Debug of a smart device*: This may use 1, 2, or 3 of the following debug capabilities:
    - DxC.Trace for instrumentation traces (e.g., Printk-type messages from the software and/or firmware) and for hardware traces
    - DxC.Dfx, DxC.GP, or the Debug Class-specific debug commands to configure and enable the traces
    - DxC.Dfx for TAP access
- *OS USB-stack debug on a multi-port OTG device or host*: This can use DxC.GP on one port communicating with a kernel debugger on the device, while another port is acting as a normal USB 3.1 host.

3.5 Functional Characteristics

Figure 3-4 shows an example debug configuration with the corresponding standard descriptors defining the debug functionality of the Target System device.

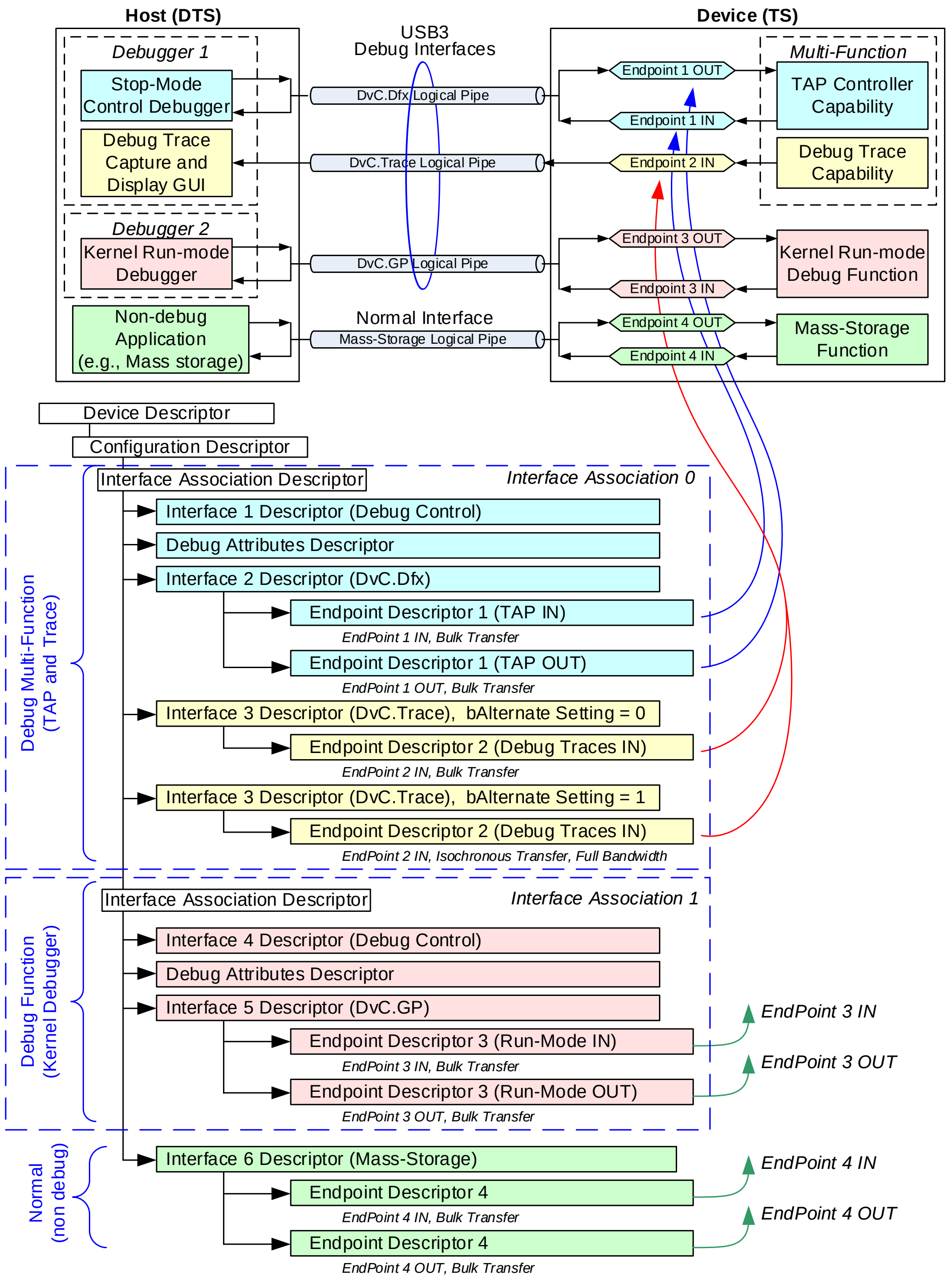

Host (DTS)
Device (TS)
Debugger 1
Stop-Mode Control Debugger
Debug Trace Capture and Display GUI
Debugger 2
Kernel Run-mode Debugger
Non-debug Application (e.g., Mass storage)
USB3 Debug Interfaces
DvC.Dfx Logical Pipe
DvC.Trace Logical Pipe
DvC.GP Logical Pipe
Normal Interface
Mass-Storage Logical Pipe
Endpoint 1 OUT
Endpoint 1 IN
Endpoint 2 IN
Endpoint 3 OUT
Endpoint 3 IN
Endpoint 4 OUT
Endpoint 4 IN
Multi-Function
TAP Controller Capability
Debug Trace Capability
Kernel Run-mode Debug Function
Mass-Storage Function
Device Descriptor
Configuration Descriptor
Interface Association Descriptor
Interface Association 0
Interface 1 Descriptor (Debug Control)
Debug Attributes Descriptor
Interface 2 Descriptor (DvC.Dfx)
Endpoint Descriptor 1 (TAP IN)
EndPoint 1 IN, Bulk Transfer
Endpoint Descriptor 1 (TAP OUT)
EndPoint 1 OUT, Bulk Transfer
Interface 3 Descriptor (DvC.Trace), bAlternate Setting = 0
Endpoint Descriptor 2 (Debug Traces IN)
EndPoint 2 IN, Bulk Transfer
Interface 3 Descriptor (DvC.Trace), bAlternate Setting = 1
Endpoint Descriptor 2 (Debug Traces IN)
EndPoint 2 IN, Isochronous Transfer, Full Bandwidth
Debug Multi-Function (TAP and Trace)
Interface Association Descriptor
Interface Association 1
Interface 4 Descriptor (Debug Control)
Debug Attributes Descriptor
Interface 5 Descriptor (DvC.GP)
Endpoint Descriptor 3 (Run-Mode IN)
EndPoint 3 IN, Bulk Transfer
Endpoint Descriptor 3 (Run-Mode OUT)
EndPoint 3 OUT, Bulk Transfer
EndPoint 3 IN
EndPoint 3 OUT
Debug Function (Kernel Debugger)
Interface 6 Descriptor (Mass-Storage)
Endpoint Descriptor 4
EndPoint 4 IN, Bulk Transfer
Endpoint Descriptor 4
EndPoint 4 OUT, Bulk Transfer
EndPoint 4 IN
EndPoint 4 OUT
Normal (non debug)

Figure 3-4: Example of the USB Descriptors for the Debug Function

Figure 3-4 is an example showing the primary USB descriptors for three DvC debug capabilities: DvC.Dfx, DvC.Trace, and DvC.GP. It also shows two debugger tools in the host – one communicates via the

DvC.GP to a kernel debugger, while the other communicates with a multi-function debug unit providing trace and Dfx capabilities. In order to support two such debuggers, two separate drivers are necessary on the host – one per debug tool. The USB uses Interface Association Descriptors (IAD) to group together the functions which pertain to a particular debug tool, thus allowing such multi-tool support.

In this example, the first three debug interfaces (i.e., Debug Control, DvC.Dfx and DvC.Trace) are grouped together into a *Debug Interface Collection* (DIC) by the Interface Association Descriptor (IAD). This grouping allows a single Debug tool to access this multi-function TAP and Trace unit.

The last debug interfaces (i.e., Debug Control and DvC.GP) is within a different DIC defined by the second IAD. This IAD is used to associate the Debug Control with the DvC.GP interface – thus the Debug commands will pertain only to the Kernel Debug function and not to the Trace and TAP multi-function unit. Thus, for example, Debugger 1 can use a Class-specific command to stop/start the hardware debug traces over the DvC.Trace interface, while Debugger 2 can use the same Class-specific command to start/stop software traces over the DvC.GP interface. Each of these commands target the particular DIC, and have no effect on the other DIC. Please see Section 5 for more details.

Section 3.6.4 provides more details on Debug Interface Collections and Interface Association Descriptors.

The host DTS in this example contains two debuggers. Debugger 1 communicates with the first DIC, while debugger 2 communicates with the second DIC. Thus the example of Figure 3-4 shows two independent debug tools interacting with different debug capabilities and functions in the target device. Debugger 1 provides trace and TAP debug support, while Debugger 2 is a kernel debugger.

To avoid overcomplicating the example, Figure 3-4 does not show the optional Debug Class-specific descriptors, which are used to define the debug topology in the TS. These are covered later in this document.

Finally, in addition to the debug interfaces, the example of Figure 3-4 also shows the device supporting a normal (i.e., non-debug) USB function (i.e., mass storage).

3.5.1 The Debug Capabilities

The Debug Capability is defined in the Interface descriptor using the bInterface, bInterfaceSubClass fields, as shown in Table 3-1.

Table 3-1: The Debug Sub-classes

bInterface Class	*bInterface Sub-Class*	*Description of Typical usage*
	DbC.GP	General-Purpose Software debug function (e.g., GNU debugger)
	DbC.Dfx	Access to hardware Dfx hooks within the host (e.g., TAP, Memory access, debug trace[1], etc.)
Diagnostic Class	DbC.Trace	Debug traces
(0xDC)	DvC.GP	General-Purpose Software debug function (e.g., GNU debugger)
	DvC.Dfx	Access to Dfx hooks within the device (e.g., TAP, Memory access, debug trace[1], etc.)
	DvC.Trace	Debug traces
	Debug Control	Control interface applicable to DxC.Trace, DxC.Dfx, DvC.GP (and optional for DbC.GP)

[1]Although, DxC.Dfx supports debug traces, DxC.Trace is the recommended interface

In general, the DbC and DvC functionality is interchangeable, and thus this Debug Class specification uses the terminology DxC to refer to either.

The USB 3.1 Debug Class provides the following capabilities:

(1) **DxC.GP (General Purpose)**: This uses a pair of standard USB Bulk IN/OUT endpoints for accessing debug software. Typical uses for DxC.GP are access to a GNU debugger or a device driver that configures the debug hooks within the device (this is analogous to the COM drivers long used for debugging purposes). DxC.GP uses bulk transfers for reliability. DvC.GP may choose to provide hardware support for this debug capability (e.g., hardware support for enumeration similar to the xHCI DbC).

(2) **DxC.Dfx:** This debug capability uses a pair of Bulk IN/OUT endpoints to access a debug hardware block within a TS. Examples of such debug blocks include the scan logic within a TS (e.g., the TAP logic), read/write access to a memory region, debug traces, etc. Typically, the Debugger uses the DxC.Dfx capability to configure and initialize the device, and to perform the usual debug run-control features such as halt and resume the CPU, perform a single-step operation, read/write memory, etc. Run-control operations require guaranteed, reliable transactions, and thus DxC.Dfx only supports Bulk transfers.

    Even though DxC.Dfx capability supports debug traces, we discourage this usage because the DTS tools may have difficulty in separating merged high-bandwidth traces from other debug traffic in real-time. Instead, we recommend using DxC.Trace exclusively for traces. However, a cost-constrained TS may only provide a single pair of debug endpoints. In this case, all debug operations, including traces, need to funnel through DxC.Dfx.

    Note that DvC.Trace supports both bulk and isochronous transfers, whilst DxC.Dfx only supports bulk transfers for reliability reasons (see Table 3-2).

(3) **DxC.Trace:** This capability is intended for the transfer of high-bandwidth debug trace to the DTS. This is the preferred capability (rather than DxC.Dfx) for traces. As an implementation note, we recommended that a TS performs the trace operations autonomously via hardware control instead of via an OS stack driver so as not to perturb the running system. This capability supports bulk and isochronous transfers.

(4) **Debug Control:** The Debug Class provides optional support for Debug Class-specific commands. For example, they allow read and write access to the configuration registers of the debug hardware, thus allowing the debugger to configure the debug hooks. Although the Debug Class-specific commands are optional, we strongly urge their use to facilitate standardization of the debug tools. See Section 5 for more details.

Table 3-2 provides the attributes of the debug capabilities. DvC may use any of the available endpoints in the device – there is no requirement to use specific, dedicated endpoints for debug (unlike the USB 2.0 Debug Device [2]).

Table 3-2: Debug Interfaces

TS	Debug Interfaces	Bus Interface	Endpoint	Data Type
DvC	DvC.GP	Enhanced SuperSpeed, SuperSpeed, HighSpeed	IN, OUT	Bulk
	DvC.Dfx		IN, OUT	Bulk
	DvC.Trace		IN	Bulk, Isochronous
DbC	DbC.GP		IN, OUT	Bulk
	DbC.Dfx		IN, OUT	Bulk
	DbC.Trace		IN	Bulk, Isochronous
DxC	Debug Control (Optional for DbC.GP)		Default Control Endpoint	Control Interrupt (Optional)

Note that the xHCI-compliant DbC only supports SuperSpeed, but the Debug Class does not impose this restriction.

3.5.2 Debug Scenario Examples

A USB 3.1 Debug Class device may implement no specific debug interface apart from supporting Debug commands over the default endpoint 0, or it may implement one or more of the debug interfaces. The simplest example is a TS that sends the debug traces to an internal buffer and uses debug commands over the default endpoint 0 to configure and extract these traces. More extensive scenarios will use multiple interfaces for trace, Dfx, and GP to access SW debug functions.

A debug lab could have multiple debuggers installed on the DTS host. For example, they may have an in-house debugger with proprietary access to a specific core in an SoC, and may have another commercial debugger that supports multiple cores, but without access to the protected features. It is quite possible that a debug user will switch between these tools during a debug session, depending on the visibility they require.

For example, one debugger controls the DvC.Dfx and DvC.Trace pipes, while another debugger controls the DvC.GP pipe. All of the DxC debug capabilities can be used concurrently.

Figure 3-5, Figure 3-6 and Figure 3-7 show a number of possible options for the debuggers within a host. These examples show the Debug Class driver in the DTS host connecting to a TS host or to a TS device.

The example options in

Figure 3-5 are:

- Option 1 shows a device that only generates debug traces. Thus, the host only requires a DvC.Trace driver. This option could use the Debug Class commands on the default endpoint 0 to configure and enable the traces, or even the Standard USB commands such Set Configuration to automatically enable the traces.
- Option 2 shows a device that only supports run-control via a DvC.Dfx interface.
- Option 3 shows three independent debugger applications running in a single host: one each for the DvC.Dfx, DvC.Trace, and the DvC.GP capability.
- Option 4 uses two debuggers connected to the three drivers

Figure 3-6 shows:

- Option 5 shows the simplest scenario of a debugger using Debug Control to access state within the device. For example, configuring the Graphics unit to send traces to a buffer in main memory, and then afterwards extracting the traces from memory.
- Option 6 shows a single, full-capability, independent debugger connected to all three drivers providing the TAP, Trace, and GNU debugger debug support
- Option 7 illustrates that a device can instantiate multiple instances of a debug capability. This example has two different DvC.Trace interfaces. For example, a SoC with integrated modem may choose to dedicate one debug-trace interface to the modem traces and a second to the non-modem traces.

Figure 3-7 visualizes:

- Option 8 is the same as Option 2 except that it is an example of host-to-host debug
- Option 9 is the same as Option 3 except that it is an example of host-to-host debug
- Option 10 assumes that the host TS is merging traces, run-control, and a memory accesses onto a single DbC.Dfx interface (see top portion of the drawing
- Figure 3-5).

An OTG device may implement both DbC and DvC, and could thus provide mutually-exclusive support for either by changing the USB cable. For example, options 5 or option 8 – depending on which USB cable is used.

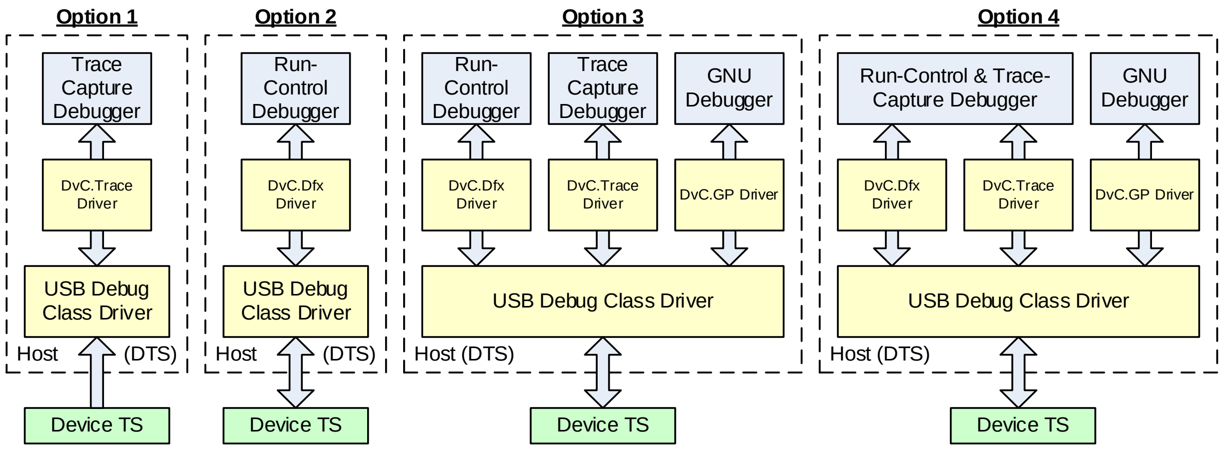


Figure 3-5: Simple Debug Scenario Examples

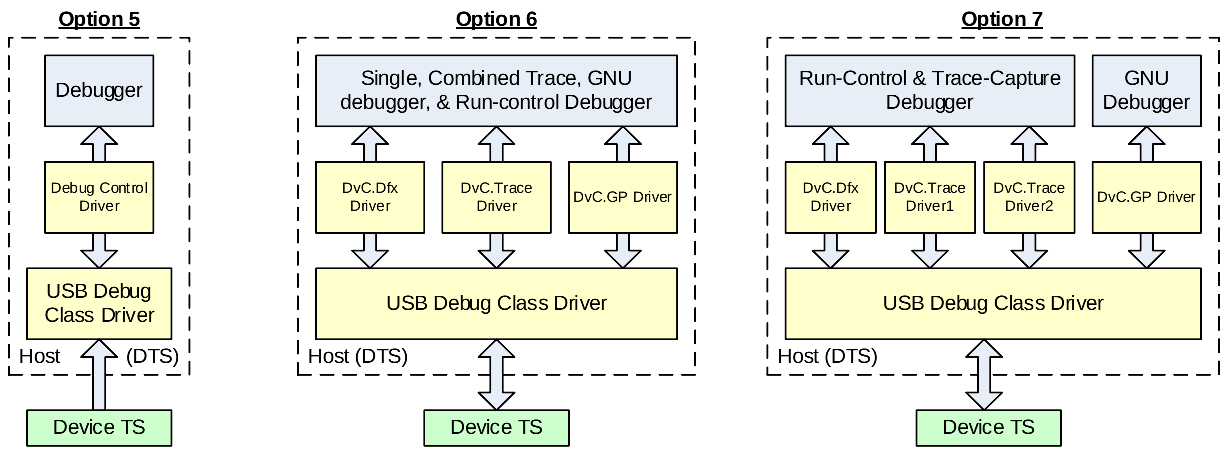


Figure 3-6: Debug Scenario Examples; Combined Tooling

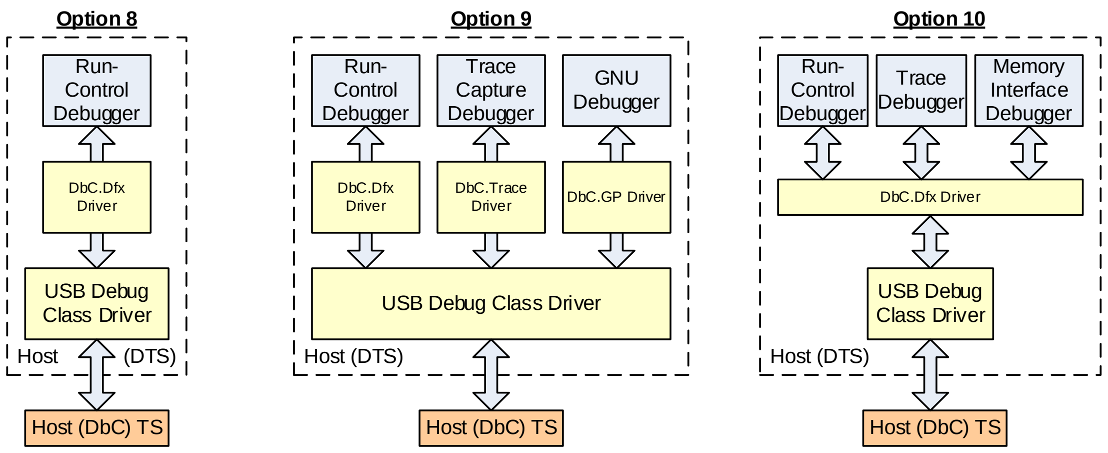


Figure 3-7: Host Debug Scenario Examples

3.5.3 Debug Function Topology

A debug function may consist of a number of interconnected components. For example, a debug trace function may consist of a network of Trace-Processing units. The optional Debug Class-specific, Debug-Unit descriptor allows one to define this network, together with the control capabilities supported by each component of the specific network.

There are two generic entities that define these components:

- Units
- Connections

Units provide the basic building blocks to fully describe the debug functions. These include agents that generate traces, such as cores, graphics units, and bus watchers, as well as merging units that combine multiple traces into a single stream. Connecting a number of such units creates a debug function (Figure 4-5 shows an example).

A Unit typically has one or more input "pins" and a single output "pin", but more complicated units consist of multiple input and output pins. Note that the term "pin" denotes a vector going in and out of a unit, and not a physical hardware pin.

One can connect multiple units together into a desired topology by connecting their I/O pins. A single output pin can connect to one or more Input pins (fan-out allowed). However, a single input pin can only connect to one output pin (fan-in disallowed) (Because it is unclear what to do when 2 or more outputs join together – for example, do they merge packets or do they drop packets on a collision). See Figure 3-8. Loops or cycles within the graph topology are disallowed.

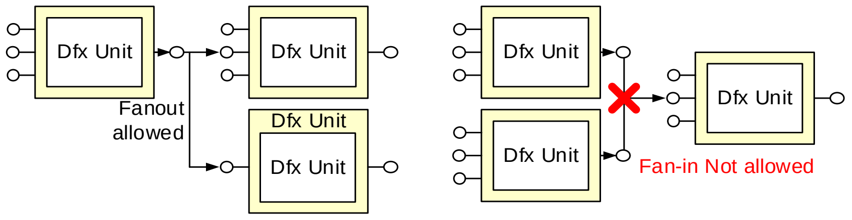


Figure 3-8: Examples of allowed and disallowed topologies

There are two types of *Connections*:

- An Input Connection (IC), which is an entity that represents a starting point for data streams inside the debug function.
- An Output Connection (OC) represents an ending point for data streams.

A USB endpoint is a typical example of an Input Connection or Output Connection.

A connection provides data streams to the debug function (i.e., IC) or consumes data streams coming from the debug function (i.e., OC).

Note: The meaning of "input" and "output" are relative to the debug unit and <u>not</u> the USB host. Hence, an Input Connection connects to an OUT endpoint, and an Output Connection connects to an IN endpoint. This use of "input" and "output" is more convenient from the perspective of the debug function.

Another example of connections is when interconnecting the debug hooks across multiple chips in a platform. In this case, one of the chips provides the primary debug interface (e.g., the main SoC), and this chip provides a debug path to the other chip (e.g., a Modem) (see top of Figure 4-5 for an example).

Input Pins of a Unit are numbered starting from one up to the total number of Input Pins on the Unit. A Pin is an entity that can be a single signal or a bus. Similarly, output pins are numbered starting from one up to the total number of Output Pins on the Unit. Connections have one Input or one Output Pin, which is always numbered one.

A Debug-Unit Descriptor (DUD) fully describes every associated Unit in the debug function. The Debug-Unit Descriptor contains all necessary fields to identify and describe the Unit. Likewise, there is a Connection Descriptor for every input and output Connection in the debug function. In addition, these descriptors provide all the necessary information about the topology of the debug function. They fully describe how Connections and Units are interconnected.

See Section 4 for more details on Debug-Unit descriptors.

This specification describes the following types of standard Connections and Units:

- Input Connection
- Output Connection
- Dfx Unit
- Select Unit
- Trace-Router Unit
- Trace-Processing Unit
- Trace-Generation Unit
- Trace-Sink Unit

These blocks can define hardware or software units. See Appendix E: for an example.

Future revisions of this specification, or companion specifications, could extend the types of Units.

In addition to the Unit and Connection descriptors, the Debug Class also defines a Debug-Control Interface descriptor together with a corresponding Debug-Attribute descriptor. This set of descriptors together with the debug-capability descriptors provides a full description of the debug function to the Host. See section 4.4 for examples and more details. These descriptors allow a host debugger to determine the topology and capabilities of the debug function on the TS.

Note: The descriptors could carry auxiliary vendor-specific information fully informing the DTS of the full details and release levels of the debug IP. Alternatively, they could provide simplified information reflecting the life-cycle support of a device. For example, in the lab, the device may provide TAP access to a modem, which is not provided in a customer device. Thus, a DTS could quickly determine the supported features within the TS.

3.5.3.1 Input Connection

The Input Connection (IC) provides an interface between the debug function and the "outside world". It serves as a receptacle for data flowing into the debug function. The IC can be a single signal or a bus. The symbol for an Input Connection is:

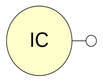


Figure 3-9: Input-Connection Icon

An Input Connection can represent inputs to the debug function other than a USB OUT endpoint. An example of such a non-USB input is JTAG pins driven by the debug function. Figure 3-10 shows Input and Output Connections connecting to a USB endpoint and to an external device via JTAG pins. In addition, the figure shows modem traces going to the Trace-Processing unit in the main SoC chip. Thus, the TS consists of multiple chips (a modem and an APE), and only the APE has a USB port.

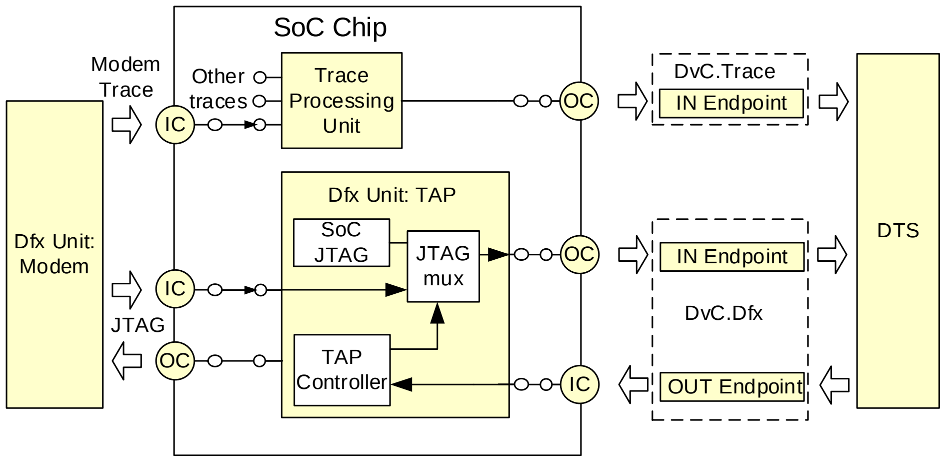


Figure 3-10: Input and Output Connections driving USB endpoint and external pins

3.5.3.1 Output Connection

The Output Connection provides an interface between the units within the debug function and the “outside world”. It serves as an outlet for debug information flowing out of the debug function. Its function is to represent a sink of outgoing data. The OC can be a single signal or a bus. The debug data stream enters the Output Connection through a single Input pin, as depicted by the Output Connection symbol:

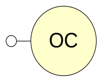
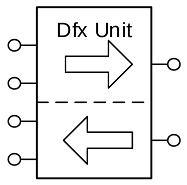


Figure 3-11: Output-Connection Icon

An Output Connection can represent outputs from the debug function other than a USB IN endpoint. The example in Figure 3-10 shows two Output Connection on the Dfx unit, where one drives JTAG pins and one drives a USB IN endpoint. If the debug stream is leaving the debug function by means of a USB IN endpoint, then there is a one-to-one relationship between that endpoint and its associated Output Connection.

3.5.3.1 Dfx Unit

The Dfx Unit is essentially a pair of units. One unit accepts n input streams, manipulates or processes the streams in some manner, and routes the result to a single output stream. The other unit accepts a single input, processes it, and creates m output streams (and the outputs need not all be the same). The symbol for a Dfx Unit is shown below together with an example beneath it:



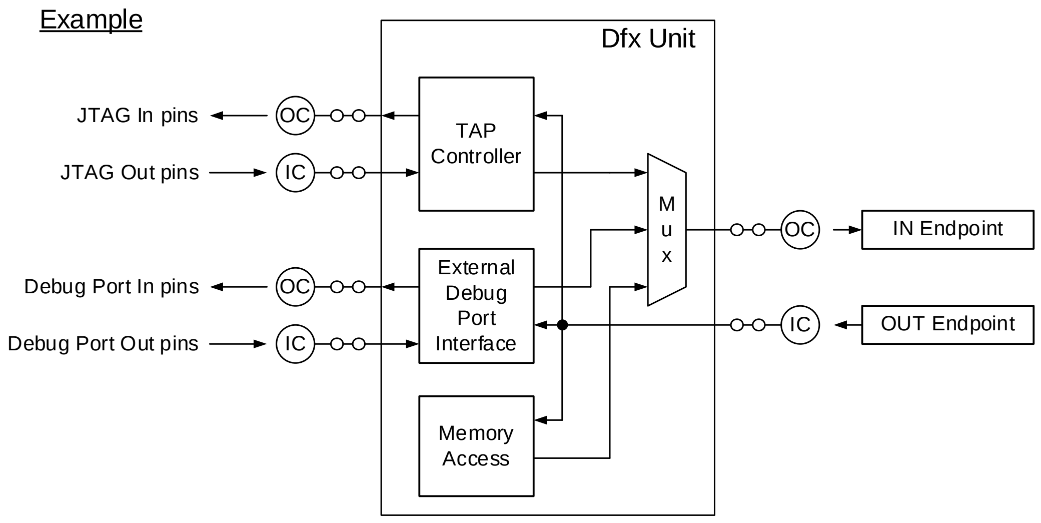


Figure 3-12: Dfx Unit icon with two sets of 2 inputs together with an example

The example in Figure 3-12 shows general debug functionality within a Dfx Unit. This particular design has a TAP controller with the ability to drive external JTAG pins, an interface to an external debug port, and a memory read/write access sub-unit. Thus, the Dfx unit is essentially a general-purpose debug block.

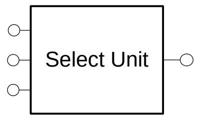
Note that the above Dfx could alternatively be partitioned into smaller Dfx units – for example, one for the TAP, one for the Memory-Access unit, etc. However, in this particular example, we assume that the Dfx unit corresponds to a single IP block, which the host debugger needs to treat as a single “black box” entity. (For example, the IP may have a single set of configuration registers, and thus the DTS needs to access this logic as an entity in order to configure it.) This explains why the Dfx icon provides multiple-inputs and multiple-outputs. See Appendix E: for more complicated scenarios.

The Debug Unit descriptor provides a field per output pin for the trace format. Thus, each output can have a different format (including no format).

Note that the Dfx unit need not only represent a hardware unit. It could also represent a debug software application accessed via DxC.GP. For example, it could represent a GNU debugger, data logger, configuration software, etc.

3.5.3.1 Select Unit

The Select unit selects one input stream from N data input pins and routes it to a single output pin. The symbol for a Select Unit is:



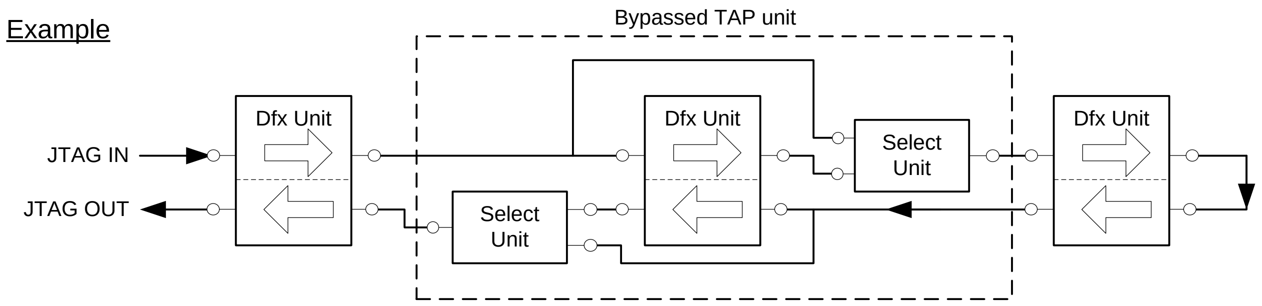


Figure 3-13: Select Unit icon with an Example TAP chain

The example in the figure shows the select units bypassing the TAP chain, where each Dfx unit is a TAP controller.

3.5.3.2 Trace-Router Unit

The Trace-Router unit redirects an input stream from a single data input pin on to 1 or more output pins that can be individually enabled. For example, it can route traces with a specific IDs in one direction, and traces with other IDs in a different direction. Note that fan out may be used if control and routing of the individual output pins is not required. The symbol for a Trace-Router Unit is:

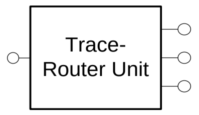


Figure 3-14: Trace-Router Unit icon with 3 outputs

The trace format on all outputs can be different.

3.5.3.3 Trace-Processing Unit

The Trace Processing unit (TPU) merges 1 or more input streams, processes them (e.g., filtering) and routes them to a single output stream. An example of a TPU is a MIPI STM. It has an Input Pin for each source stream and a single Output Pin. The symbol for a Trace-Processing Unit is:

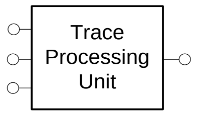


Figure 3-15: Trace Processing Unit icon with 3 inputs

An example of a TPU with a single input and single output is a trace convertor that maps an input trace stream into an output stream that supports a different trace format.

3.5.3.4 Trace-Generation Unit

The Trace Generation unit generates a single trace stream. The symbol for a Trace-Generation unit is:

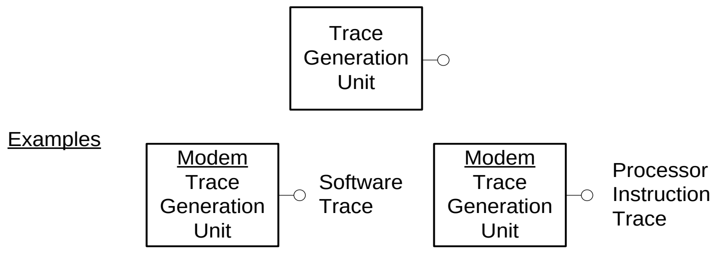

Trace Generation Unit
Examples
Modem Trace Generation Unit
Software Trace
Modem Trace Generation Unit
Processor Instruction Trace

Figure 3-16: Trace-Generation Unit Icon with an example

It is possible for a debug agent to generate multiple different traces that it sends to different debug ports. For example, a modem may generate processor traces and firmware instrumentation traces, and these traces go to different units (such as different Trace Processing units). We express this by using multiple instantiations of the Trace-Generation Unit, as shown in the example above.

3.5.3.1 Trace-Sink Unit

A Trace-Sink unit absorbs a trace – for example, a memory buffer. The symbol for a Trace-Sink unit is:

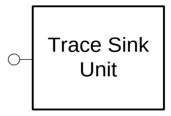


Figure 3-17: Trace-Sink Unit

A trace-Sink unit has a single Input pin and no output pins.

3.5.4 Debug Control of the Debug Units

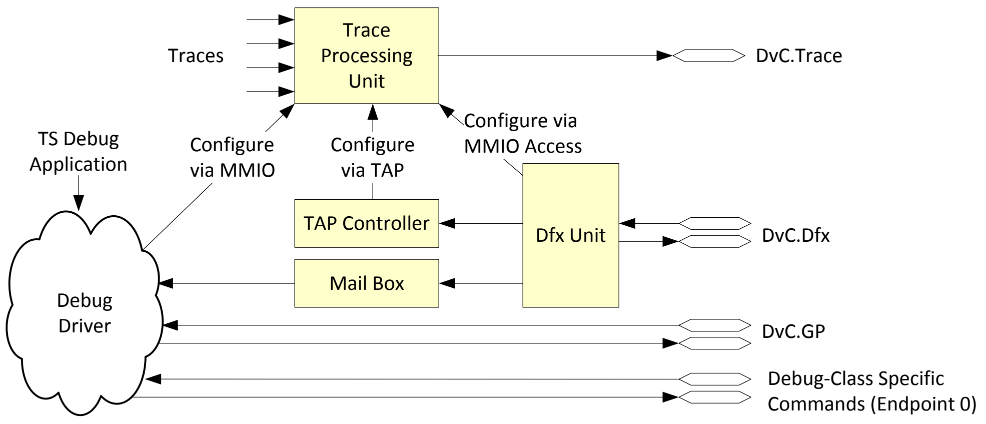


Figure 3-18: Possible means of configuring a debug unit

There are many ways to configure the debug units. For example, Figure 3-18 shows six possible ways of configuring a debug unit (in this case a Trace-Processing unit), as follows:

1) Debug Class Specific commands: These use the default endpoint 0 to communicate with a debug driver, which then configures the Trace-Processing unit.
2) Via DxC.GP: This interface communicates directly with a debug driver
3) Via an application running on the TS device
4) Via a “mailbox”: the Dfx unit uses the debug driver to configure the Trace-Processing unit via a mailbox (e.g., a UART)
5) The Dfx unit may provide direct memory read and write capability that allows access to the configuration registers in the Trace-Processing unit
6) Via TAP commands: the Dfx unit instructs a TAP Controller to scan in the configuration state into the Trace-Processing unit.

Note: The Debug Class provides elementary read/write capability of the Debug units. Other Specification bodies may provide their own set of commands that control a particular debug unit. For example, MIPI may provide a set of commands for their MIPI STM unit. In addition, IP vendors may provide a set of commands for their IP.

Section 5 provides more details on the Debug Class-specific commands.

3.6 Debug Operational Model

3.6.1 Alternate Settings

There are a number of debug scenarios where we need to provide alternative capabilities:

1. To reuse endpoints for an alternative debug capability. For example, switching the endpoints between Dfx and GP. This is of value in a cost-constrained implementation with only one endpoint pair available for debug usage. The Alternate settings would switch the endpoints amongst the different capabilities in a mutually-exclusive manner. See Section 3.6.2. Naturally, the DTS driver will have to have the capability to handle these alternate interfaces.

2. To enable multiple debuggers to reside concurrently on a host, and to switch between them. For example, during a debug session, the user may wish to rapidly switch between a commercial debug tool and a proprietary tool, where each provides a different set of capabilities. See Section 3.6.5 for more information on multiple, mutually-exclusive debug tools.

   There is a related requirement to initialize the debug function prior to switching to another debug tool, so that the new debug tool sees the function in a known clean state. We use Alternate Settings for this purpose also. See Section 3.6.5.1.

3. To select between different bandwidth options for isochronous traces. See Section 3.6.8.

These options are not mutually exclusive; they may be combined (see Figure 3-35 for an example).

This switching is achieved via the SET_INTERFACE request directed to the desired Interface with a different bAlternativeSetting.

3.6.1.1 Alternative Settings Usage Notes

When the host configures the debug configuration, it uses the first default, Interface descriptors with the bAlternativeSettings equal to zero. However, during operation, the host can send a SET_INTERFACE request directed to the desired Interface with a different bAlternativeSetting (e.g., 1, 2, etc.) and thus enable one of the other debug capabilities.

Typically, the operating system loads the driver based on the bDeviceClass/ bDeviceSubClass, and bDeviceProtocol fields in the Device descriptor. However, a debug device tends to be a Composite device, which has more than one Interface descriptor. In this case, the composite device sets the Class/SubClass/Protocol fields in the Device Descriptor to 0, and defines the multiple drivers using the Class/SubClass/Protocol fields in the various Interface descriptors. Thus, the OS loads a special Composite driver, which walks through the Interface descriptors of a device, loading the appropriate driver as a function of the bInterfaceClass/ bInterfaceSubClass/ bInterfaceProtocol values. See top of Figure 3-19 for an example.

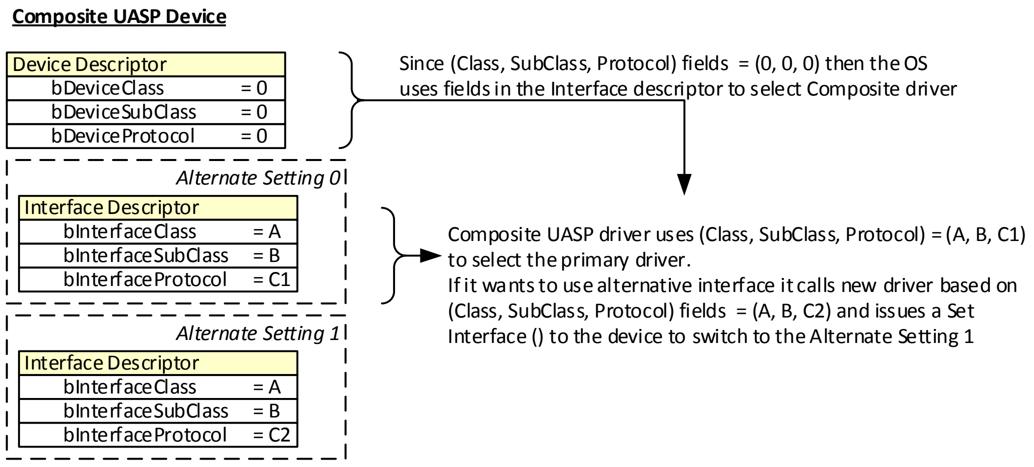

Composite UASP Device
Device Descriptor
bDeviceClass = 0
bDeviceSubClass = 0
bDeviceProtocol = 0
Since (Class, SubClass, Protocol) fields = (0, 0, 0) then the OS uses fields in the Interface descriptor to select Composite driver
Alternate Setting 0
Interface Descriptor
bInterfaceClass = A
bInterfaceSubClass = B
bInterfaceProtocol = C1
Composite UASP driver uses (Class, SubClass, Protocol) = (A, B, C1) to select the primary driver.
If it wants to use alternative interface it calls new driver based on (Class, SubClass, Protocol) fields = (A, B, C2) and issues a Set Interface () to the device to switch to the Alternate Setting 1
Alternate Setting 1
Interface Descriptor
bInterfaceClass = A
bInterfaceSubClass = B
bInterfaceProtocol = C2

Figure 3-19: Composite UASP (USB Attached SCSI Protocol) Device Example

Originally, the intent was for all alternate Interfaces to define the same InterfaceClass/ bInterfaceSubClass/ bInterfaceProtocol values in their Interface descriptors. Consequently, this means that the device driver loaded for Alternate Interface 0 had to understand what the functions of the various Alternate Interfaces are.

However, this approach proved too restrictive, and different device classes extended the capabilities. The Debug class follows the approach adopted by the USB Attached SCSI Protocol (UASP) [6]. A summary of this scheme is provided below to explain the concept:

> *All USB Storage products, Flash Drives, Thumb Drives, Hard Drives, and SSDs use a transfer protocol called Bulk Only Transfer (BOT) protocol. This is a straightforward protocol and works well for USB 2.0. However, USB 3.0 provides a new feature, called Bulk Streams, which reduces protocol overhead by allowing multiple in-flight packets. The new UASP driver class supports USB 3.0 Bulk Streams, offering greater performance than BOT. However, not all USB 3.0 devices support Bulk Streams, and for backward compatibility with a USB 2.0 device, the USB 3.0 may need to support BOT in addition to UAS (USB Attached SCSI).*
>
> *For USB2 backward compatibility, the device shall present BOT as the primary interface (i.e., Alternate interface 0) and UAS (USB Attached SCSI) as the secondary alternate interface (i.e., Alternate interface 1). However, a device that does not need backward compatibility with BOT shall only present UAS as alternate interface zero – in this case, there is no secondary Alternate interface. In USB 2.0 systems, the BOT driver or an associated filter driver may need to issue a SET_INTERFACE request for Alternate interface 1 and then allow the UAS driver to load. See Figure 3-19.*
>
> *The UASP uses the bInterfaceProtocol field to select between BOT and UAS.*

The Debug Class takes this approach a stage further and uses the bInterfaceSubClass to differentiate between DxC.GP, DxC.Dfx, and DxC.Trace, and uses the bInterfaceProtocol value to select variants within these capabilities.

Consequently, each Alternate interface in the Debug Class declares the same bInterfaceClass field, but a different pair of bInterfaceSubClass and bInterfaceProtocol values. The driver would be loaded based on the default 0th Alternate Interface. This version of the driver could then issue a SET_INTERFACE request to switch the endpoints owned by the interface to the alternate debug capability.

Thus, for example, if the 0th Alternate-Interface driver is DvC.Dfx, then it will have the capability to switch to another alternate interface driver such as DvC.Trace based on the bInterfaceSubClass field. See Figure 3-20.

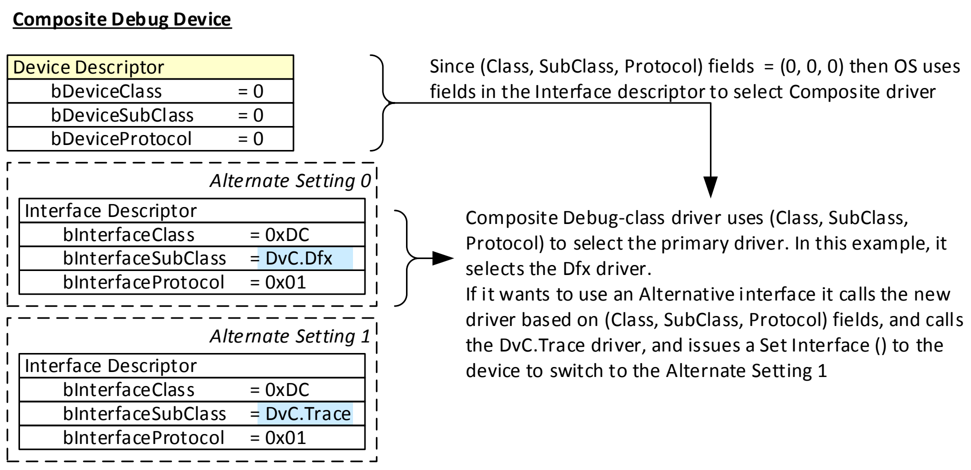

Composite Debug Device
Device Descriptor
bDeviceClass = 0
bDeviceSubClass = 0
bDeviceProtocol = 0
Since (Class, SubClass, Protocol) fields = (0, 0, 0) then OS uses fields in the Interface descriptor to select Composite driver
Alternate Setting 0
Interface Descriptor
bInterfaceClass = 0xDC
bInterfaceSubClass = DvC.Dfx
bInterfaceProtocol = 0x01
Composite Debug-class driver uses (Class, SubClass, Protocol) to select the primary driver. In this example, it selects the Dfx driver.
If it wants to use an Alternative interface it calls the new driver based on (Class, SubClass, Protocol) fields, and calls the DvC.Trace driver, and issues a Set Interface () to the device to switch to the Alternate Setting 1
Alternate Setting 1
Interface Descriptor
bInterfaceClass = 0xDC
bInterfaceSubClass = DvC.Trace
bInterfaceProtocol = 0x01

Figure 3-20: Composite Debug Class Device Example

3.6.2 Changing Debug Capabilities via Alternate Settings

An interface within a configuration may have alternate settings that redefine the number or characteristics of the associated endpoints. DbC and DvC can both use Alternate settings. For example, a TS may allocate one IN endpoint permanently to debug traces, and share a second pair of IN/OUT endpoints, mutually-exclusively between a GNU debugger (via DxC.GP) and a TAP interface (via DxC.Dfx). Thus, for the case of DvC, we have:

- Endpoint 1 ⇦ Debug Trace
- Endpoint 2 has two, mutually exclusive, alternate settings:
  1. Alternate Setting 0: Endpoint 2 ⬄ GNU debugger via DvC.GP
  2. Alternate Setting 1: Endpoint 2 ⬄ JTAG interface via DvC.Dfx

Figure 3-21 illustrates the descriptors for this example:

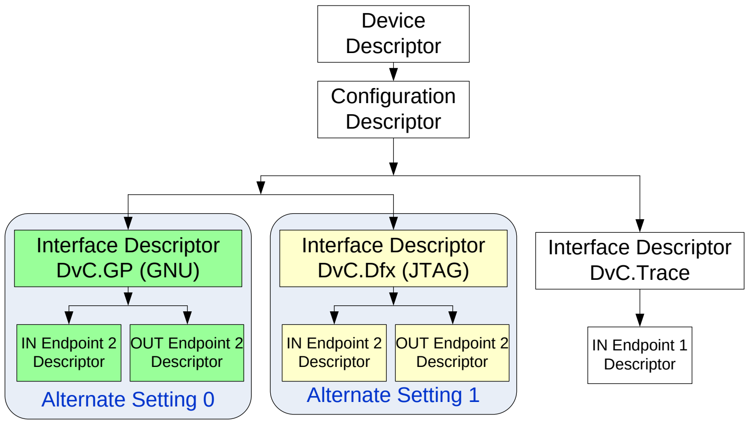


Figure 3-21: Alternate Settings Example

Switching between Alternate settings does not affect any other interface. Thus, the debug trace interface on endpoint 1 in Figure 3-21 is not affected by changing an alternate setting on endpoint 2.

3.6.3 Changing Debug Capabilities using Different Configurations

A debug device may use a different configuration instead of an alternate interface to share endpoints with other functions. Figure 3-22 gives an example. This is similar to the prior example of Figure 3-21, in the sense that one configuration provides Trace and a GNU debugger, while the other configuration provides Trace and JTAG. However, switching between these two configurations requires all endpoint traffic to stop before selecting the other configuration. Thus debug tracing will stop during the changeover, unlike the prior example using Alternate interfaces. Depending on the context, this may be a disadvantage. A further disadvantage is that not all Operating Systems support multiple configurations. Consequently, using different Configurations is not recommended.

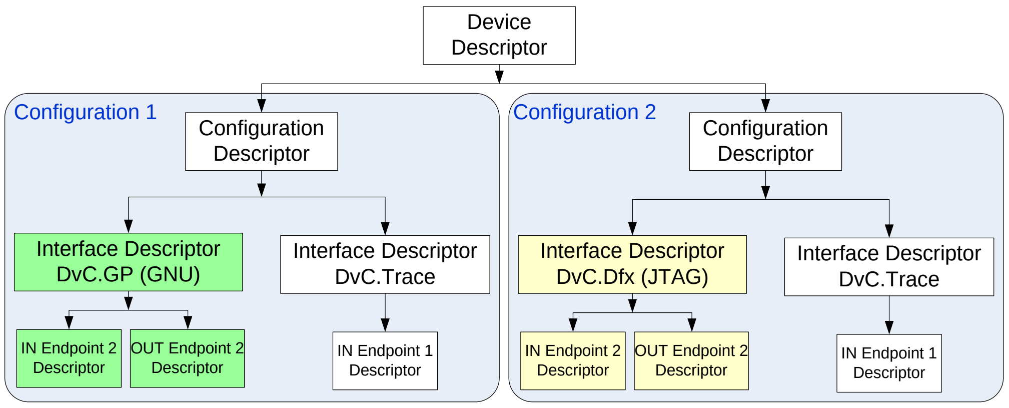


Figure 3-22: Example of two configurations

3.6.4 Interface Association Descriptor (IAD)

The USB 3.1 Debug Class supports Debug Control over the default endpoint 0. We need a mechanism to associate the debug control operations with a particular debug unit, a particular capability, or with the complete TS. For example, one may need to enable the voltage and clocks for a specific debug trace source, or for a set of trace sources, or indeed for all of the debug functions within a SoC.

In addition, we need a mechanism to group a number of functions together. For example, a debug tool may support both a kernel debugger and a stop-mode TAP debugger. It is not uncommon for a program under development to go “into the weeds” and hang. In this situation, the kernel debugger is useless and the TAP debugger is needed to break in and permit debug. Such a multi-function debugger could use the DxC.Dfx interface for the Stop-mode debugger, and the DxC.GP interface for the kernel debugger.

This specification uses the Interface Association Descriptor (IAD) [7] for this purpose, although other means could be used – refer to Section 3.6.6.

The Interface Association Descriptor (IAD) groups together two or more consecutive interfaces (and any alternate settings associated with these interfaces) into a single function. We call such a collection a Debug Interface Collection (DIC). See Figure 3-23.

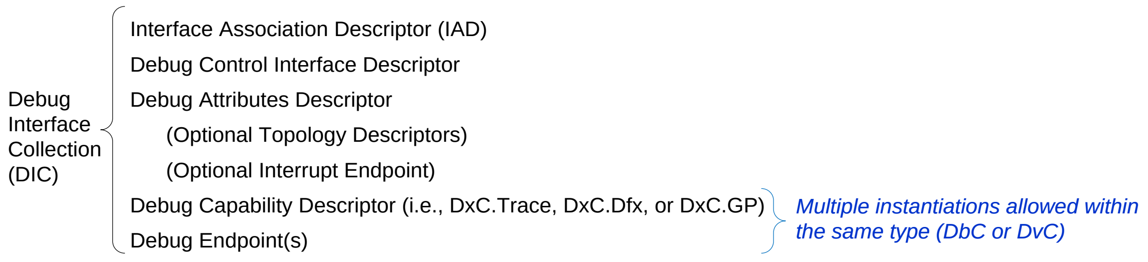

Debug Interface Collection (DIC)
Interface Association Descriptor (IAD)
Debug Control Interface Descriptor
Debug Attributes Descriptor
(Optional Topology Descriptors)
(Optional Interrupt Endpoint)
Debug Capability Descriptor (i.e., DxC.Trace, DxC.Dfx, or DxC.GP)
Debug Endpoint(s)
Multiple instantiations allowed within the same type (DbC or DvC)

Figure 3-23: Debug Interface Collection

A DIC consists of four components:

1) The optional Debug Control requests and other features supported by the debug function. These are defined by two descriptors:
    a. The Debug-Control Interface descriptor, which is a standard USB interface descriptor that characterizes the interface itself
    b. The Debug-Attributes descriptor, which is in essence an extension of the Debug-Control descriptor and provides specific details of the features supported by the debug function.
2) Optional topology information describing how the various debug units within the function interconnect.
3) An optional Interrupt endpoint for the control capability. This could, for example, be used to enable trace capture by the host when the smartphones screen is touched.
4) The debug capability or capabilities supported by the debug function (e.g., DvC.Trace) together with their endpoints.

An IAD requires >1 interface, and thus a DIC must have two or more interface descriptors (e.g., Debug Control and DvC.Trace).

The operating system calls a single composite driver per DIC, which will then call the appropriate drivers for the associated debug functions (e.g., Dfx TAP access, trace capture, etc.).

A device shall use an Interface Association Descriptor to describe a Debug Interface Collection for each device function that requires a Debug-Control Interface and one or more Debug Capability interfaces.

To help understand IADs and DICs, we will examine a few scenarios. Suppose there are two debug functions using the DvC.Dfx and DvC.Trace interfaces, and that we have two separate debug tools for these functions. It thus makes most sense to group each capability within a DIC using an IAD, as this will ensure that the host calls a debug driver for each DIC. Consequently, each of the separate debug tools will have their own driver. Figure 3-24 shows such an example, where two IADs create two DICs. One

DIC consists of a Debug-Control Interface and a DvC.Dfx interface, while the other consists of a Debug-Control Interface and a DvC.Trace interface. They are labeled as IAD 1 and IAD 2, and DIC 1 and DIC 2.

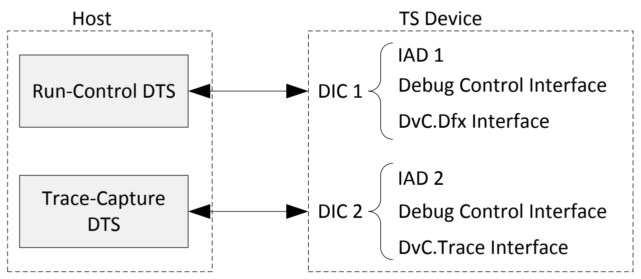


Figure 3-24: Example showing two IADs grouping the Control with the appropriate debug interface

Figure 3-25 shows an alternative grouping of the debug interfaces of Figure 3-24, where we use a single IAD to group the trace and Dfx functions together. This makes most sense when we have a multi-function debug tool that, for example, supports TAP and Trace. In this case, the IAD groups the DvC.Trace interface with the DvC.Dfx run-control interface, so that the host evokes a single driver for the multi-function debug tool.

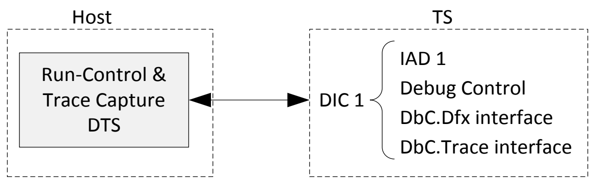


Figure 3-25: Example showing a single IAD grouping the control for a DvC.Dfx and DvC.Trace

Finally, Figure 3-26 shows an example of a TS containing a Graphics unit, a main core, and a modem, each with their own independent Dfx and trace capabilities. This scenario may occur when different IP blocks are used in the implementation of the SoC, and each has its own dedicated debug tool.

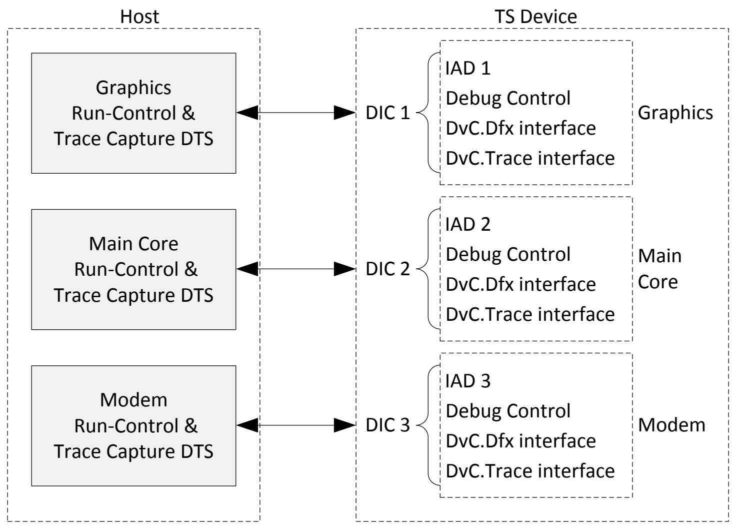


Figure 3-26: TS device with three DICs

Note that IADs do not support nesting, and thus nesting multiple DICs via an IAD is not possible. Thus, in Figure 3-26 we cannot have an IAD4 that groups IAD1, IAD2, and IAD3 together to feed a single driver.

One may now wonder how one decides between these various options. One possibility, in an Android-based TS, is to use adb to create the appropriate descriptors, which become active and persistent in the next reboot of the system. Thus, for example, one could default to the scheme depicted in Figure 3-24, and then later change the descriptors to the scheme shown in Figure 3-25 via adb. Alternatively, for an implementation with permanent descriptors cast in ROM, one could use the SET_ALT_STACK command (See 5.4.8) to access alternative descriptors. Thus one could provide basic, default descriptors in hardware that become active at reset, and then use more extensive descriptors later after the OS has booted. This will allow basic debug prior to the OS boot and extensive debug thereafter.

3.6.5 Multiple Mutually-Exclusive Host Drivers

An SoC may consist of multiple different IPs from different IP vendors (e.g., audio, graphics, modem, etc.). Each IP vendor may provide a dedicated debug tool for only their IP, and no other. Consequently, debugging such an SoC requires multiple debuggers. For convenience, it is desirable to have these debug tools all installed on the same host platform, so that during a debug session one can switch between the tools quickly. This requires that the multiple drivers serving the various debug tools are all resident on the host.

Figure 3-27 shows one possible scheme for how two debuggers can mutually-exclusively access a single TAP debug function. It provides a separate DIC for each debugger. Each DIC evokes a separate driver on the host, thus allowing the drivers for the different tools to be both resident on the host at the same time. Section 4.3.1.6 describes how this can be achieved by using different values for the bInterfaceProtocol field of the interface descriptors.

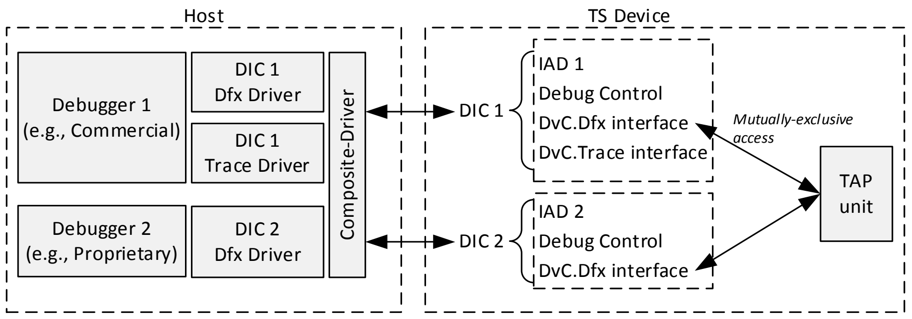


Figure 3-27: Multiple debuggers accessing common Debug-Function Example

Note that how the TAP unit is shared by the two debuggers is implementation specific and thus beyond the scope of this document. Most TAP debug tools require complete ownership of the TAP network, and thus cannot share the TAP network with another tool. In order to share a debug resource, some high-level arbitration and lock mechanism is necessary. The USB 3.1 Debug Class specification provides a mechanism to share the USB interfaces and functions, but how the tools actually share these resources is beyond the scope of a USB class specification.

However, the scheme shown in Figure 3-27 requires endpoints for each Dfx interface, and is thus wasteful. It would be better if the debuggers could share the same endpoints, since typically, debug resources are restricted. The following, alternative scheme, of using Alternate settings is more efficient:

- Interface 1: Debug Trace
- Interface 2 has two, mutually exclusive, alternate settings:
    1. Alternate Setting 0: DvC.Dfx for Debugger 1
    2. Alternate Setting 1: DvC.Dfx for Debugger 2

Figure 3-28 illustrates the descriptors for this example. The default is Alternate Setting 0, and this driver (or an associated filter driver) can issue a SET_INTERFACE request for Alternate Interface 1 and then allow the driver for Debugger 2 to load. Similarly, debugger 2 can issue a SET_INTERFACE request for Alternate Interface 0 and then allow the driver for Debugger 1 to reload.

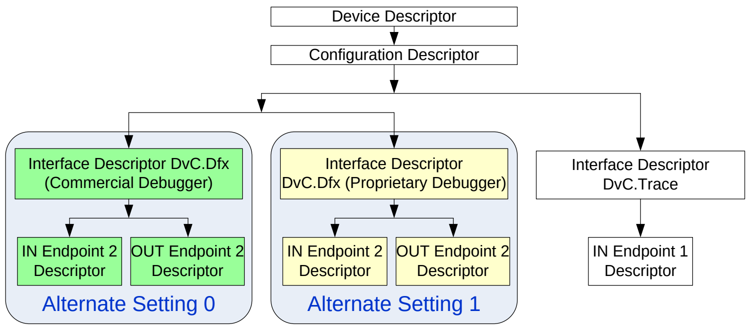

Device Descriptor
Configuration Descriptor
Interface Descriptor DvC.Dfx (Commercial Debugger)
IN Endpoint 2 Descriptor
OUT Endpoint 2 Descriptor
Alternate Setting 0
Interface Descriptor DvC.Dfx (Proprietary Debugger)
IN Endpoint 2 Descriptor
OUT Endpoint 2 Descriptor
Alternate Setting 1
Interface Descriptor DvC.Trace
IN Endpoint 1 Descriptor

Figure 3-28: Alternate Settings used to select between multiple Debuggers on the same Endpoints.

The above mechanism allows one to switch between different debug tools and debug functions within the TS. However, this in itself is insufficient: we also need to place the debug function into a known state after each SET_INTERFACE request, which is described in the next section.

3.6.5.1 Initializing the Debug Function prior to Debugger changeover

The Network Control Model (NCM) devices class [8] solves the initialization problem described above by using Alternate settings to place the network aspects of a function in a known state. Essentially, Alternate setting 0 is purely a “reset/init” setting, while Alternate Setting 1 is the operational setting. Thus, toggling between Alternate Setting 0 and 1 will reset the network. In addition, NCM uses commands to set parameters before switching to the operational setting, so that the network initializes to a known, desired state.

The USB 3.1 Debug Class uses a similar approach:

- Alt Setting 0: Master Debugger. This Master could be an “inert” driver that provides no debug capability and is simply a mechanism to switch between tools. Alternatively, it could provide debugger functionality.
- Before switching to another debugger, the Master first issues debug commands to init/reset the desired debug function(s) to a known state. Ideally, the TS should provide a service, via the Set Service debug command to initialize the function.
- Alt Setting >0 for the Slave Debuggers

Figure 3-29 gives an example. Alternate Setting 0 evokes a Master application on the host that, in this example, uses the DvC.GP interface to access a Debug driver on the TS. This driver provides a service to initialize the Dfx function.

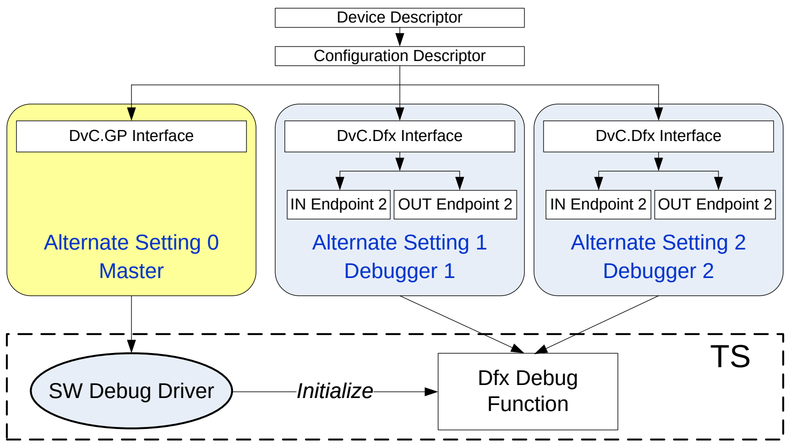


Figure 3-29: Master Switch with two Slave Debuggers

The Master in this example provides no Dfx capability – the two slave debuggers on Alternate Settings 1 and 2 provide that support. On enumeration, the host evokes the Master. The debug user can now select one of the two debuggers. The Master achieves this by issuing a Set Alternate standard command to the device and then initializing the Dfx unit before evoking the driver corresponding to the new Alternative setting.

Later, to switch in a different debugger, it repeats the sequence:

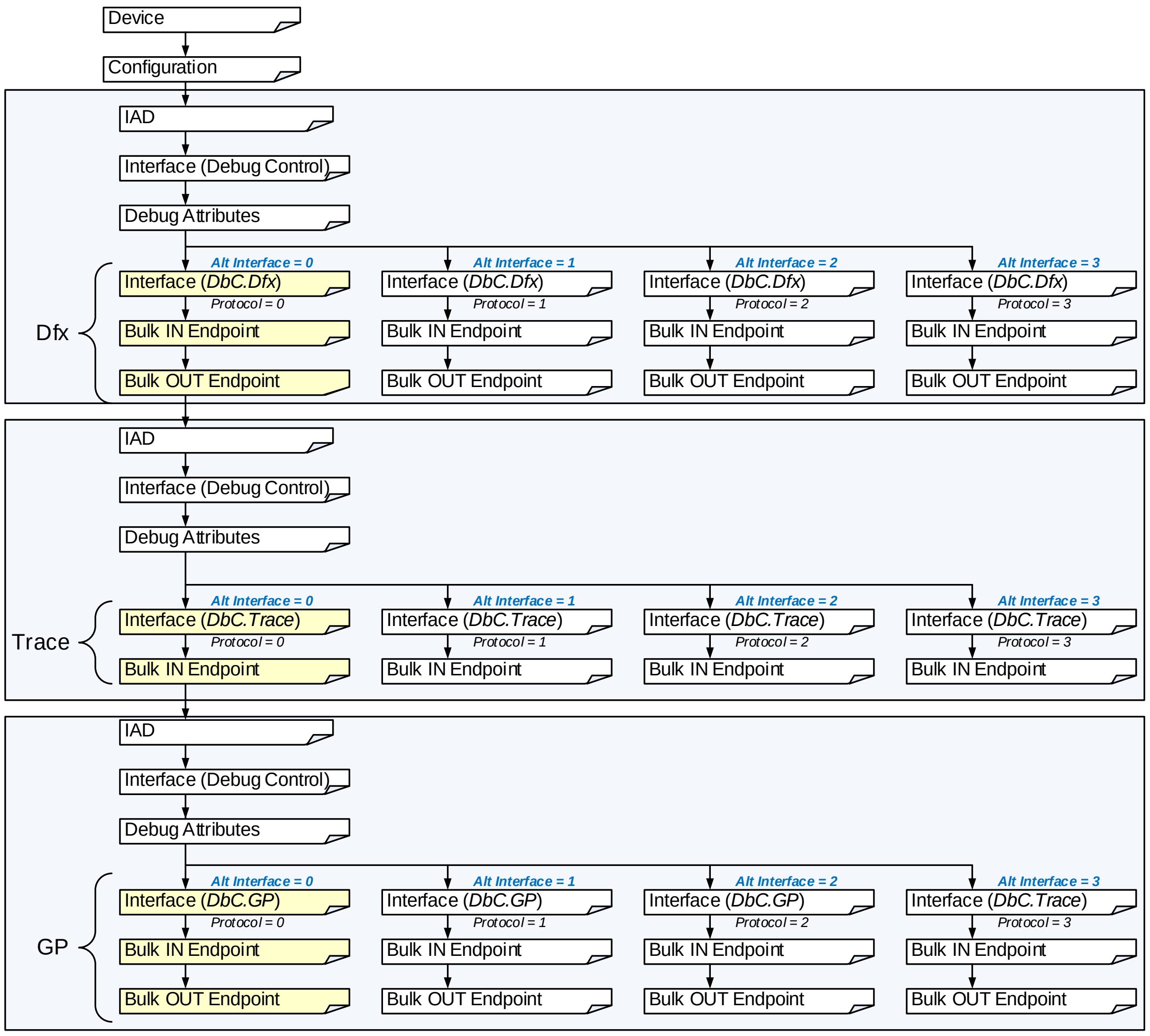
- Alt Setting 0
- Debug command to init/reset the new desired debug function(s)
- Issues Set Alt Setting ≠0 to the device and instantiates the new driver on the host for the new debugger application.

The Master in Figure 3-29 could have been an actual Dfx debugger, in which case it would have used the DvC.Dfx capability instead of the DvC.GP capability.

Note that although the USB 3.1 Debug Class provides a mechanism to switch between debug tools via a Master, it is beyond the scope of a USB Class specification to specify such a Master. This requires a new standard to define the necessary services for the various industry tools.

Figure 3-30 shows a more elaborate descriptor example that supports four different debug tools per debug capability. For example, a lab environment may use a proprietary debug tool for the DxC.Trace, DxC.Dfx, and DxC.GP capabilities. However, occasionally the lab debugger may need to use a Commercial tool for some or all of the debug capabilities. They would then use the Set Alternate capability to switch in the alternative tool.

Note that the TS does not know or care which tools are on the host – it simply provides the capability for the host to have up to 4 tools installed. Different users will install different tools. For example, the modem debug team may wish to have their modem debug tool as the primary, default tool for DxC.Trace, DxC.Dfx, and DxC.GP. Occasionally, the bug may require debugging different portions of the TS, and then they may switch to a Core or Audio debug tool. These secondary tools may also provide access to the DxC.Trace, DxC.Dfx, and DxC.GP capabilities.



Example Use Scenarios

bInterfaceProtocol	0	1	2	3
Scenario 0	Proprietary Debug Tool	Commercial Debug tool	*Not Used*	*Not Used*
Scenario 1	Commercial Debug tool	*Not Used*	*Not Used*	*Not Used*
Scenario 2	Modem Debug Tool	Commercial Debug tool	*Not Used*	*Not Used*
Scenario 3	Graphics Debug Tool	Commercial Debug tool	Proprietary Debug Tool	*Not Used*
Scenario 4	Android Debug tool suite	Sensor-Hub Debug tool	Proprietary Debug Tool	Commercial Debug tool

Figure 3-30: Example showing support for Multiple Debug tools

Figure 3-30 shows a few example scenarios:

- Scenario 0: Lab based debug using a proprietary debug tool for the DxC.Dfx, DxC.Trace and DxC.GP capabilities, which has access to proprietary debug logic within the TS. This tool could have extensive access to the primary core. Occasionally, the debugger switches to a Commercially-available tool for the DxC.Dfx, DxC.Trace and DxC.GP capabilities, because this tool provides better access to the remainder of the SoC. For example, the proprietary tool may only support traces and Dfx with the main core, while the commercial tool captures traces and provides Dfx support with all of the debug units within the SoC.

- Scenario 1: Customer debug using only a commercial debug tool
- Scenario 2: Modem debug lab with the Modem debug tools accessing the Dfx, Trace and GP capabilities of the Modem logic. Occasionally, the debugger switches to a Commercially-available tool for the DxC.Dfx, DxC.Trace and DxC.GP capabilities of the APE, when the bug appears to extend beyond the modem logic.
- Scenario 4: General debug of the OS and applications using Android and SW tracing tools. Occasionally, the debugger may need to debug other issues such as the sensor hub.

3.6.6 Enumerating Interface Collections

Different USB device classes have developed different means of enumerating interface collections:

- Vendor-supplied callback routines
- Union-Function descriptors (UFD). This method is used by the Wireless Communication device class
- Interface Association Descriptor
- Legacy Audio method

These, sometimes incompatible mechanisms, have led to complexities in current smartphones that integrate a number of classes. For example, a smartphone device could integrate debug and wireless-communication devices together, where the Debug Class uses IADs and the latter uses UFDs. These two mechanisms cannot be intermixed, and instead IADs or UADs have to be used for all devices. This is a known issue for device-driver developers, and is the same problem as trying to integrate Video with a wireless-communication device in a smartphone (this is because the Debug class uses the same IAD mechanism as the Video class or the AV class).

In addition, the support for UFD’s and IAD’s is inconsistent across operating systems, and it is not unusual for the device to provide numerous different descriptor solutions for the various possible host operating systems (some current generation smartphones provide 9 or more options). This Debug Class specification specifies IADs, but an actual driver on a device is free to use some other method to enumerate interface collections. This is outside the scope of this document.

3.6.7 Debug-Control Interface

As explained in Section 3.5.4, there are a number of possible mechanism for controlling the debug units, which are all optional:

1. DxC.Dfx interface using TAP or some other mechanism
2. DxC.GP using a software driver that access the debug-units configuration registers via MMIO
3. Debug-Control Interface via the default endpoint 0 control endpoints.

A TS need not provide any means of controlling the debug functions via the USB. For example, a shipping device may always provide default trace capability that is always enabled once configured.

This section describes the Debug Control method, which has the capability to control any particular or groups of units within a debug function or multi-function. For example, one has the ability to enable the power for the complete TS, or a particular DIC, or a particular unit or units within a DIC. To make these objects accessible, the debug function shall expose a single Debug-Control Interface. This interface can contain the following endpoints:

- A Control endpoint for manipulating TS, DIC, and Unit settings and retrieving the state of the debug function (for example, the debug state at the end of a debug session). This endpoint is mandatory, and the default endpoint 0 is used for this purpose.
- An interrupt IN endpoint for status returns (for example, when a debug breakpoint fired). This endpoint is optional.

A device shall use an Interface Association Descriptor to describe a Debug Interface Collection for each device function that requires a Debug-Control Interface and one or more Debug interfaces. (This requirement is necessary to satisfy the IAD specification, which requires >1 interface).

The Interface Association Descriptor shall always be returned as part of the device's complete configuration descriptor in response to a GET_DESCRIPTOR (Configuration) request. The Interface Association Descriptor shall be located before the Debug-Control Interface and its associated Debug Interface (including all alternate settings). All of the interface numbers in the set of associated interfaces shall be contiguous (there can be no gaps in the list of interface numbers).

The Debug-Control interface is the single entry point to access the internals of the debug function. Thus, any Debug-Control request for the DIC or a Unit within the DIC shall be directed to the Debug-Control interface of the debug function. Figure 3-31 provides an example. The Debug-Control interface is the single entry point for the request, and thus the request is shown targeting interface1. The figure shows a read request (i.e., GET_CONFIG_DATA) of a global configuration register in the SoC (i.e., the complete TS). This SoC contains many different registers, shown labelled as Configuration, Power-management, etc. Some of these registers pertain to the complete TS, some to the complete DIC, and some to a particular Unit. This example is targeting the complete TS because the wValue field is 0 (see Section 5.3.2 "Request Examples" for more information). The wValue and wIndex fields of the Debug Request are used to direct the command to control the complete TS, or the complete DIC, or a specific Unit within a DIC.

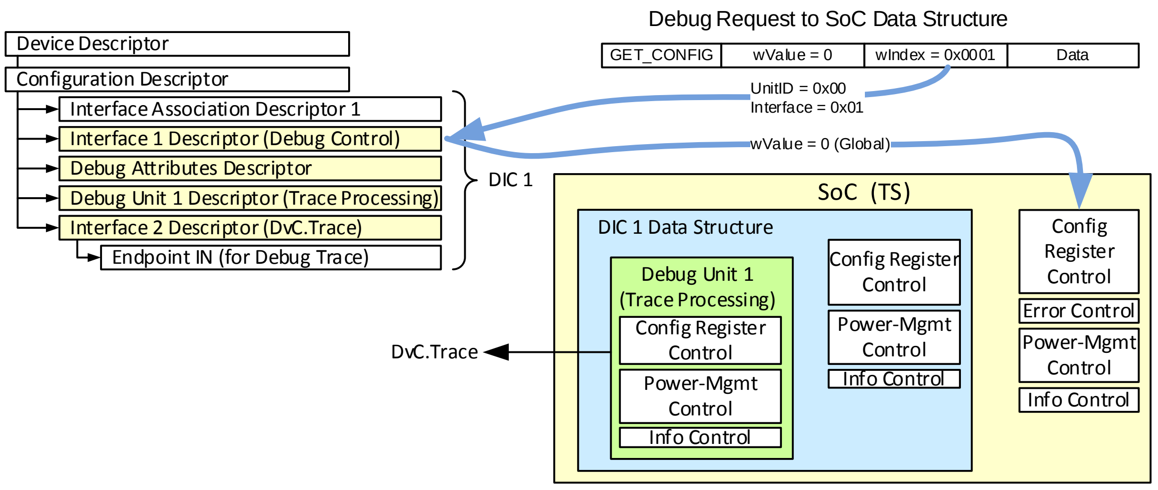


Figure 3-31: Example of a Debug Control targeting the Global Configuration Register Control

Table 3-3 lists the available Debug requests.

Table 3-3: Supported Debug Commands

Request	Description
GET/SET_CONFIG_DATA, SET_CONFIG_DATA_SINGLE	This requests reads or writes the configuration registers in the TS, DIC, or specific Unit (e.g., Trace-Processing unit)
GET/SET_CONFIG_ADDRESS	This request reads or writes the Address used by the GET or SET_CONFIG_DATA commands
GET/ SET_ALT_STACK	An SoC contains multiple cores, and any number of these could support the USB stack. In addition, debug may have a special hardware stack for this purpose. This command allows the host to select an alternative core or hardware for the USB stack support. The GET command also returns status on when the OS has booted so that the host knows when it can use the normal OS USB stack.
GET/SET_OPERATING_MODE	This request reads or configures the power-management mode for the TS, DIC, or specific Unit. For example, it can place the device into a debug power-mode.
GET/SET_TRACE	This requests sets or reads the vendor-specific trace configuration. The vendor can define one of 255 possible trace configurations. For example, Trace Configuration 1 may enable all traces within the TS, while Trace configuration 2 only enables the modem traces. This register is not a bit mask but a number corresponding to a set of enabled traces.
SET_BUFFER	This command performs basic operations on a trace buffer (e.g., flush, initialize).
GET_BUFFER	This command reads the buffer size for the TS, DIC, or specific Unit (e.g., Trace-Processing unit)
SET_RESET	This command resets the TS, DIC, or debug unit to its default state. This is useful if the debug logic has hung.
GET_INFO	This provides general information on the capabilities and support for the various Debug Class commands in the TS, DIC, or specific unit. Note that this is not information pertaining to the Debug

	Function as such, but information pertaining to the supported debug commands and their capabilities.
GET_ERROR	This reports status on a debug request (e.g., success, fail, etc). This does not contain error information pertaining to a debug operation per se, but rather to the debug control requests.

Full descriptions of the Debug Control requests are given in Section 5.

3.6.7.1 Control Endpoint

The Debug Class uses endpoint 0 (the default pipe) as the standard way to control the debug function using class-specific requests. These requests are directed to the complete TS, to a DIC, or to a Unit within a Debug Function. The format and contents of these requests are detailed in Section 5.

USB Control transfers minimally have two transaction stages: *Setup* and *Status*. A control transfer may optionally contain a *Data* stage between the *Setup* and *Status* stages (see Figure 3-32 for an example). The Setup stage contains all information necessary to address a particular entity, specify the desired operation, and prepare for an optional Data stage. A Data stage can be host-to-device (OUT transactions), or device-to-host (IN transactions), depending on the direction and operation specified in the Setup stage via the bmRequestType and bRequest fields.

In the context of the Debug Class specification, SET_* requests will always involve a Data stage from host to device, and GET_* requests will always involve a Data stage from device to host.

The device shall use Protocol stall (and not Function stall) during the Data or Status stages if the device is unable to complete the Control transfer. The reasons for using Protocol stall include unsupported operations, invalid target entity, unexpected Data length, or invalid Data content. The device shall update the value of Request Error Control, and the host may use that control to determine the reason for the Protocol stall (see Section 5.4.13 "GET_ERROR"). The device shall not NAK, NRDY, or STALL the Setup transaction.

Typically, the host will serialize Control Transfers, which means that the next Setup stage will not begin until the previous Status stage has completed. However, in situations where the Setup transaction is sent before the completion of the previous control transfer, then the device shall abandon the previous control transfer.

Due to this command serialization, it is important that the duration of control transfers (from Setup stage through Status stage) be kept as short as possible. For this reason, as well as the desire to avoid polling for device status, this specification defines an interrupt status mechanism to convey status changes independently of the control transfers that caused the state change. This mechanism is described in Section 3.6.7.2, "Status Interrupt Endpoint". Any control that requires more than 10ms to respond to a SET_* request (referred to as "*Slow control*"), or that can change independently of any external SET_* request ("*Self-Generated control*"), shall send a Control-Change status interrupt. These characteristics will be reflected in the GET_INFO response for that control.

If a SET _* request is issued to a Slow Control (i.e., >10ms slow response) with unsupported operations, invalid target entity, unexpected data Length or invalid data content, the device shall use Protocol stall since the device is unable to complete the Control transfer. The device shall update the value of the Request Error Control (see Section 5.4.13 "GET_ERROR").

In the case of a SET_* request with valid parameters to an Slow Control, the Control transfer operation shall enter the Status stage immediately after receiving the data transferred during the Data stage. Once the Status stage has successfully completed, the device shall eventually send a Control Change Interrupt that will reflect the outcome of the request:

- If the request succeeded, the Control Change Interrupt will advertise the new value (see in 3.6.7.2, "Status Interrupt Endpoint").
- If the request could not be executed, the device shall send a Control Change Interrupt using the Control Failure Change mechanism to describe the reason for the failure

The amount of time between the end of a successful Status stage and the Control-Change interrupt is implementation specific. For instance, a transition from a normal (non-debug), power state to a power mode that powers up the debug logic may take hundreds of milliseconds.

The following flow diagrams show the Setup, Data and Status stages of SET_OPERATING Control Transfers. The example on the left successfully completes within 10ms. The example on the right takes longer, and thus the device issues a Control-Change Interrupt as soon as the operation completes.

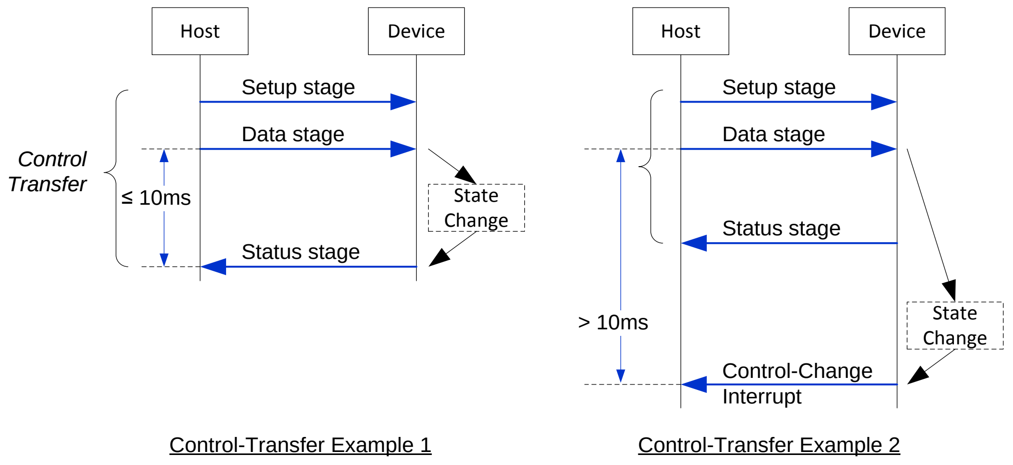


Figure 3-32: Control Transfer Examples

3.6.7.2 Status Interrupt Endpoint

The Debug-Control interface can support an optional IN interrupt endpoint to inform the Host about the status of the different addressable entities (units and interfaces) inside the debug function. The interrupt endpoint, if present, is used by the entire Debug Interface Collection to convey status information to the Host. It is considered part of the Debug-Control interface because this is the anchor interface for the DIC.

Possible uses for the interrupt endpoint include:

- The device supports debug breakpoints or debug events (e.g., pressing a virtual button on a smartphone screen may start a debug trace). Hardware debug breakpoints of the main core are difficult to support via the Interrupt breakpoint, because the debug breakpoint may halt the core preventing the USB handler from supporting the Interrupt transaction. However, an implementation may choose to use a secondary core in an SoC, which is not being utilized by the application being debugged, to provide the necessary support for the Interrupt transaction.
- The device implements any *Self-Generated controls* (controls supporting device initiated changes). For example, a debug device may automatically change a trace source upon some event (e.g., low battery, flight-mode, GPS was enabled, the core powered down, etc.), or it may dynamically request more available trace bandwidth (e.g., instructing the host debugger to turn off some other debug sources) based on new circumstances (e.g., graphics rendering started for WebGL content).
- The device implements any *Slow* controls (i.e., the device requires more than 10ms to respond to a debug Control request).

The interrupt packet is a variable-size data structure depending on the originator of the interrupt status. The bStatusType and the bOriginator fields contain information about the originator of the interrupt. The bEvent field contains information about the event triggering the interrupt. If the originator is the Debug-Control Interface, the bSelector field reports which control issued the interrupt (e.g., Config, Power, etc.)

Any addressable entity inside a debug function can be the originator of an interrupt.

The contents of the bOriginator field shall be interpreted according to the code in the bits <3:0> of the bStatusType field. If the originator is the Debug-Control Interface, the bOriginator field contains the Unit ID of the entity that caused the interrupt to occur. A bOriginator field set to zero indicates the *virtual entity* interface, which is used to report global Debug-Control Interface changes to the Host. This scheme is unambiguous because Units are not allowed to have an ID of zero. If the originator is any debug capability DxC.Dfx, DxC.GP, or DxC.Trace interface), then the bOriginator field contains the interface number of this interface.

If the originator is the Debug-Control Interface, the bAttribute field indicates the type of Control change.

The contents of the bEvent field shall also be interpreted according to the code in bStatusType<3:0>. If the originator is a DxC.GP, DxC.Dfx or DxC.Trace interface, there are additional debug events as defined in the table below – e.g., pressing a virtual button on a Smartphone may start trace generation.

For all originators, there is a Control-Change event defined. Controls that support this event will trigger an interrupt when a host-initiated or externally-initiated control change occurs. The interrupt shall only be sent when the operation corresponding to the control changes is completed by the device.

A control shall support Control-Change events if any of the following is true:

- The Control state can be changed independently of the host control (e.g., on the Smartphone, a virtual button disables a debug-power mode).
- The Control can take longer than 10ms from the start of the Data stage through the completion of the Status stage when transferring to the device (i.e., for a SET_* operation)

If a control is required to support Control-Change events, the event shall be sent for all SET_* operations, even if the operation can be completed within the 10ms limit (and thus appears to be unnecessary). The device indicates support for Control-Change events for any particular control via the GET_INFO attribute.

Table 3-4, Table 3-5, and Table 3-6 specify the format of the Status packet

Table 3-4: Status Packet Format

Offset	Field	Size (Bytes)	Value	Description
0	bStatusType	1	BitMap/ Number	<3:0>: Originator <0>: Debug-Control Interface <1>: Any of the possible DxC.Dfx interfaces <2>: Any of the possible DxC.Trace interface <3>: Any of the possible DxC.GP Trace interface <7:4>: *reserved*
1	bOriginator	1	Number	Unit ID or interface that is reporting the interrupt

When the originator is a Debug-Control Interface, the rest of the structure is:

Table 3-5: Status Packet Format (Debug-Control Interface as Originator)

Offset	Field	Size (Bytes)	Value	Description
2	bEvent	1	Number	0x00: Control Change *Otherwise: reserved*

3	bAttribute	1	Number	Specify the type of control change: 0x00: Control Value change 0x01: Control Info change 0x02: Control failure change *Otherwise: reserved*
5	bValue	1	Number	**bAttribute: Description** 0x00 Equivalent to GET_CCONFIG_DATA 0x01 Equivalent to GET_INFO 0x02 Equivalent to GET_ERROR *Otherwise: reserved*

When the originator is a DxC.Dfx, DxC.GP, or DxC.Trace interface, then the remainder of the structure is:

Table 3-6: Status Packet Format (DxC.Dfx, DxC.GP, or DxC.Trace as Originator)

Offset	Field	Size (Bytes)	Value	Description
2	bEvent	1	BitMap/ Number	All originators: <3:0>: *reserved* <7:4>: Vendor specific
3	bValue	1	Number	Debug Event: 0x00: Debug “Button” released 0x01: Debug “Button” pressed *Otherwise: reserved*

3.6.7.3 Hardware Trigger Interrupts

One possible usage of the Status-Interrupt endpoint is for hardware triggers to notify host software that a debug breakpoint/event occurred. A breakpoint could occur on an instruction match, data match, debug button press, etc. When the hardware detects a debug event, the Status-Interrupt endpoint will generate an interrupt originating from the relevant DxC.Dfx, DxC.GP, or DxC.Trace interface. The event triggering the interrupt (button press or release) is indicated in the interrupt packet. The default, initial state of the button is the "release" state.

The device specifies whether it supports hardware triggers, and how the host software should respond to hardware-trigger events. These are specified in the class-specific Debug-Attributes descriptor (i.e., bmSupportedFeatures field) within the relevant DxC.Dfx, DxC.GP, or DxC.Trace interface. See Section 4, "Descriptors".

3.6.8 DxC.Trace Interface

The DxC.Trace interface sends debug traces from the trace function to the Host. It is optional. A debug function can have zero or more DxC.Trace interfaces associated with it, each possibly carrying data of a different nature and format. Each DxC.Trace interface can have one isochronous or Bulk IN data endpoint for the data trace. Appendix C: describes a possible data format for the traces.

There are a number of debug trace scenarios that require use of Alternate settings. These are:

- Case1: To select different bandwidths for isochronous traces. This is the typical usage case for alternate settings (e.g., video streaming in the Video class)
    - In this case, the bInterfaceClass = DxC.Trace, and the bInterfaceProtocol are the same value in all of the Alternate settings
- Case 2: To select between different drivers for trace capture. For example, during a debug session, the user may wish to rapidly switch between a commercial debug tool and a proprietary tool, where each provides a different set of capabilities. See Section 3.6.5 for more information on multiple, mutually-exclusive drivers.
    - In this case, the bInterfaceClass = DxC.Trace, and the bInterfaceProtocol will be a different value for the different Alternate settings. Each Alternate setting will thus evoke a different driver
- Case 3: This is similar to case 2, but in this case, it selects between alternate debug capabilities. A TS may have a restricted set of endpoints available for debug usages, and uses alternate settings to share these endpoints amongst different tools in a mutually-exclusive manner.
    - In this case, the bInterfaceClass will be a different value for each of the different Alternate Settings (e.g., DbC.Trace and DbC.Dfx).

These options are not mutually exclusive, and may be combined. We give an example later (see Figure 3-35).

3.6.8.1 Alternate Settings – Case 1

Case 1 is when the Alternate setting selects between various different isochronous bandwidths.

An isochronous interface provides guaranteed bandwidth. However, a host may not be able to satisfy the requested bandwidth if it has already allocated bandwidth to another isochronous interface. For this reason, an isochronous interface needs to provide a set of bandwidth requirements (e.g., 50MB, 100MB, & 200MB) to allow the host application the flexibility to select the next-best bandwidth option.

Thus, the rule is: A DvC.Trace interface with isochronous endpoints shall have alternate settings, which the host can use to change the bandwidth requirements that an active isochronous pipe imposes on the USB.

In addition, such an endpoint shall incorporate a zero-bandwidth, default alternate setting (alternate setting zero)[2]. This setting gives the host software the option to temporarily relinquish USB bandwidth by switching to this alternate setting if required. For example, this may occur if a video application requires isochronous transfers and there is insufficient link bandwidth for both the current debug isochronous traffic and the video traffic to run concurrently. The zero-bandwidth, alternate setting for the isochronous interface shall not contain a non-zero bandwidth DvC.Trace isochronous data endpoint descriptor[3].

Figure 3-33 shows a possible example.

[2] A Bulk endpoint is acceptable as a zero-bandwidth alternate setting. That is, the zero bandwidth setting is implied by the omission of an isochronous endpoint for alternate 0

[3] This statement only applies if one uses a zero-bandwidth isochronous endpoint instead of a Bulk endpoint for the zero bandwidth setting.

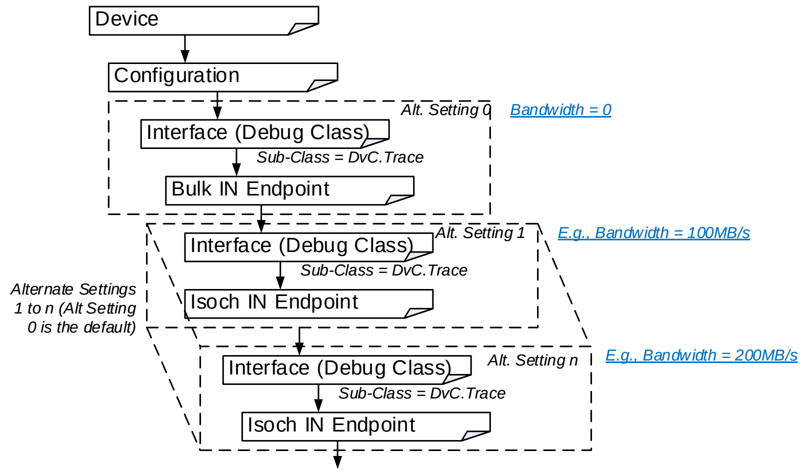


Figure 3-33: Example of DvC.Trace Descriptors

Debug traces vary significantly in bandwidth requirements. For instance, software messages (e.g., printf-type messages) typically require low bandwidth (2-30MB/s), whereas hardware traces from bus watchers, and processor-instruction traces can consume considerable bandwidth (800MB/s or more). During a debug session, the debugger may have configured the device to only send out software messages, hardware messages, or both. Consequently, an isochronous DvC.Trace interface should support a range (greater than two) of alternate interface settings with varying bandwidths. By doing so, the host would be able to select an appropriate alternate setting for a given debug-trace scenario that best utilizes the bus bandwidth.

3.6.8.1 Alternate Settings – Case 2 and Case 3

Case 2 is when the Alternate setting selects between a number of different Debug trace drivers. Case 3 is similar, except that now the alternate setting selects between different debug capabilities (e.g., between DxC.Trace and DxC.Dfx). Either of these cases allows the sharing of different debug tools (including their drivers) across the same endpoints. Figure 3-34 shows the endpoints being shared across trace and Dfx.

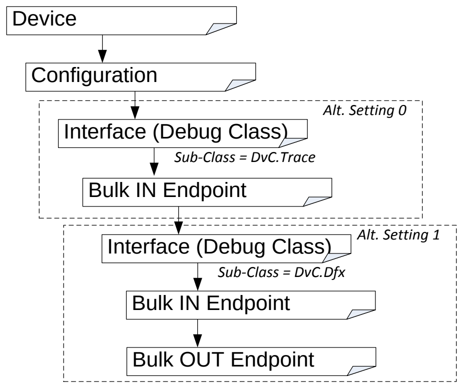


Figure 3-34: Example of Alternative setting for DvC.Trace and DvC.Dfx

Figure 3-35 is a more extensive example that uses Cases 1, 2, and 3. The first set of Alternate settings evoke host driver A. This is for an isochronous trace and is an example of Case 1. The interfaces for this case have bInterfaceSubClass = DxC.Trace and bInterfaceProtocol = 0.

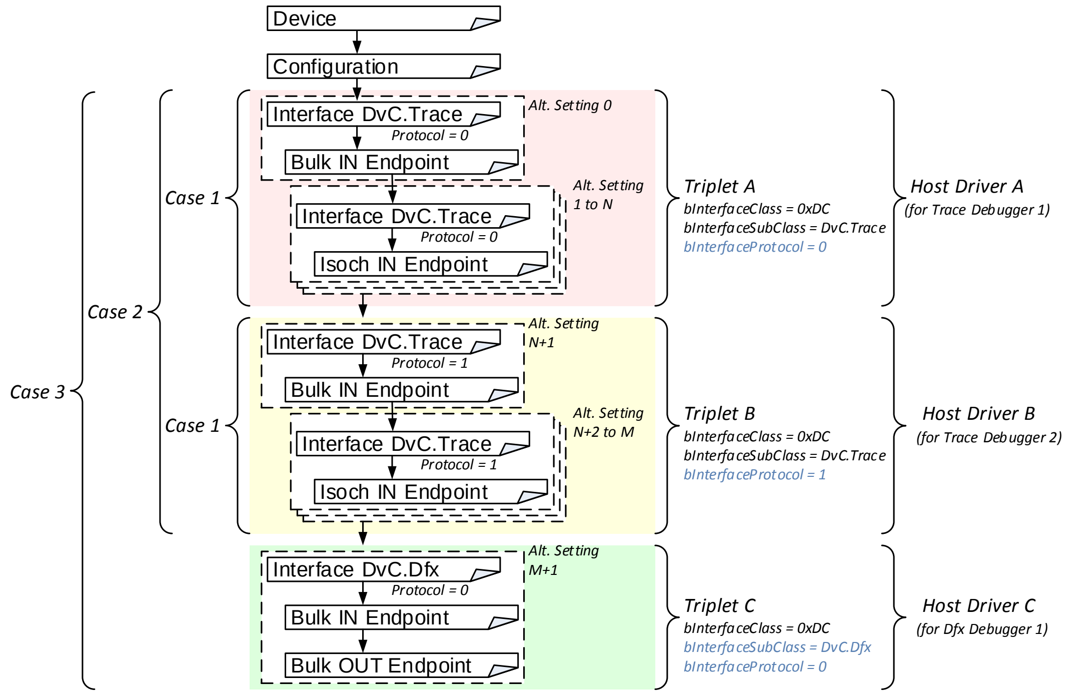


Figure 3-35: DvC.Trace Example using multiple different types of Alternate settings

The second set of Alternate settings in Figure 3-35 evoke host driver B. The interfaces in this example are also for an isochronous trace, but this time they evoke a different trace driver. Thus, these two sets of interfaces (corresponding to Triplets A and B) are examples of Case 2. The interfaces for these two sets of interfaces have different values for bInterfaceProtocol, as highlighted in Figure 3-35. Note that within each of these two sets of interfaces we have examples of Case 1.

The third set of Alternate settings is an example of Case 3 where the Alternate setting selects a Dfx host driver C. In this case, the bInterfaceSubClass field of the Interface descriptor is now equal to DvC.Dfx.

3.6.8.2 DvC.Trace Isochronous Trace Comments

Certain debug scenarios require the debug traces to share bandwidth with normal USB traffic (for example, when debugging a smartphone that is acting as a mass-storage device). Bulk transfers share the link bandwidth, and thus the debug trace will receive whatever bandwidth is left over. Isochronous transfers on the other hand, guarantee a minimum bandwidth, allowing the debugger to choose the appropriate bandwidth for a debug trace via an alternate setting.

There is no negotiation involved between the debugger and the TS when assigning bandwidth (unlike, for example, the Video class). The debuggers know the type of traces they are capturing (i.e., software messages, hardware messages, processor traces, etc.) and can thus choose an alternate setting that provides sufficient bandwidth.

The advantages of using isochronous transfers for debug traces are:

- Certain debug traces, such as processor-instruction traces, require a minimum, guaranteed bandwidth to be useful. Such traces typically contain internal synchronization points that allow them to recover from an occasional loss of trace, but if these gaps become too frequent then the trace becomes worthless. When debug is sharing the USB bus with another function (e.g., mass storage), and both are in Bulk mode, then a burst of activity by a non-debug function could ruin the debug trace. If the debug sighting requires a non-debug function to be active, and this function robs the debug trace of its necessary bandwidth, then this could prevent debug of the sighting.

- Generally, it is better to guarantee sufficient bandwidth for a quality debug trace, and hope that the remaining USB bandwidth is sufficient to provoke the bug scenario. The quality of a debug trace is paramount because processor-instruction traces can take many hours (even a day) to process. At the time of debug capture, one knows if the trace contains the bug sighting, and thus one can keep repeating the test until it does. The converse scenario of capturing a bad trace for the bug sighting will waste many hours/days before one discovers that it is necessary to repeat the test.

There are disadvantages to using isochronous. Debug traces can be very bursty with a high average bandwidth. This requires large buffers to smooth out the traffic. The minimum isochronous service-interval period is 125µs, which requires a 48KB debug trace buffer to sustain the bandwidth in the case of sporadic debug traces. For some implementations, dedicating a 48KB buffer for debug could be prohibitive. Thus some other solution is needed to reduce the size of the debug trace buffer. Appendix C: describes such a solution. However, the buffer cannot be too small because isochronous traffic accounts for 80% of the service interval. Figure 3-36 gives an example of a processor-instruction trace. This is typically bursty, as shown in the figure. If the device provides a trace buffer that is too small (e.g., smaller than 16KB), then portions of the trace will be lost every service interval. Such gaps in the trace can make it useless.

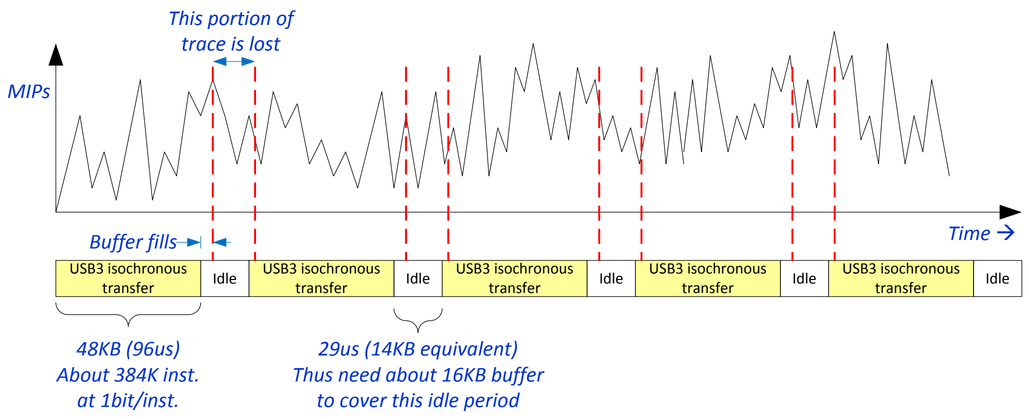

This portion of trace is lost
MIPs
Time →
Buffer fills
USB3 isochronous transfer
Idle
USB3 isochronous transfer
Idle
USB3 isochronous transfer
Idle
USB3 isochronous transfer
Idle
USB3 isochronous transfer
Idle
48KB (96us) About 384K inst. at 1bit/inst.
29us (14KB equivalent) Thus need about 16KB buffer to cover this idle period

Figure 3-36: Lost processor-instructions trace segments caused by inadequate trace buffers

4 Descriptors

4.1 Descriptor Layout Overview

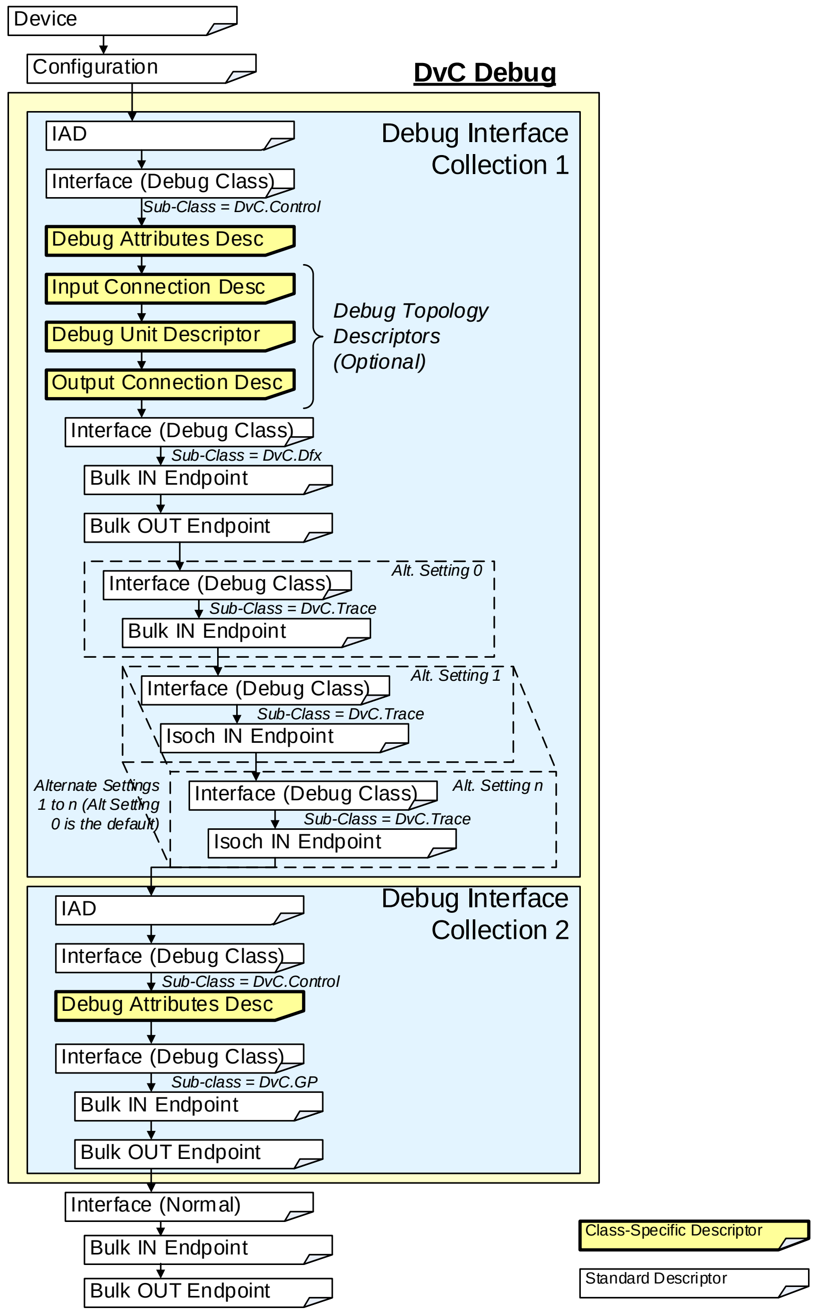


Figure 4-1: Debug-Descriptor Sample Layout

Figure 4-1 shows an example descriptor layout for the DvC Debug capability. DbC is identical except that it does not support non-debug "normal" USB interfaces.

Figure 4-1 shows all three debug interfaces (DvC.Dfx, DvC.Trace, and DvC.GP). It assumes isochronous traffic for the debug traces, and thus shows a number of alternate settings for the various bandwidth options. In addition, the example shows the device supporting a "normal", non-debug function. It also shows an IAD forming a DIC out of the Debug Control, DvC.Dfx, and the DvC.Trace interfaces. There is a second DIC for the DvC.GP capability. The DICs shown in this figure are purely examples, and other DIC configurations are possible.

The Debug Class allows for multiple configurations and multiple alternate settings (see Sections 3.6.1 and 3.6.2). One reason for this flexibility is because a TS may only have a very small number of endpoints available for debug, and would thus need to share them via alternate settings or multiple configurations.

The Debug Class supports a number of Debug Class-specific descriptors. It contains an IAD followed by a Debug-Control Interface descriptor, followed by a Debug-Attributes descriptor. See Figure 3-23. The Debug-Attributes descriptor describes which debug features the DIC supports. Next, there are optional topology descriptors followed by an optional interrupt endpoint (not shown in Figure 4-1). Finally, the DIC contains a Debug Capability interface descriptor or descriptors (e.g., DxC.Trace and/or DxC.Dfx and/or DxC.GP) together with their corresponding endpoints.

4.1.1 Class-Specific Topology Descriptors

Figure 4-2 is an example of a simple debug topology that is merging and selecting traces from a number of sources. Associated with each debug unit (e.g., Core, Merge unit, etc.) is an optional Debug Class-specific descriptor that provides the following information:

- A unique Unit ID that identifies each of the debug units in the topology. The Unit ID = 0 is reserved for accessing a “virtual” unit corresponding to the complete TS or a Debug-Interface collection. See the Control Section 5 for more details.
- Type of debug unit (e.g., Trace-Generation unit, Trace-Processing unit, etc.)
- Sub-type of debug unit (e.g., audio, graphics, core, modem for a Trace-Generation unit)
- Unit ID of an Alias debug unit. For example, a trace-generator unit can only generate a single output trace, but a functional unit may create multiple output streams (e.g., a core may generate software messages from the firmware and the OS, and a processor-instruction trace). The Alias field links this descriptor to the Unit ID of the same physical unit, thus informing the debugger that they are the same functional unit. If we need to alias more than two units, then we arbitrarily chose one as the reference.

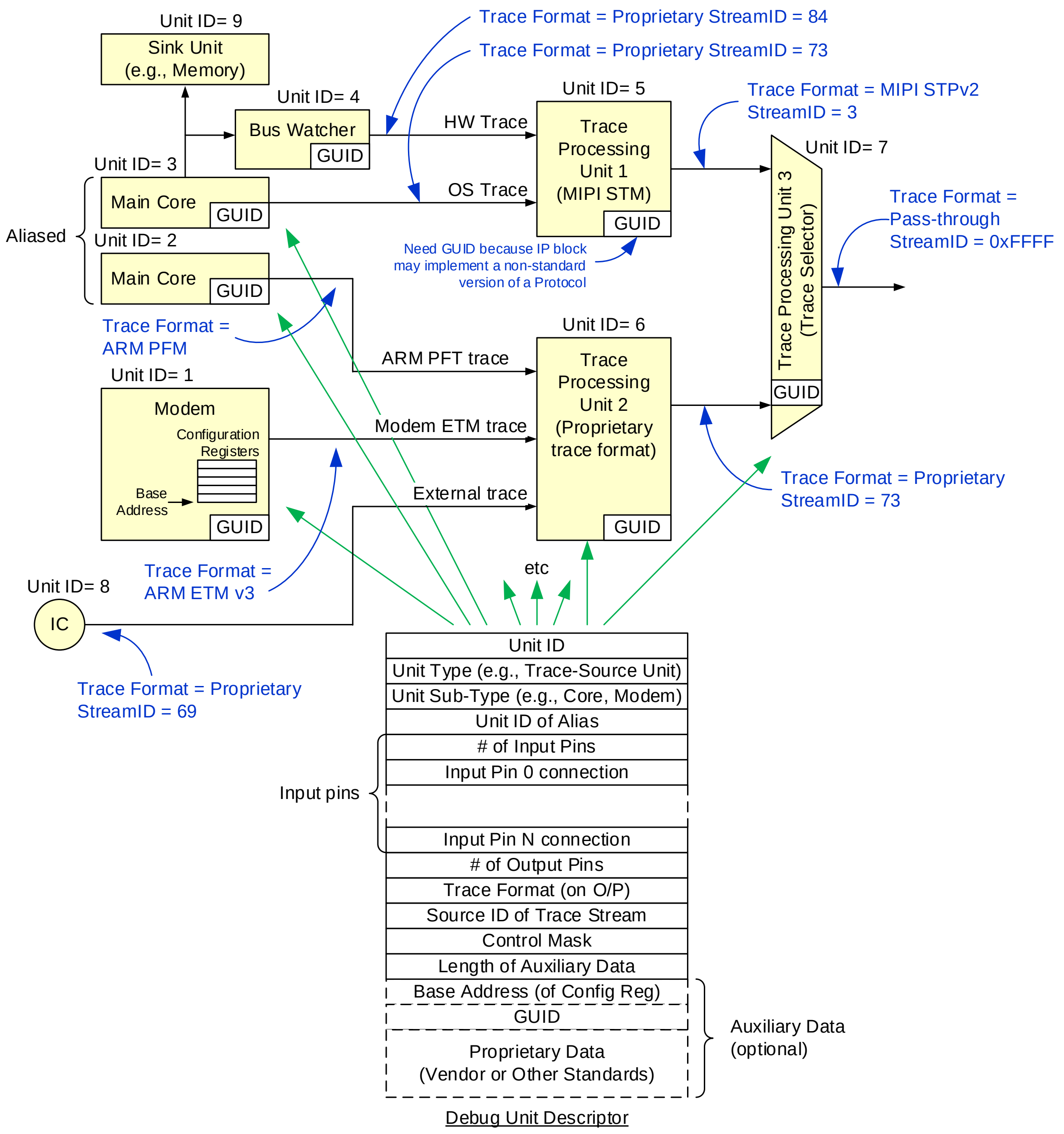

Unit ID= 9
Sink Unit (e.g., Memory)
Unit ID= 4
Bus Watcher
GUID
Trace Format = Proprietary StreamID = 84
Trace Format = Proprietary StreamID = 73
Unit ID= 5
HW Trace
OS Trace
Trace Processing Unit 1 (MIPI STM)
GUID
Trace Format = MIPI STPv2 StreamID = 3
Unit ID= 7
Trace Processing Unit 3 (Trace Selector)
GUID
Trace Format = Pass-through StreamID = 0xFFFF
Unit ID= 3
Main Core
GUID
Aliased
Unit ID= 2
Main Core
GUID
Need GUID because IP block may implement a non-standard version of a Protocol
Trace Format = ARM PFM
Unit ID= 6
ARM PFT trace
Trace Processing Unit 2 (Proprietary trace format)
GUID
Unit ID= 1
Modem
Configuration Registers
Base Address
GUID
Modem ETM trace
External trace
Trace Format = Proprietary StreamID = 73
Unit ID= 8
IC
Trace Format = ARM ETM v3
etc
Trace Format = Proprietary StreamID = 69
Unit ID
Unit Type (e.g., Trace-Source Unit)
Unit Sub-Type (e.g., Core, Modem)
Unit ID of Alias
# of Input Pins
Input Pin 0 connection
Input Pin N connection
Input pins
# of Output Pins
Trace Format (on O/P)
Source ID of Trace Stream
Control Mask
Length of Auxiliary Data
Base Address (of Config Reg)
GUID
Proprietary Data (Vendor or Other Standards)
Auxiliary Data (optional)
Debug Unit Descriptor

Figure 4-2: Debug-Unit Descriptor Example

- Number of input pins and their connectivity. The connectivity defines the Unit ID and the output pin ID of the source driving the input pin.
- Number of output pins
- Trace Format on the output pins of the unit. Each output pin can have a different Trace format. They are listed in order in the descriptor fields.
- Stream ID of the output trace. This is implementation dependent. For example, it could denote the identifier of the trace source (e.g., Master ID for a MIPI STPv1 trace). There is a StreamID per output trace, listed in order.
- Control Mask – this defines the Debug Class-specific commands that the debug unit supports
- Optional Auxiliary Data:
    - o Base Address of the Debug Configuration registers

- A Global-Unique Identifier (GUID) for the debug unit
- Supplementary debug data – a vendor &/or a Standards body could provide additional information in this field

Note that in the figure there are two instantiations of the same Main core. The Alias ID associates these debug units together. This is necessary because a Trace Generator unit can only generate a single output. However, in this example, the main core is actually generating two debug traces: a software instrumentation trace from the OS, and a processor-instruction trace. Consequently, two Trace Generator icons are required to define these traces.

The first Debug Class descriptors is the optional *Debug-Control* descriptor, followed by its associated *Debug-Attributes* descriptor, followed by the optional debug-topology (Input-*Connection, Output-Connection*, and *Debug-Unit*) descriptors. The debug-topology descriptors apply to all of the Debug interfaces (i.e., DxC.Dfx, DxC.Trace, and DxC.GP) within a DIC. The *wTotalLength* field in the *Debug-Attributes* descriptor defines the total size for the immediately following Debug Class-specific descriptors.

Note that the GUID in the topology descriptors is for the particular debug unit (e.g., Trace-Processing unit). This allows the debugger to recognize different variants of a specific IP block. For example, a particular IP block maybe an early adaptor of a protocol, and thus may not fully satisfy the protocol standard. The Debug Attributes descriptor provides the global GUID, which could, for example, be used as a unique link to a XML file describing additional information on the SoC.

The StreamID is an identifier that denotes information about the trace stream. For example, it could denote the identifier of the trace source. For example, it could be a trace from a particular core or it could be the instrumentation trace from the operating system or the application running on the core.

Typically the identifier denoting the trace source remains static throughout a debug session. However, it may change during a debug session when there are more trace sources than can be expressed by the identifier field of the trace protocol. Consequently, the TS may reassign the trace identifier during a debug session. The Stream ID field of the descriptor provides the initial assignment of the trace identifier. For some implementations this could be a static assignment that never varies; while for others it could be a dynamic value that can vary depending on when the descriptors are accessed (e.g., during USB enumeration). In other words, for some implementations the descriptors could be constantly updated by a debug application running on the TS, and the Host can access the updated descriptors during a debug session via a GET_DESCRIPTOR command.

It is implementation dependent how the TS informs the debugger of a reassignment of the stream ID should the implementation support dynamically varying StreamIDs. One option is for the debugger to periodically poll with a USB standard GET_DESCRIPTOR request. Alternatively, the debugger could periodically issue Debug Class-specific GET_CONFIG request to an implementation-specific configuration register throughout the debug session.

Figure 4-3 is an example implementation showing the MIPI STM unit of Figure 4-2. The two input traces to the MIPI STM have StreamIDs (i.e., Master IDs) of 84 and 73, while the output of the MIPI STM has a StreamID of 3. The StreamID for the output of the MIPI STM unit in Figure 4-3 corresponds to the Master ID of the output MIPI STP trace. This output trace is the merged result of the two input streams, and thus the StreamID identifier's of the input traces is actually embedded within the output trace – see Figure 4-3.

The Stream ID number space is unique to each trace stream. Thus in this particular example, the StreamID values are all different for the three different traces. However, it is possible that all three traces have the same value by happenstance. For example, all three traces shown in Figure 4-3 could have the same value for StreamID = 5, 5, and 5, instead of 84, 73, and 3. The fact that the StreamID is the same is pure coincidence and does not imply any correlation between the three traces.

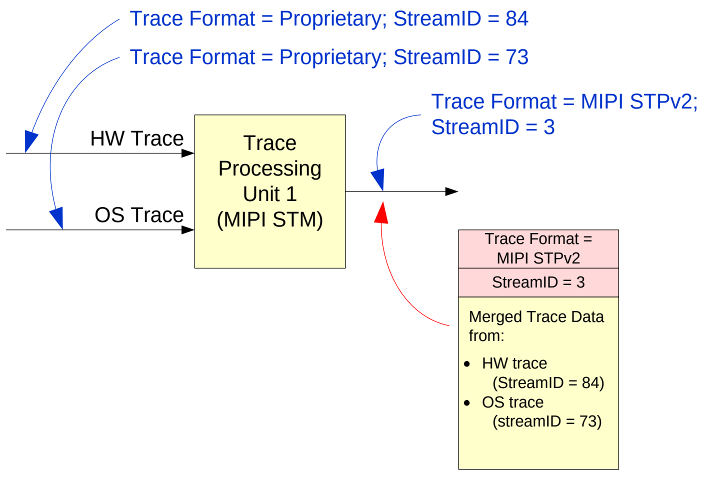

Trace Format = Proprietary; StreamID = 84
Trace Format = Proprietary; StreamID = 73
HW Trace
OS Trace
Trace Processing Unit 1 (MIPI STM)
Trace Format = MIPI STPv2; StreamID = 3
Trace Format = MIPI STPv2
StreamID = 3
Merged Trace Data from:
HW trace (StreamID = 84)
OS trace (streamID = 73)

Figure 4-3: StreamID Example

4.2 xHCI-Compliant DbC Standard Descriptors

The USB3.1 Debug Class supports the legacy, xHCI-compliant DbC. Please refer to the xHC specification for details of these Descriptors [3].

4.3 Debug Standard Descriptors

4.3.1 USB 2.0 Descriptors

If the device provides USB 2.0 debug support then it shall support the following standard USB 2.0 descriptors for DxC:

- **Device:** Each USB device has one device descriptor (per *USB Specification*).
- **Configuration**: Each USB device has at least one default configuration descriptor, which supports at least one interface. (That is, multiple configurations are allowed, but not recommended – see Section 3.6.2).
- **Interface**: The device shall support at least one debug interface. Some devices *may* support additional (normal, non-debug) interfaces to provide other capabilities (e.g., Mass storage).
- **Endpoint:** The device shall support at least one of the debug endpoint sets, in addition to the default pipe that is required of all USB devices (see Table 4-1):

Table 4-1: DxC Debug Endpoints

Debug Capability	Endpoint	Data Type
DxC.Trace	IN	Bulk, Isochronous
DxC.Dfx	IN, OUT	Bulk
DxC.GP	IN, OUT	Bulk
Debug Control	IN, OUT	Control Interrupt (optional)

Some devices *may* support additional endpoints to provide other non-debug capabilities. The host shall use the first reported endpoints for the selected interface.

- **String:** The device shall supply a unique serial number.

The rest of this section describes the standard USB device, configuration, interface, endpoint, and string descriptors for the device. For superseding information about these and other standard descriptors, see Chapter 9, "USB Device Framework," of the *USB Specification* [4].

4.3.1.1 USB 2.0 Device Descriptor

Because debug functionality always resides at the Interface level, this class specification does not define a specific debug Device descriptor.

If a Debug Class device uses an Interface Association Descriptor in order to describe a Debug Interface Collection, then it shall set the **bDeviceClass**, **bDeviceSubClass** and **bDeviceProtocol** fields to 0xEF, 0x02, and 0x01 respectively. This set of class codes defines the Multi-interface Function Class codes.

If there is no IAD, then the device descriptor shall indicate that class information is to be found at the interface level. Therefore, the **bDeviceClass** field of the device descriptor shall contain zero so that enumeration software looks down at the interface level to determine the Interface Class. The bDeviceSubClass and bDeviceProtocol fields shall be set to zero.

All other fields of the device descriptor shall comply with the definitions in section 9.6.1 "Device" of USB Specification [4]. There is no class-specific Device descriptor.

4.3.1.2 USB 2.0 Device-Qualifier Descriptor

The Device-Qualifier descriptor is required for all USB 2.0 high-speed capable devices. The rules that apply for setting the **bDeviceClass**, **bDeviceSubClass** and **bDeviceProtocol** fields in the Device Descriptor apply for this descriptor as well. All other fields of the device qualifier descriptor shall comply with the definitions in section 9.6.2 "Device Qualifier" of USB Specification [4].

4.3.1.3 USB 2.0 Configuration Descriptor

The Configuration descriptor for a device containing a debug function is identical to the standard Configuration descriptor defined in section 9.6.3 "Configuration" of *USB Specification* [4]. There is no class-specific configuration descriptor.

4.3.1.4 Other_Speed_ Configuration Descriptor

The Other_Speed_Configuration descriptor is required for USB 2.0 devices that are capable of operating at both full-speed and high-speed modes. It is identical to the standard Other_Speed_Configuration descriptor defined in section 9.6.4 "Other_Speed_Configuration" of *USB Specification* [4].

The Debug Class recommends High-speed only.

4.3.1.5 Interface Association Descriptor

A device shall use an Interface-Association Descriptor to describe a Debug-Interface Collection. See Section 3.6.4 for more details and examples.

When using an IAD, the iFunction field in the IAD and the interface field in the Standard Debug Class Interface descriptor for this Debug-Interface Collection shall be equal.

Table 4-2 defines the Interface-Association Descriptor.

Table 4-2: Interface Association Descriptor

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	Number
bDescriptorType	1	1	INTERFACE ASSOCIATION Descriptor	Constant
bFirstInterface	2	1	Interface number of the first Debug-Control Interface that is associated with this function	Number
bInterfaceCount	3	1	Number of Debug interfaces that are associated with this function. The interface numbers in the set of associated interfaces are contiguous (there can be no gaps in the list of interface numbers). The count includes the first Debug-Control interface and all its associated Debug interfaces (i.e., DxC.Dfx, DxC.Trace).	Number
bFunctionClass	4	1	Class code	0xDC CC_DEBUG. See Appendix A:
bFunctionSubClass	5	1	Sub-class code	SC_DEBUG. See Appendix A:
bFunctionProtocol	6	1	Protocol code	PC_DEBUG See Appendix A:
iFunction	7	1	Index of string descriptor describing this function. The value is zero if there is no string descriptor.	xxh

The USB 3.1 specification strongly recommends that device implementations utilizing the IAD use the Multi-Interface Function class codes in the device descriptor. This allows simple and easy identification of these devices and allows on some operating systems, installation of an upgrade driver that can parse and enumerate configurations that include the IAD. The Multi-Interface Function class is documented at http://www.usb.org/developers/docs.

The class and subclass fields of the IAD are not required to match the class and subclass fields of the interfaces in the interface collection that the IAD describes. However, Microsoft recommends that the first interface of the collection have class and subclass fields that match the class and subclass fields of the IAD. Table 4-3 indicates which fields should match.

Table 4-3: IAD and Interface Descriptor Matching

IAD field	Corresponding field of the 1st Interface	Value
bFunctionClass	bInterfaceClass	CC_DEBUG
bFunctionSubclassClass	bInterfaceSubClass	SC_DEBUG

Typically, a DIC will have start with a Debug-Control Interface descriptor, and thus the Class field will be DCh for both the IAD and the Debug-Control Interface, and the SubClass = SC_DEBUG_CONTROL = 0x08.

The bFirstInterface field of the IAD indicates the number of the first interface in the function. The bInterfaceCount field of the IAD indicates how many interfaces are in the interface collection. Interfaces in an IAD interface collection shall be contiguous (there can be no gaps in the list of interface numbers), and so a count with a first interface number is sufficient to specify all of the interfaces in the collection.

4.3.1.6 USB 2.0 Interface Descriptor

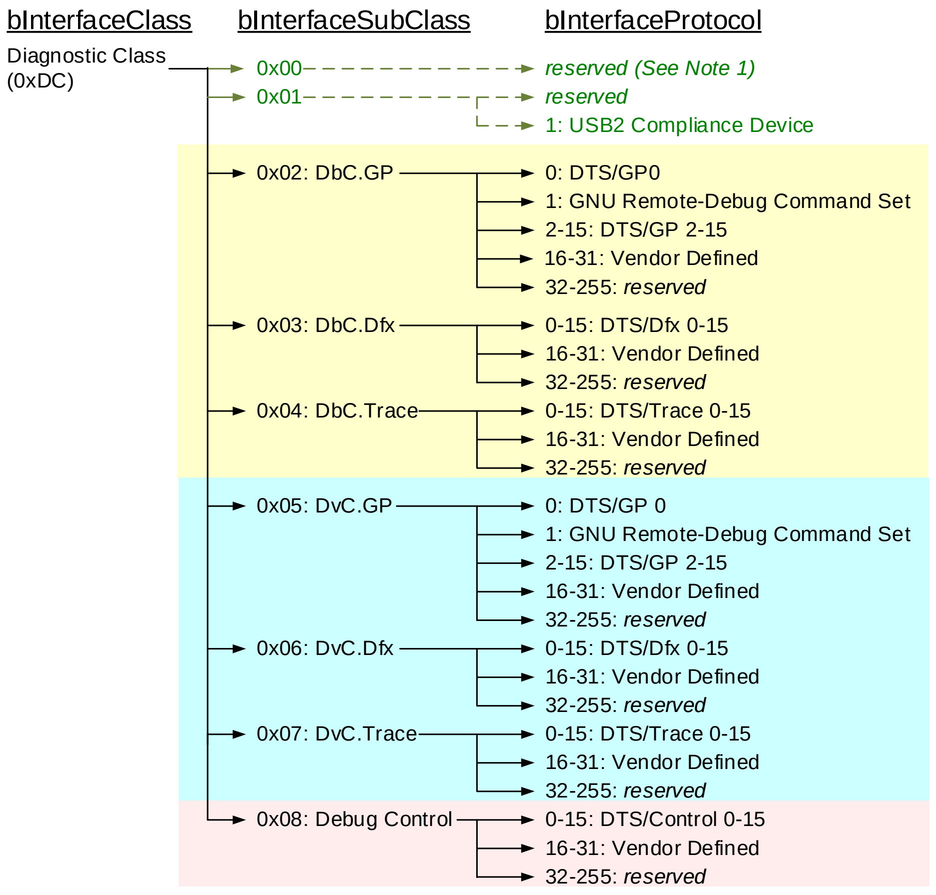
This section defines the Interface Descriptor for the Debug class.

Table 4-4: USB 2.0 Standard Interface Descriptor for the Debug Class

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	09h
bDescriptorType	1	1	Interface Descriptor Type (assigned by USB)	04h
bInterfaceNumber	2	1	Number of the interface. A zero-based value identifying the index in the array of concurrent interfaces supported by this configuration	xxh
bAlternateSetting	3	1	Value used to select alternate setting for the interface identified in the prior field.	xxh
bNumEndpoints	4	1	Number of endpoints used by this interface (excluding endpoint zero). This number is 0 or 1 depending on whether the optional status interrupt endpoint is present	xxh
bInterfaceClass	5	1	Class code	0xDC (Diagnostic Class)

Part	Offset (Byte)	Size (Bytes)	Description	Value
bInterfaceSubClass	6	1	Sub-class code: Debug Capability DbC.GP, DbC.Dfx, DbC.Trace DvC, DvC.Dfx, DvC.Trace Debug Control	SC_DEBUG. See Appendix A:
bInterfaceProtocol	7	1	Protocol code:	PC_DEBUG See Appendix A:
iInterface	8	1	Index of string descriptor describing this interface.	xxh

The Interface descriptor of a Debug Class device includes a Sub-class field and a Protocol field, as shown in Figure 4-4.



Note 1: The xHCI-Compliant DbC specifies bInterfaceSubClass = 0x00. Thus, bInterfaceClass = 0XDC, bInterfaceSubClass = 0x00 can point to a legacy xHCI-Compliant DbC device.

Figure 4-4: Diagnostic Class, Sub-Class, and Protocol partitioning

The subclass field defines the debug capability. The Protocol field defines is used to define different instantiations of debug interface (e.g., DvC.GP0, DvC.GP1, etc.), or to support different debug tools via the Set Alternate interfaces (see Section 3.6.5) in the first 16 entries, and the following 16 are vendor defined. The actual debug tools in the first 16 entries are also vendor defined. The 16 DTS slots allow up to sixteen different drivers to be resident on the host machine corresponding to sixteen different debuggers.

It is not-unusual to have multiple debuggers in the lab, with varying capabilities. For example, one TAP debugger could be a commercial offering while another is a vendor-proprietary tool that provides access to proprietary data structures. Thus, for DxC.Dfx, the TS vendor may choose to use bInterfaceProtocol = 1 for one of these debuggers and bInterfaceProtocol = 2 for the other. This will result in the host instantiating two different drivers, which will simplify the coexistence of two debuggers on a single host. See Section 3.6.5.

The assignment of Protocol fields for DbC.GP matches that of the original xHCI DbC. For compatibility, DvC.GP uses the same assignment.

4.3.1.7 USB 2.0 Endpoint Descriptors

The endpoint descriptor is identical to the standard endpoint descriptor defined in section 9.6.6 “Endpoint” of *USB Specification* [4].

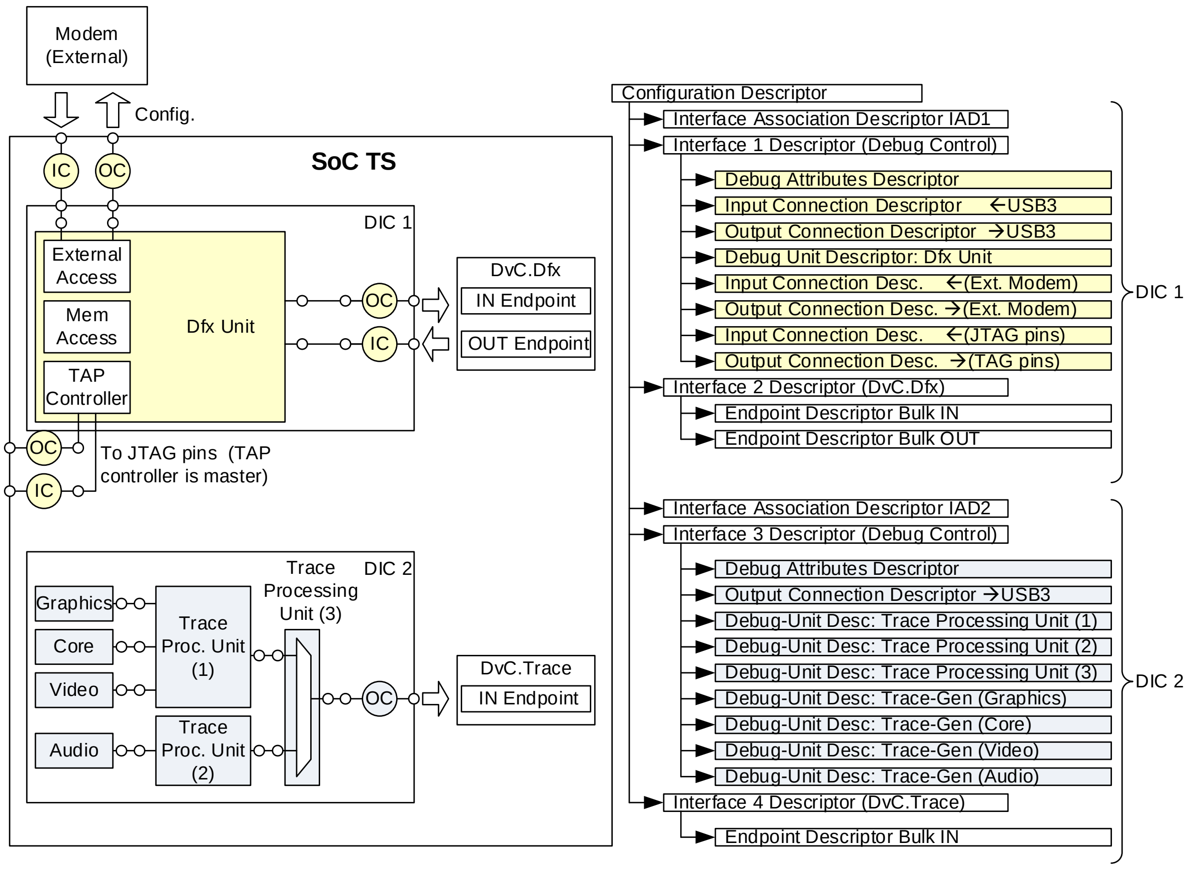
4.3.2 USB 3.1 Standard Descriptors

The USB 3.1 descriptors are defined in the USB 3.1.0 specification and are not duplicated here.

4.4 Debug Class-Specific Descriptors

4.4.1 Introduction

There are a number of Debug Class-specific descriptors associated with the various debug capabilities (DxC.Dfx, DxC.Trace, and DxC.GP). Figure 4-5 shows an example implementation of an SoC device, which also provides debug connectivity to an external modem, allowing the debug logic within the SoC to configure and capture debug traces from the Modem. The right-hand-side of the figure shows the standard and class-specific descriptors associated with these debug hooks. The yellow and blue shaded descriptors are the class-specific descriptors. These class-specific descriptors define the capabilities of the debug hooks (e.g., whether the core can create processor traces or not), and how these hooks are interconnected.



Modem (External)
Config.
SoC TS
IC
OC
DIC 1
External Access
Mem Access
TAP Controller
Dfx Unit
DvC.Dfx
IN Endpoint
OUT Endpoint
To JTAG pins (TAP controller is master)
Trace Processing Unit (3)
DIC 2
Graphics
Core
Video
Audio
Trace Proc. Unit (1)
Trace Proc. Unit (2)
DvC.Trace
IN Endpoint
Configuration Descriptor
Interface Association Descriptor IAD1
Interface 1 Descriptor (Debug Control)
Debug Attributes Descriptor
Input Connection Descriptor ←USB3
Output Connection Descriptor →USB3
Debug Unit Descriptor: Dfx Unit
Input Connection Desc. ←(Ext. Modem)
Output Connection Desc. →(Ext. Modem)
Input Connection Desc. ←(JTAG pins)
Output Connection Desc. →(TAG pins)
Interface 2 Descriptor (DvC.Dfx)
Endpoint Descriptor Bulk IN
Endpoint Descriptor Bulk OUT
DIC 1
Interface Association Descriptor IAD2
Interface 3 Descriptor (Debug Control)
Debug Attributes Descriptor
Output Connection Descriptor →USB3
Debug-Unit Desc: Trace Processing Unit (1)
Debug-Unit Desc: Trace Processing Unit (2)
Debug-Unit Desc: Trace Processing Unit (3)
Debug-Unit Desc: Trace-Gen (Graphics)
Debug-Unit Desc: Trace-Gen (Core)
Debug-Unit Desc: Trace-Gen (Video)
Debug-Unit Desc: Trace-Gen (Audio)
Interface 4 Descriptor (DvC.Trace)
Endpoint Descriptor Bulk IN
DIC 2

Figure 4-5: Debug Topology and Descriptor Hierarchy Example 1

The example in Figure 4-5 uses the DvC.Dfx and DvC.Trace capabilities. The class-specific descriptors are shaded the same color as their corresponding debug logic.

Note that the example in Figure 4-5 shows hardware debug units. However, one can use the topology to describe the interconnection of software applications (e.g., GNU debugger, data loggers, etc.). These would most naturally use the DxC.GP interface.

The example in Figure 4-5 shows the debug interfaces grouped into a pair of Debug-Interface Collections using Interface Association Descriptors. IAD1 group Interface 1 and 2, while IAD2 groups Interfaces 3 and 4. These two DICs will connect to two different drivers in the host. Alternatively, Figure 4-6 uses a single IAD to group together interfaces 1, 2, and 3. In this case, a single driver in the host will control all of the debug interfaces. We recommend the grouping of Figure 4-5 rather than Figure 4-6 when the two debug functions are independent. However, Figure 4-6 is appropriate if the implementation uses an IP block for complete Dfx/Trace unit and the Dfx needs to deal with this IP block as an entity (e.g., when using a vendor’s IP for a composite debug block with an associated debugger).

Note that a device may ship with the descriptors corresponding to Figure 4-5 and the user can change them to those of Figure 4-6 via Android adb or some other similar mechanism.

The DvC.Dfx interface in either Figure 4-5 or Figure 4-6 communicates with a Dfx unit, which in this example contains a TAP controller, a memory-access unit, and an external-access unit that interfaces to a modem outside the SoC. This outbound path allows the Dfx unit to configure the modem to generate debug traces. The modem traces return on the inbound path.

The DvC.Trace portion consists of four agents generating traces (graphics, core, video, and audio units). The Trace-Processing unit 1 combines the traces from the graphics, core, and video units. The Trace-Processing unit 2 packetizes the single audio traces into a standard trace format. The Trace-Select unit chooses between the two possible traces streams.

The Output Connection (OC) and an Input Connection (IC) define inputs and outputs to the debug logic. Three of these Connections connect to the USB 3.1 endpoints, two connect to an external modem, and the final two connect to the JTAG pins on the device. The later capability allows the TAP controller to act as a JTAG master and control external chips via a JTAG chain.

The class-specific, Debug-Unit descriptors defines the capabilities of the debug unit (e.g., whether the unit is a trace-generator unit, or a Trace-Processing unit, etc.) It also defines the type of the debug unit (e.g., Audio unit, graphics unit, modem, etc.).

The example in Figure 4-5 shows a generic Dfx Unit, which consists of three sub components (i.e., TAP controller, memory-access unit, and external-access unit). There are no specific descriptors for these sub components. Thus, the debugger will treat the Dfx unit as single entity.

Note that an implementation may choose to define a Debug-Unit descriptor of type Dfx for each of these sub-components. This is implementation specific. For some designs, it may be preferable to treat these 3 sub-components as independent Dfx units, while for others it may be preferable to treat these as a single, combined unit.

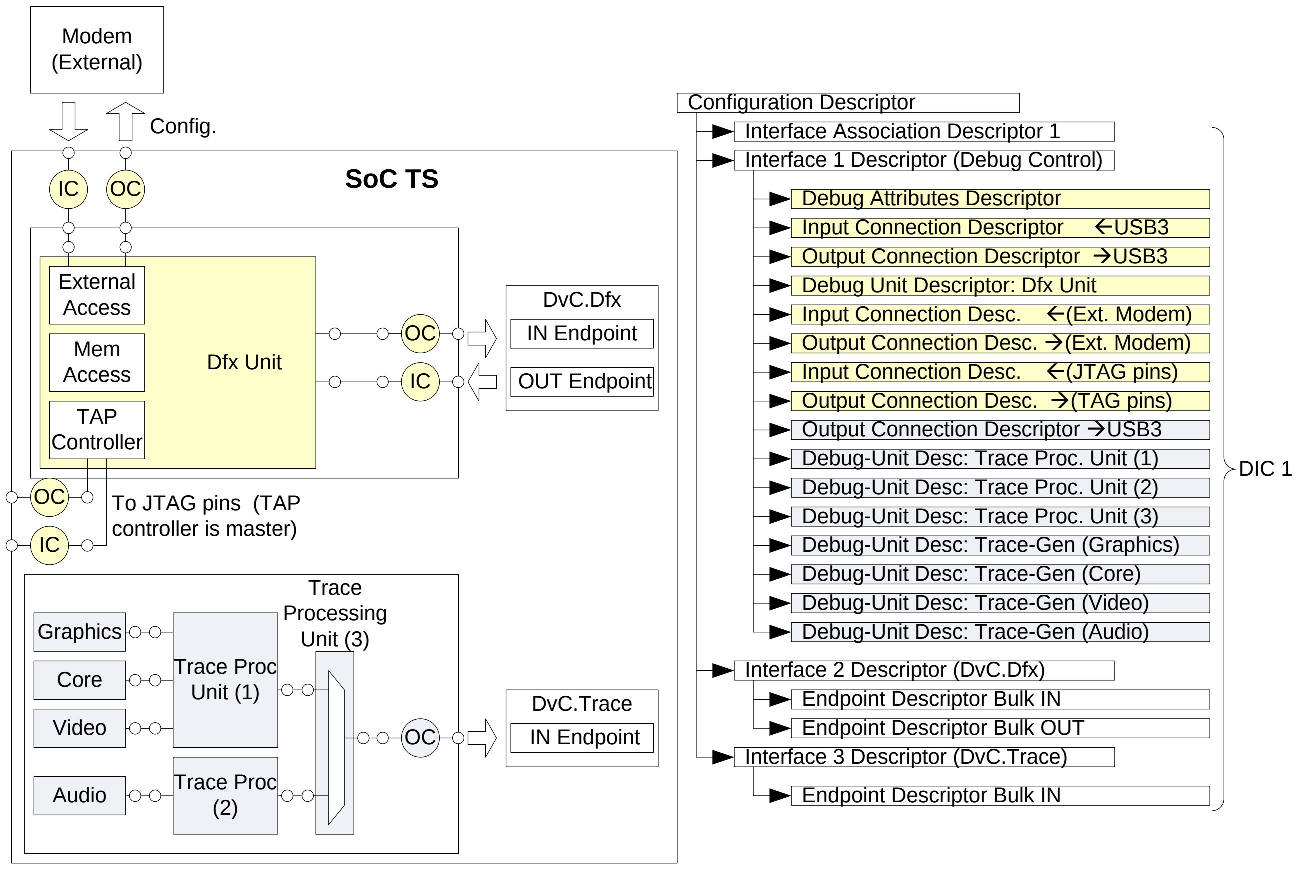

Modem (External)
Config.
SoC TS
IC
OC
External Access
Mem Access
TAP Controller
Dfx Unit
DvC.Dfx
IN Endpoint
OUT Endpoint
To JTAG pins (TAP controller is master)
Trace Processing Unit (3)
Graphics
Core
Video
Audio
Trace Proc Unit (1)
Trace Proc (2)
DvC.Trace
IN Endpoint
Configuration Descriptor
Interface Association Descriptor 1
Interface 1 Descriptor (Debug Control)
Debug Attributes Descriptor
Input Connection Descriptor ←USB3
Output Connection Descriptor →USB3
Debug Unit Descriptor: Dfx Unit
Input Connection Desc. ←(Ext. Modem)
Output Connection Desc. →(Ext. Modem)
Input Connection Desc. ←(JTAG pins)
Output Connection Desc. →(TAG pins)
Output Connection Descriptor →USB3
Debug-Unit Desc: Trace Proc. Unit (1)
Debug-Unit Desc: Trace Proc. Unit (2)
Debug-Unit Desc: Trace Proc. Unit (3)
Debug-Unit Desc: Trace-Gen (Graphics)
Debug-Unit Desc: Trace-Gen (Core)
Debug-Unit Desc: Trace-Gen (Video)
Debug-Unit Desc: Trace-Gen (Audio)
Interface 2 Descriptor (DvC.Dfx)
Endpoint Descriptor Bulk IN
Endpoint Descriptor Bulk OUT
Interface 3 Descriptor (DvC.Trace)
Endpoint Descriptor Bulk IN
DIC 1

Figure 4-6: Debug Topology and Descriptor Hierarchy Example 2

4.4.2 Debug-Control Interface Descriptors

The optional Debug-Control Interface descriptors contain all relevant information to fully characterize the corresponding Debug function. There are two descriptors associated with debug control:

1. Debug-Control Interface descriptor: This is a standard USB interface descriptor that characterizes the interface itself. This descriptor is optional.
2. Debug-Attributes descriptor: This is a class-specific interface descriptor that provides additional information concerning the internals of the debug function. It specifies the revision level and lists

the general debug capabilities of the complete TS. This descriptor is mandatory if the associated Debug-Control Interface descriptor exists.

The topology of the debug function is defined by zero or more of the following optional descriptors in any order:

- Input Connection descriptor
- Output Connection descriptor
- Debug Unit Descriptor

The Debug Control requests can manipulate/control any of the units within the above topology, or the DIC that encompasses this topology, or even the complete TS that encompasses one or more DICs.

The Debug-Control interface has no dedicated endpoints associated with it. It uses the default pipe (endpoint 0) for all communication purposes, except for optional event notification, in which case the interrupt endpoint is used. Class-specific debug control requests are sent using the default pipe.

The Debug-Control Interface may use multiple alternate setting. For example, when sharing an endpoint between GP and Dfx, as per the example in Section 3.6.2, then each capability may require different Debug Commands, and would thus need different Debug Control and Debug Attributes descriptors.

The standard Debug-Control Interface descriptor is identical to the standard interface descriptor defined in section 9.6.5 "Interface" of USB Specification Revision 2.0, except that some fields have dedicated values (see Table 4-5).

Table 4-5: Standard Debug-Control Interface Descriptor

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	09h
bDescriptorType	1	1	INTERFACE	04h
bInterfaceNumber	2	1	Index of this interface	xxh
bAlternateSetting	3	1	Value used to select alternate setting for the interface identified in the prior field.	xxh
bNumEndpoints	4	1	1 optional endpoint (interrupt endpoint)	xxh
bInterfaceClass	5	1	CC_DEBUG	DCh
bInterfaceSubClass	6	1	SC_DEBUG_CONTROL	80h
bInterfaceProtocol	7	1	Not used. Set to PC_PROTOCOL_UNDEFINED	00h
iInterface	8	1	This is a TAG that has to match the iFunction field in the Debug Interface Collection IAD.	xxh

4.4.3 Debug-Attributes Descriptor

The Debug Control & Debug-Attributes interface descriptors contain all relevant information to fully characterize the corresponding debug function. The standard, Debug-Control interface descriptor characterizes the interface itself, whereas the class-specific Debug-Attributes interface descriptor provides pertinent information concerning the internals of the debug function. It specifies revision level information and lists the capabilities of each Unit and Terminal.

This Debug-Attributes descriptor is located immediately after the Debug-Control Interface descriptor and is mandatory if the Debug-Control Descriptor exits. Thus, the Debug-Attributes descriptor is always paired with the Debug-Control descriptor.

Table 4-6 defines the Debug-Attributes descriptor.

Table 4-6: Debug Class Debug-Attributes Descriptor

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	Number
bDescriptorType	1	1	CS_INTERFACE	24h
bDescriptorSubType	2	1	DC_DEBUG_ATTRIBUTES	04h
bcdDC	3	2	Revision number of Debug Class specification that this TS/DIC is based on.	0100h (rev 1)
wTotalLength	5	2	Total size of the topology and interrupt class-specific descriptors for this debug function. It does not include the debug Capability descriptors (e.g., DxC.Dfx)	Number
bTSorDIC	7	1	Defines whether this descriptor pertains to the complete TS or to a DIC 0: DIC 1:TS Otherwise: *reserved*	Number
bmSupportedEvents	8	1	Defines if debug interrupt events are supported (i.e., triggers, hot button): D0: Debug Event supported on TS if true D1: Debugger starts trace capture if debug “button” asserts and D0 is 1. *D2: D3: reserved* D4-D7: Vendor specific	Bitmap
bControlSize	9	1	Size of the bmControls field, in bytes: n	Number

Part	Offset (Byte)	Size (Bytes)	Description	Value
bmControl (See Section 5.1 for more information)	10	n	A bit set to 1 indicates that the following Debug-Control requests are supported by the DIC/TS depending on the setting of bTSorDIC. Note that a Debug unit within the TS or DIC may support completely different debug commands – these are defined in the corresponding field of the Debug-Unit descriptor. D0: SET_CONFIG_DATA_SINGLE D1: SET_CONFIG_DATA D2: GET_CCONFIG_DATA D3: SET_CONFIG_ADDRESS D4: GET_CONFIG_ADDRESS D5: SET_ALT_STACK D6: GET_ALT_STACK D7: SET_OPERATING_MODE D8: GET_OPERATING_MODE D9: SET_TRACE_CONFIGURATION D10: GET_TRACE_CONFIGURATION D11: SET_BUFFER D12: GET_BUFFER D13: SET_RESET D(n*8-1)..14: *reserved* If SET_CCONFIG_ADDRESS is not supported but SET/GET_CCONFIG_DATA is supported, then Configuration address defaults to value 0.	Bitmap
bAuxDataSize	10+n	1	This field defines the size of the next two fields. If there is no auxiliary data then = 0, otherwise 24.	Number
qBaseAddress	11+n	8	Base Address to the Configuration registers of the DIC or the TS depending on the value of bTSorDIC. A Base Address = 0 is used to indicate that there is no Base Address.	Constant
hGlobalID	19 + n	16	Identifier for the complete TS or DIC depending on the value of bTSorDIC. For example, this could be the GUID for the TS.	Constant

Part	Offset (Byte)	Size (Bytes)	Description	Value
wVendorDataSize	10+n or 35+n	2	This field defines the size of the remaining bytes in the descriptor in bytes: q	Number
Vendor Data	12+n or 37+n	q	Vendor defined data (q bytes)	Number

The bcdDC is the revision of Debug Class specification that this TS/DIC is based upon. Note that a DIC may contain a DxC.Trace interface and a DxC.Dfx interface that each support a different revision of the specification. In this case, the DIC requires Debug-Unit descriptors for the Trace and Dfx to state which revision they support. For example:

- DIC containing Trace and Dfx interfaces, and the DIC itself only supports Rev 1.0 Debug Commands (e.g., Rev 1.0 commands to power-on the debug logic).
    - ▪ Thus, Debug Attribute (bTSorDIC = 1, bcdDC = 1.0)
  - o Debug-Unit Descriptor for Trace unit supports rev 2.0 & thus has bcdDC = 2.0
  - o Debug-Unit Descriptor for Dfx unit supports rev1.0 & thus has bcdDC = 1.0

The wTotalLength field reflects the total length in bytes of all the descriptors that are used to fully describe the debug function, which is the topology and any interrupt descriptor. Thus, all Debug-Unit descriptors, all Input-Connection and Output-Connection descriptors, together with the Interrupt descriptor.

The bTSorDIC is used to define whether this Debug-Attributes descriptor pertains to the complete TS or the DIC. In particular, this is used to define Debug Commands that are specific to the TS or to a DIC. For example, it may only be possible to Power-on/off all the debug logic within a TS but not power-on/off individual DICs. Thus, to allow this capability, we need the following:

TS: Debug Control & Attributes (bTSorDIC) = TS and bmControl allows Operating Modes

DIC: Debug Control & Attributes (bTSorDIC) = DIC & bmControl does not allow Operating Modes

Figure 4-7 shows an example where there is a pair of Debug-Control and Debug Attributes descriptors defining the capability of the complete TS, and a pair of Debug-Control and Debug Attributes descriptors defining the capability of a DIC.

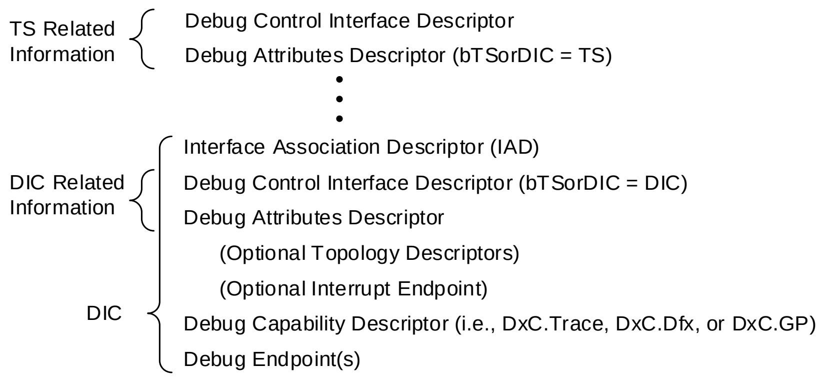

TS Related Information
Debug Control Interface Descriptor
Debug Attributes Descriptor (bTSorDIC = TS)
DIC Related Information
Interface Association Descriptor (IAD)
Debug Control Interface Descriptor (bTSorDIC = DIC)
Debug Attributes Descriptor
(Optional Topology Descriptors)
(Optional Interrupt Endpoint)
DIC
Debug Capability Descriptor (i.e., DxC.Trace, DxC.Dfx, or DxC.GP)
Debug Endpoint(s)

Figure 4-7: Example of TS and DIC Debug Control & Attributes descriptor usage

The bmSupportedEvents field indicates if the TS or DIC supports Interrupts for breakpoints, low-power-transitions, etc. If supported, then an Interrupt interface is mandatory. The Debug Class only supports IN endpoints for the Interrupt interface, and thus the TS can only send out interrupts to the DTS, but not vice-versa. If the DTS needs to communicate with the TS, it can use the Debug-Control Interface.

The bmControl field is a bitmask indicating which commands the DIC (if bTSorDIC = 0) or TS (if bTSorDIC=1) supports. The optional Debug-Unit descriptors define the Debug Commands supported by the optional individual debug units.

A non-zero wAuxDataSize indicates that the descriptor contains an auxiliary debug data structure for the remainder of the descriptor starting immediately after the wAuxDataSize field. This data structure consists of a number of fields:

- Input and Output buffer sizes for the DIC
- Address to this data structure in the qBaseAddress field
- The GUID for this TS in the hGlobalID field. The hGlobalID provides an ID (e.g., GUID) for the complete debug entity (and not for this particular DIC). Thus if there are multiple DICs, then each Debug Attributes descriptor should provide the same information (or the subsequent descriptors should provide zero Auxiliary Data). Otherwise, if each Auxiliary data structure is different in the various DICs, then the interpretation is vendor-specific.

  The hGlobalID could, for example be used as a unique link to a XML file providing additional debug information on the debug entity.
- Optional supplementary data. This data could be proprietary to a vendor or defined via a standards body.

Following the Debug-Attribute descriptor are zero or more class-specific descriptors. There is at least one Debug Class-specific descriptor if the bTotalLength value exceeds the bLength field. These class-specific descriptors are the optional debug-topology descriptors. The layout of the topology descriptors depends on the type of Unit or Connection they represent.

Note: A Standards body will want to create a set of Debug commands for their particular Dfx/Trace unit. It is unlikely (at least for the foreseeable future) that a Standards group will want to create debug commands for the complete DIC or TS. Consequently, the Debug Class specification only provides support for Vendor data and not for Standards bodies. Instead, the Debug-Unit descriptor provides this support (see later). This shortcoming could be addressed in a future revision of the Debug Class specification should this be necessary.

The Debug-Control Interface may use multiple alternate settings, and thus each of these alternate settings will have an associated Debug-Attributes descriptor. There is no Alternate Settings field in the Debug-

Attributes descriptor because the Debug-Control and the Debug-Attributes descriptors are always paired with each other:

Debug-Control (Alt.Setting = 0)	Debug-Control (Alt.Setting = 1)
Debug-Attributes descriptor A	Debug-Attributes descriptor B

4.4.4 Input-Connection Descriptor

The Input-Connection descriptor describes functional aspects of the Input-Connection to the device.

The value in the bConnectionID field uniquely identifies an Input-connection. No other Unit or Connection within the same DIC may have the same ID. For example, each Connection and each Unit within a DIC shall have a unique ID, but different DICs can reuse the same ID.

The bConnectionType field defines where the Input Connection connects. This could be a USB OUT endpoint, an external debug-in connection, etc.

The bAssocConnection field associates an Output Connection to this Input Connection, effectively implementing a bi-directional Connection pair. For instance, this would link the JTAG input and output pins. If the bAssocConnection field is used, both associated Connections shall belong to the bi-directional Connection Type group. If no association exists, the bAssocConnection field shall be set to zero.

The Host software can treat the associated Connections as being physically or logically related. In many cases, one Connection cannot exist without the other. An index to a string descriptor is provided to further describe the Input-Connection.

The IC may be carrying traces. The trace format is defined by the (optional) dTraceFormat field. This field is at the end of the descriptor, and thus if the IC does not carry traces then this field is not required. In this case, the bLength = 07h, otherwise it is 0Bh.

The optional dStreamID provides information on the ID of the trace. For example, this could be the Master ID for a MIPI STP trace.

Table 4-7 describes the Input-Connection descriptor:

Table 4-7: Input Connection Descriptor

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	07h or 0Bh
bDescriptorType	1	1	CS_INTERFACE	24h
bDescriptorSubType	2	1	DC_INPUT_CONNECTION	01h
bConnectionID	3	1	A non-zero constant that uniquely identifies the Connection within the debug capability (DxC.Dfx or DxC.Trace).	Constant

Part	Offset (Byte)	Size (Bytes)	Description	Value
bConnectionType	4	1	Constant that characterizes the type of Connection. 0: USB OUT endpoint 1: Debug Port input pin (Control, e.g., JTAG) 2: Debug Port input pin (Data) 3: Debug Port input pin (Data or Control) 4 – 127: *reserved*	Constant
bAssocConnection	5	1	ID of the Output Connection to which this Input Connection is associated, or zero (0) if no such association exists.	Constant
iConnection	6	1	Index of a string descriptor, describing the Input Connection	Constant
dTraceFormat (optional field)	7	4	Trace Format on the input pins to the Debug Unit. See Table 4-11	Constant
dStreamID (optional field)	11	4	ID for the output trace (e.g., Master ID for MIPI STP). The TS may change this value during a debug session. A Stream_ID = 0xFFFF indicates that there is no Stream_ID.	Constant

4.4.5 Output Connection Descriptor

The Output Connection descriptor describes functional aspects of the Output Connection to the host.

The value in the bConnectionID field uniquely identifies an Output Connection. No other Unit or Connection within the same debug capability may have the same ID. For example, each Connection and each Unit within DxC.Dfx shall have a unique ID, but DxC.Dfx and DxC.Trace can reuse the same ID.

The bConnectionType field defines where the Output Connection connects. This could be a USB IN endpoint, an external debug out connection, etc.

The bAssocConnection field associates an Input Connection to this Output Connection, effectively implementing a bi-directional Connection pair. For instance, this would link the JTAG input and output pins. If the bAssocConnection field is used, both associated Connections shall belong to the bi-directional Connection Type group. If no association exists, the bAssocConnection field shall be set to zero.

The Host software can treat the associated Connections as being physically related. In many cases, one Connection cannot exist without the other. An index to a string descriptor is provided to further describe the Output Connection.

The following table describes the Output Connection descriptor:

Table 4-8: Output Connection Descriptor

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	09h
bDescriptorType	1	1	CS_INTERFACE	24h
bDescriptorSubType	2	1	DC_OUTPUT_CONNECTION	02h
bConnectionID	3	1	A non-zero constant that uniquely identifies the Connection within the debug debug capability (DxC.Dfx or DxC.Trace).	Constant
bConnectionType	4	1	Constant that characterizes the type of Connection. 0: USB IN endpoint 1: Debug Port output pin (Control) 2: Debug Port output pin (Data) 3: Debug Port output pin (Data or Control) 4 – 127: *reserved*	Constant
bAssocConnection	5	1	ID of the Input Connection to which this Output Connection is associated, or zero (0) if no such association exists.	Constant
wSourceID	6	2	ID of the Unit or Connection to which the Input Pin of this Output Connection is connected in the first byte, and the output pin is in the second byte.	Constant
iConnection	8	1	Index of a string descriptor, describing the Output Connection	Constant

4.4.6 Debug-Unit Descriptor

This descriptor defines the type of debug unit (e.g., Dfx unit, Trace-Processing unit, Trace-Generation unit, etc.) together with connectivity information describing which output pins on which debug units are driving the input pins on this unit. Figure 4-8 gives an example interconnect between four Dfx Units and Table 4-9 defines the Debug-Unit descriptor fields.

The value in the bUnitID field of the Dfx-Unit descriptor uniquely identifies the debug Unit. No other Unit or Connection within the same DIC may have the same ID.

The bSourceID field describes the input connectivity for this Debug Unit. It contains the ID of the Unit or Connection to which this Debug Unit connects via its Input Pin, together with the output pin driving this input pin (see Figure 4-8)

The bDebugUnitType defines the type of debug unit (e.g., Trace-generation unit, Trace-Router unit, etc).

The Debug-Unit Descriptor allows the hardware designer to define any arbitrary debug functionality that the class driver passes to the host debugger application. This could be some special debug hardware, or debug associated hardware, such as authentication, or a test-pattern generator, that is not covered by this specification. This could also be a software application or structure.

The bNrInPins field defines the number of input pins. The wSourceID defines the connectivity of each of these input pins. The wSourceID consists of a pair of bytes: the first defines the unit ID driving this input, while the second byte defines the actual output pin driving the signal.

The bNrOutPins field defines the number of output pins.

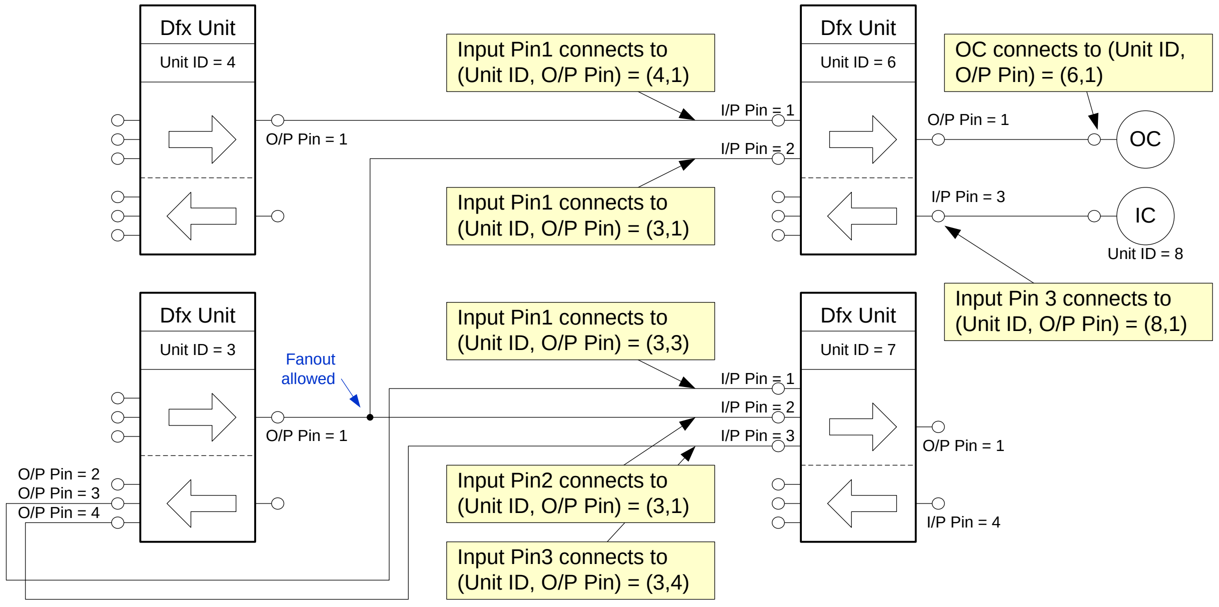

Dfx Unit
Unit ID = 4
O/P Pin = 1
Input Pin1 connects to (Unit ID, O/P Pin) = (4,1)
Dfx Unit
Unit ID = 6
OC connects to (Unit ID, O/P Pin) = (6,1)
I/P Pin = 1
I/P Pin = 2
O/P Pin = 1
OC
Input Pin1 connects to (Unit ID, O/P Pin) = (3,1)
I/P Pin = 3
IC
Unit ID = 8
Input Pin 3 connects to (Unit ID, O/P Pin) = (8,1)
Dfx Unit
Unit ID = 3
Fanout allowed
O/P Pin = 1
Input Pin1 connects to (Unit ID, O/P Pin) = (3,3)
Dfx Unit
Unit ID = 7
I/P Pin = 1
I/P Pin = 2
I/P Pin = 3
O/P Pin = 1
I/P Pin = 4
O/P Pin = 2
O/P Pin = 3
O/P Pin = 4
Input Pin2 connects to (Unit ID, O/P Pin) = (3,1)
Input Pin3 connects to (Unit ID, O/P Pin) = (3,4)

Figure 4-8: Example interconnect between a number of Dfx Units

The dTraceFormat fields define the trace format for each of the output pins of the debug unit. Note that the inputs to the Dfx unit could each have a different trace format, which the Dfx unit may convert into a set of different formats on its output pins. See Figure 4-2 for an example.

The dStreamID give the source of a Trace/Stream Protocol. This value may change during a debug session. Debug Software running on the TS may thus update this value. See Section 4.1.1 for more details.

The qBaseAddress provides a 64-bit address to the Configuration registers in the debug unit.

The bmControls field is a bitmap, indicating the availability of certain debug controls for the debug unit stream. For future expandability, the number of bytes occupied by the bmControls field is indicated in the bControlSize field. The bControlSize field is permitted to specify a value less than the value needed to cover all the control bits (including zero), in which case the unspecified bmControls bytes will not be present and these control bits are implicitly zero.

A non-zero wAuxDataSize indicates that the descriptor contains a data structure for supplemental debug data. This data structure consists of:

- The qBaseAddress field pointing to a implementation-specific, Debug-Data structure
- A GUID for this debug unit in the hIPID field. The hIPID provides an ID (e.g., GUID) for the actual implementation of the debug unit. For example, a certain IP block may have implemented an early version of a protocol and may thus suffer from a number of limitations. The hIPID allows the debugger to recognize such units and act accordingly.

A non-zero wStandardsDataSize indicates that the descriptor contains a data structure for supplemental debug data defined by a Standards body. See Section 4.5 for more details and examples. A Standards body can define a debug unit, and there could be many debug units within a TS defined by different standards. This data structure consists of:

- An identifier, bStandardsID, indicating the Standards body
- Standards Data, which is a data structure defined by the Standards body.

A non-zero wVendorDataSize indicates that the descriptor contains a data structure for supplemental debug data defined by the vendor. For example, the vendor may need to know the state of the TS prior to the start of a debug session. They may use this data field for this purpose.

Note that the standard GET_DESCRIPTOR request can fetch at most 64KB. Thus the complete descriptor, including any Standards body's and Vendor's extensions must not exceed this limit.

An index to a string descriptor (iDebugUnitType) is provided to further describe the Debug Unit.

The following table defines the Debug-Unit descriptor:

Table 4-9: Debug Unit Descriptor

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	41 + 2p
bDescriptorType	1	1	CS_INTERFACE	24h
bDescriptorSubType	2	1	DC_DEBUG_UNIT	03h
bcdDC	3	2	Revision number of Debug Class specification that this Debug Unit is based on.	0100h (rev 1)
bUnitlD	3	1	A non-zero constant that uniquely identifies the Debug Class unit.	Constant

Part	Offset (Byte)	Size (Bytes)	Description	Value
bDebugUnitType	4	1	0: Un-defined unit 1: Dfx Unit 2: Select Unit 3: Trace-Router Unit 4: Trace-Processing Unit 5: Trace-Generation Unit 6: Trace-Sink Unit 7: Control Unit 8-63: *reserved* 64:95: Vendor Specific 96:127: For use by Standards body 128-255: *reserved*	xxh
bDebugSubUnitType	5	1	E.g., Audio, GFX, Core, Modem, etc. – see Table 4-10.	
bAliasUnitID	6	1	ID of the Debug Unit to which this debug unit is associated, or zero (0) if no such association exists. For example, a Trace-Generator unit only has a single output and thus can only generate a single output trace. However, a CPU core may generate multiple trace streams, e.g., software messages from the OS and also processor-instruction traces. In this case, there will be two Trace-Generator units, and the bAliasUnitID field would link these together, indicating that they are the same physical device. This information maybe useful for the DTS when it is powering on/off debug units. If 3 or more units need to be aliased together, then arbitrarily choose one as the reference unit that the others will alias to.	Constant
bNrInPins	7	1	Number of Input Pins on this Unit: p	Constant
wSourceID(1)	8	2	ID of the Unit or Connection to which the first Input Pin of this Debug Unit is connected in the first byte, and the output pin is in the second byte.	Constant
⋮	⋮	⋮	⋮	

Part	Offset (Byte)	Size (Bytes)	Description	Value
wSourceID(p)	8 + 2(p-1)	2	ID of the Unit or Connection to which the pth Input Pin of this Debug Unit is connected in the first byte, and the output: pin is in the second byte.	Constant
bNrOutPins	8 + 2p	1	Number of Output Pins on this Unit: q	Constant
dTraceFormat(1)	9 + 2p	4	Trace Format on the first output pins for the Debug Unit. See Table 4-11	Constant
dStreamID(1)	13+2p	4	ID for the the above output trace (e.g., Master ID for MIPI STP). The TS may change this value during a debug session. It is implementation specific how the Debugger reads the new value at the end of the debug session. 0xFFFF: Null (no StreamID)	Constant
⁞	⁞	⁞	⁞	
dTraceFormat(q)	9 + 2p + 8q	4	Trace Format on the qth output pins for the Debug Unit. See Table 4-11 Note: If all outputs have the same trace then all of these fields will be the same.	Constant
dStreamID (q)	13+2p + 8q	4	ID for the output trace (e.g., Master ID for MIPI STP). 0xFFFF: Null (no StreamID)	Constant
bControlSize	17+2p + 8q	1	Size of the bmControls field, in bytes: n	Number

Part	Offset (Byte)	Size (Bytes)	Description	Value
bmControl (See Section 5.1 for more information)	18+2p + 8q	n	A bit set to 1 indicates that the mentioned Debug-Control request is supported: D0: SET_CCONFIG_DATA_SINGLE D1: SET_CONFIG_DATA D2: GET_CCONFIG_DATA D3: SET_CONFIG_ADDRESS D4: GET_CONFIG_ADDRESS D5: *reserved* D6: *reserved* D7: SET_OPERATING_MODE D8: GET_OPERATING_MODE D9: *reserved* D10: *reserved* D11: SET_BUFFER D12: GET_BUFFER D13: SET_RESET D23..D14: *reserved* D31..D24: Vendor-specific If SET_CONFIG_ADDRESS is not supported but SET/GET_CONFIG_DATA is supported, then Configuration address defaults to value 0.	Bitmap
bAuxDataSize	18+2p + 8q+n	1	This field defines the size of the next two fields. If there is no auxiliary data then = 0, otherwise 24.	Number
qBaseAddress	19+2p + 8q+n	8	Base Address to the configuration registers of the debug IP block. If there are no configuration registers then qBaseAddress = 0	Constant
hGUID	27+2p + 8q+n	16	Global-unique identifer (GUID) for the IP	Constant
wStandardsDataSize	45+2p + 8q+n	2	This fields defines the size of the next two fields, which are p bytes in size. If there is no Standards data then p = 0.	Number

Part	Offset (Byte)	Size (Bytes)	Description	Value
bStandardsID	47+2p + 8q+n	1	0: Data format is not a Standard 1: MIPI Standards Organization 2: IEEE Standards Organization otherwise: *reserved*	Number
Standards Data	48+2p + 8q+n	p-1	Standards data. For example, which Specification, bit mask for supported commands, buffer sizes, etc. Size is (p-1) bytes	Constant
wVendorDataSize	47+2p +8q+n +p	2	This field defines the size of the remaining bytes in the descriptor in bytes: q	Number
Vendor Data	49+2p +8q+n +p	q	Vendor defined data (q bytes)	Number
iDebugUnitType	49+2p +8q+n +p+q	1	Index of a string descriptor, identifing the Debug Unit (and Sub-unit) type	Index

Table 4-10: Debug Sub-Unit Type

Part	Description	Value
bDebugSubunitType	Constant that characterizes the subtype of Unit: 0: Not defined (Null unit) 1: CPU 2: Graphics 3: Video 4: Imaging 5: Audio 6: Modem 7: Bluetooth 8: Power-Management agent 9: Security agent 10: Sensor Unit 11: Bus-Watcher 12: Location (GNSS, GPS, Glonass) 13: Trace Compressor 14: TAP Controller 15: Memory Access Unit 16: Configuration Unit 17 - 62: *reserved* 63: Other 64: SW Trace Logger 65: SW Router 66: SW Unit 67: SW Configuration Unit 68: SW Debugger 69 - 127: *reserved* 128 - 191: Vendor Specific 192-254: *reserved* 255: Standards Body	Constant

Table 4-11: dTraceFormat

dTraceFormat <31:24>	Type/ Vendor	dTraceFormat <23:0>	Value
0x00	N/A	0: Pass-through (no change of trace format) 1: Debug Header Format (see Appendix C:) 2: Debug Footer Format (see Appendix C:) 3 - 4: *reserved* 5: Use GUID for trace format (i.e., Proprietary trace format) 6: UTF8 string format Otherwise: *reserved*	Constant

dTraceFormat <31:24>	Type/ Vendor	dTraceFormat <23:0>	Value
0x01	Intel	Vendor defined	Constant
0x02	ARM	ARM defined	Constant
0x03	ST	Vendor defined	Constant
0x04	TI	Vendor defined	Constant
0x05	Qualcomm	Vendor defined	Constant
0x06	AMD	Vendor defined	Constant
0x07-0x7F	*reserved*		Constant
0x80	MIPI Standards	MIPI defined	Constant
0x81	Nexus Standards	Nexus defined	Constant
Otherwise	*reserved*		Constant

4.5 Standards-Body Support

Standard bodies can extend the Debug Class to support a new debug function by defining extensions to the Debug-Unit descriptor. Figure 4-9 shows an example of a new debug unit, a Bus-Watcher, which sends trace packets to the DxC.Trace interface when it observes specific bus traffic. The standards-body could define a set of commands pertinent to this unit, such as SET_MATCH_ADDRESS, SET_MATCH_DATA, and SET_ENABLE_BUS_WATCHER. This Debug Class specification does not define these new commands; instead, it allows a standards body to define these commands and capabilities as an adjunct to this specification.

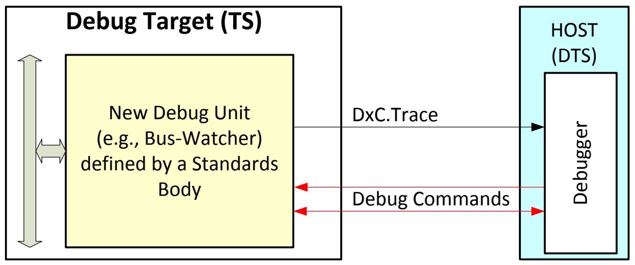

Debug Target (TS)
New Debug Unit (e.g., Bus-Watcher) defined by a Standards Body
DxC.Trace
Debug Commands
HOST (DTS)
Debugger

Figure 4-9: New Debug Function created by a Standards Body accessed via the Debug Class

A standards body shall use a Debug Unit descriptor to define its commands and capabilities. See Figure 4-10. Note that the Debug-Attributes descriptor also defines commands, but these are Debug Class-specific, and a standards body cannot change or extend these. Furthermore, the Debug-Attribute descriptor defines the commands pertinent to the TS or DIC level; the Debug-unit descriptor defines commands pertinent to a debug unit. Hence, a standards body cannot define commands for the TS or the DIC; it can only define commands for a Debug unit.

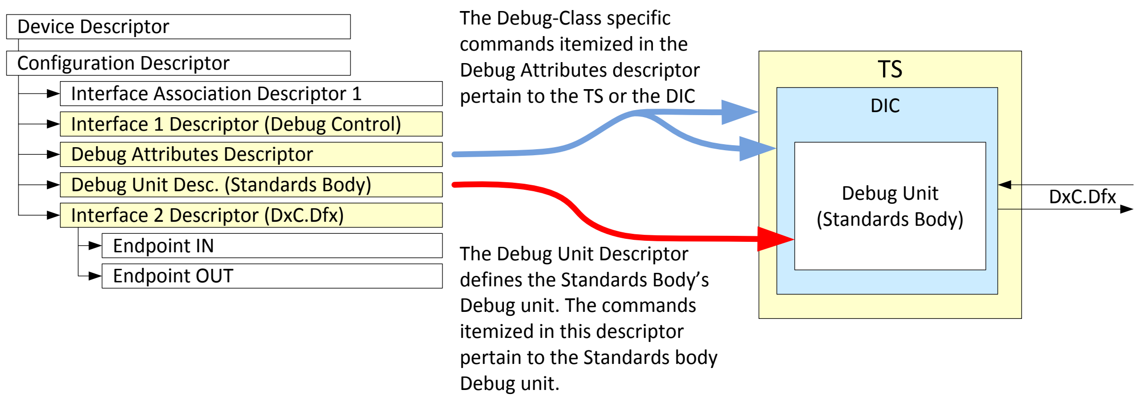

Device Descriptor
Configuration Descriptor
Interface Association Descriptor 1
Interface 1 Descriptor (Debug Control)
Debug Attributes Descriptor
Debug Unit Desc. (Standards Body)
Interface 2 Descriptor (DxC.Dfx)
Endpoint IN
Endpoint OUT
The Debug-Class specific commands itemized in the Debug Attributes descriptor pertain to the TS or the DIC
The Debug Unit Descriptor defines the Standards Body's Debug unit. The commands itemized in this descriptor pertain to the Standards body Debug unit.
TS
DIC
Debug Unit (Standards Body)
DxC.Dfx

Figure 4-10: Standards bodies can only define commands at the Debug-unit level

For illustrative purposes, Table 4-12 shows the fields of the Debug-Unit descriptor pertinent to a Standards body. The hypothetical values are for a Dfx unit defined by the MIPI Standards body.

Table 4-12: Debug-Unit Descriptor fields for an Example Standard Body's Dfx Unit

Part	Value	Description
bDebugUnitType	0	A Dfx Unit
bDebugSubUnitType	255	Standards Body
…		
wStandardsDataSize	6	The next 6 bytes of the descriptor pertain to the Standards body
bStandardsID	1	A MIPI standard

Part	Value	Description
Standards Data	Number	5 Byte Data structure defined by the MIPI Standards organization. For example, this data structure could define the Specification version number; a bit mask for supported commands; etc. See Table 4-13 for an example of a 5-Byte structure.
…		

Table 4-13 is a hypothetical example of a possible Standards-Data structure, given purely for illustrative purposes. This example provides a specification version number, a bitmask for the supported commands together with a size field defining this length of thus bitmask. The tabulated commands are hypothetical and are given purely as an example.

Table 4-13: Example of Standards Data

Part	Offset (Byte)	Size (Bytes)	Description	Value
bcdStandard	0	2	Revision of Standards body specification.	0100h
bStandCntrlSize	2	1	Size of the bmStandControl field, in bytes: n	Number
bmStandControl	3	2	A bit set to 1 indicates that the mentioned Standard-body specific Control request is supported: D0: SET_INITIALIZE D1: SET_ADDRESS_TRIGGER D2: GET_ADDRESS_TRIGGER D3: SET_DATA_TRIGGER D4: GET_DATA_TRIGGER D5: *reserved* D6: *reserved* D7: SET_START D8: SET_HALT D9..D15: *reserved*	Bitmap

A TS may contain multiple, different debug units, each defined by a different Standards body. Figure 4-11 shows an example of a TS containing two such debug units.

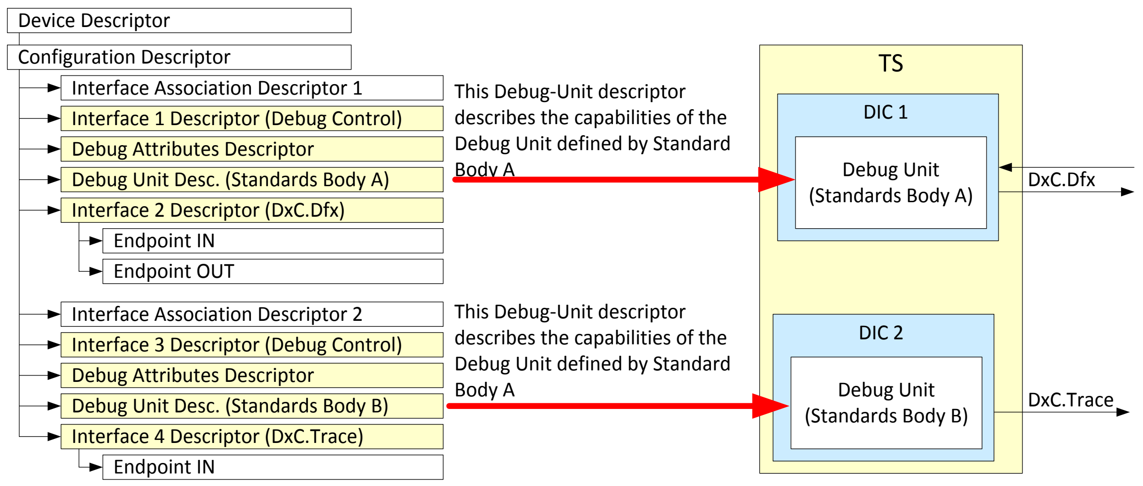

Device Descriptor
Configuration Descriptor
Interface Association Descriptor 1
Interface 1 Descriptor (Debug Control)
Debug Attributes Descriptor
Debug Unit Desc. (Standards Body A)
Interface 2 Descriptor (DxC.Dfx)
Endpoint IN
Endpoint OUT
Interface Association Descriptor 2
Interface 3 Descriptor (Debug Control)
Debug Attributes Descriptor
Debug Unit Desc. (Standards Body B)
Interface 4 Descriptor (DxC.Trace)
Endpoint IN
This Debug-Unit descriptor describes the capabilities of the Debug Unit defined by Standard Body A
This Debug-Unit descriptor describes the capabilities of the Debug Unit defined by Standard Body A
TS
DIC 1
Debug Unit (Standards Body A)
DxC.Dfx
DIC 2
Debug Unit (Standards Body B)
DxC.Trace

Figure 4-11: A TS containing two Debug units defined by different Standards bodies

5 Class-Specific Requests

5.1 Introduction

The Debug Class-specific requests are an optional set of commands that allow the debugger to perform basic operations on the debug logic via the default endpoint. These include reads and writes (i.e., GET and SET) of data structures (e.g., configuration registers); the enabling of power-management modes; the selection of a particular trace generation configuration; the ability to select a backup core for the USB stack on the TS in case the main OS hangs[4]; and so on. Table 5-1 lists the available commands.

[4] Presumably, the TS will use hardware decode for this request, otherwise it defeats the purpose of providing this command

The Debug Commands are completely optional, but if they are supported then the following descriptors are mandatory:

- If TS supports ANY debug commands then
    - (Debug Command Interface + Debug Attributes) Descriptor are mandatory
        - Debug Attributes defines the supported TS commands
    - GET_INFO & GET_ERROR are mandatory to TS, DIC, or Unit even if there is no DIC or Unit (i.e., fail safe if the SW mistakenly targeted the wrong thing).
- If DIC supports ANY debug commands then
    - IAD is mandatory to create DIC
    - (Debug Command Interface + Debug Attributes) Descriptor are mandatory
        - Debug Attributes defines the supported DIC commands
    - GET_INFO & GET_ERROR are mandatory to TS, DIC, or Unit
- If a Debug Unit supports ANY debug commands then
    - IAD is mandatory to create DIC containing the Unit
    - (Debug Command Interface + Debug Attributes) Descriptor are mandatory for the DIC containing the Unit
        - Debug Unit Descriptor defines the supported Unit commands
    - GET_INFO & GET_ERROR are mandatory to TS, DIC, or Unit

Figure 5-1 is an example showing which of the various bmControl fields of the Debug Attributes and the Debug-Unit descriptors define the commands supported by that particular level (TS, DIC, or Debug Unit).

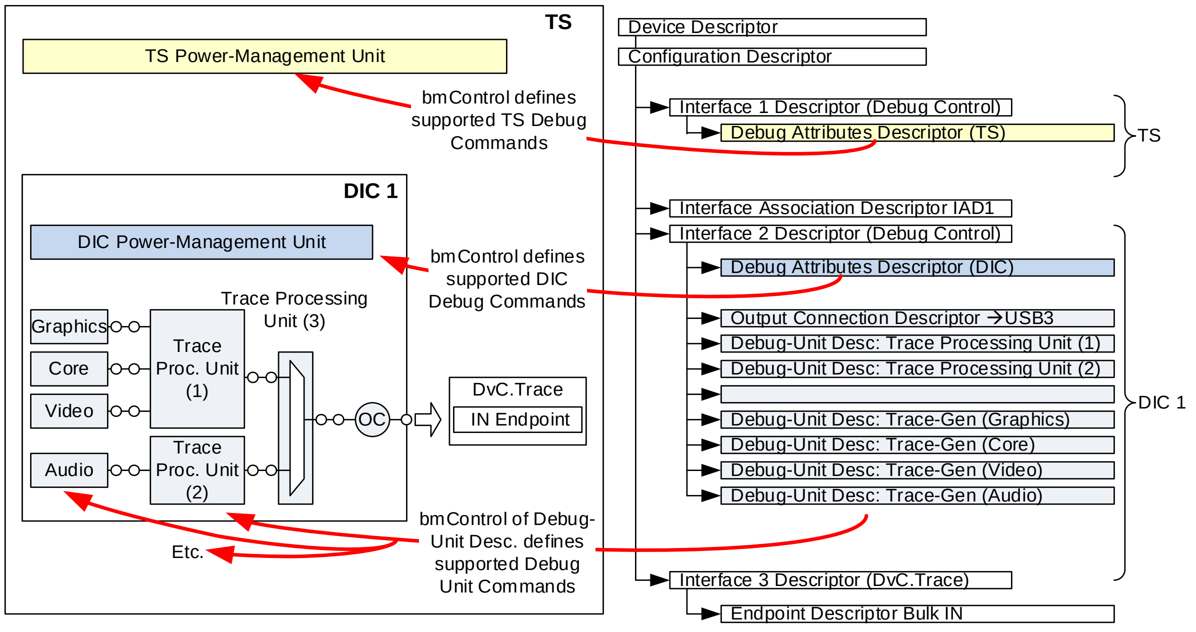

TS
TS Power-Management Unit
bmControl defines supported TS Debug Commands
DIC 1
DIC Power-Management Unit
bmControl defines supported DIC Debug Commands
Trace Processing Unit (3)
Graphics
Core
Video
Audio
Trace Proc. Unit (1)
Trace Proc. Unit (2)
OC
DvC.Trace
IN Endpoint
Etc.
bmControl of Debug-Unit Desc. defines supported Debug Unit Commands
Device Descriptor
Configuration Descriptor
Interface 1 Descriptor (Debug Control)
Debug Attributes Descriptor (TS)
TS
Interface Association Descriptor IAD1
Interface 2 Descriptor (Debug Control)
Debug Attributes Descriptor (DIC)
Output Connection Descriptor →USB3
Debug-Unit Desc: Trace Processing Unit (1)
Debug-Unit Desc: Trace Processing Unit (2)
Debug-Unit Desc: Trace-Gen (Graphics)
Debug-Unit Desc: Trace-Gen (Core)
Debug-Unit Desc: Trace-Gen (Video)
Debug-Unit Desc: Trace-Gen (Audio)
DIC 1
Interface 3 Descriptor (DvC.Trace)
Endpoint Descriptor Bulk IN

Figure 5-1: Which bmControl field defines the Debug Control for the TS, DIC, and Unit level

The Debug Class only supports a basic, limited set of debug requests, because there is considerable variation in the debug hooks provided by the chip vendors, and thus it is difficult to define more extensive commands. For example, a bus-watcher requires address, command, and data filters, but there is considerable variation in the number of such filters, their masking capabilities, and so on. Furthermore, some bus-watchers observe proprietary sideband signals, while others provide event counters with varying outcomes, and so on. Consequently, it is difficult to develop a set of generic debug commands that are applicable to the majority of designs.

Nonetheless, the set of requests provided are still valuable, and a Standards body could develop their own set of commands, using these as a basis. The current set of commands allow the following useful scenarios:

- The Debug-Control interface allows for a low-cost debug trace solution using a single DxC.Trace endpoint. In this scenario, the debugger uses the default endpoint 0 to configure and enable the TS to generate debug traces. Otherwise, the debugger would require additional endpoints to provide a means to configure the device, such as a COM-type interface via DxC.GP or a TAP-type interface via DxC.Dfx. Such a single-endpoint, debug solution is very attractive for low-cost devices.
- A TS may support just the Debug-Control interface and no other DxC interface. In this case, there will be no endpoints used for debug apart from the default endpoint 0. Figure 5-2 shows a possible scenario. In this example, the Debug Class specific SET_CONFIG_DATA requests configure the Trace Unit and later extract trace data from the memory buffer.

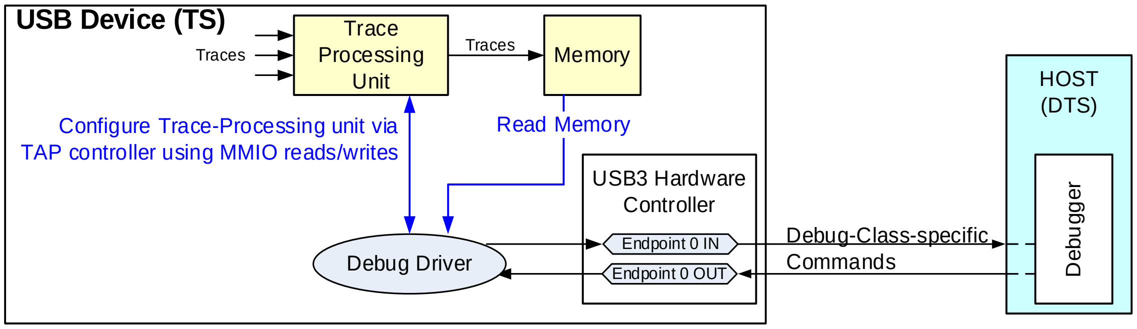


Figure 5-2: Debug Example using only the Debug-Control Interface

5.2 Debug-Control Overview

The Debug-Control requests may target the complete TS, or a particular DIC, or a specific Unit. For example, the debugger may wish to power-down all of the debug-related logic in the complete TS; or perhaps, the debugger may wish to power-down the trace logic but leave the TAP logic powered on; or the debugger may wish to power-down the debug-trace logic associated with the graphics unit, but maintain power on all the other trace-generation units. Table 5-1 shows whether the Debug Control requests target the Global entity (i.e., complete TS), a Local entity (i.e., DIC) or a Specific entity (i.e., Debug unit).

Table 5-1: Debug-Control Request Resolution

Debug Control Request	Global (Complete TS)	Local (DIC)	Specific (Debug Unit)
SET/ GET_CONFIG_DATA SET_CONFIG_DATA_SINGLE	Yes	Yes	Yes
SET/GET_CONFIG_ADDRESS	Yes	Yes	Yes
SET/GET_OPERATING_MODE	Yes	Yes	Yes
SET/GET_TRACE	Yes	Yes	Yes
SET/GET_ALT_STACK	Yes	N/A	N/A
SET/GET_BUFFER	Yes	Yes	Yes

Debug Control Request	Global (Complete TS)	Local (DIC)	Specific (Debug Unit)
SET_RESET	Yes	Yes	Yes
GET_INFO	Yes	Yes	Yes
GET_ERROR	Yes	Yes	Yes

The actual commands supported by the TS, DIC or Debug unit are defined via the bmControl fields in the Debug-Attributes descriptor for the TS and for the DIC, and in the Debug-Unit descriptor for the debug unit. Thus, after enumeration, the debugger will know which commands are supported by which entities. See Figure 5-1 for an example.

If a Debug unit supports a command, then this does not imply that the DIC also supports this command. For example, a DIC may consist of a collection of Trace-Processing units, some of which provide the ability to be powered-down. However, the DIC itself may not provide the facility to be completely powered down. Thus, the bmControl field of the Debug-Attribute descriptor is not inclusive of its children debug units.

The Debug-Attribute Descriptor may have the bmControl = 0, indicating that the DIC supports no Control requests. In this case, the Debug-Control descriptors simply define the debug topology and the trace format.

If a device supports any Debug Control Requests then it shall support the mandatory requests (i.e., Get Error & Get Info). If a debug function does not support a certain request, it shall indicate this by *Stalling* the control pipe when that request is issued to access the function.

5.3 Request Layout

The following paragraphs describe the general structure of the SET and GET requests. Subsequent paragraphs detail the use of the SET/GET requests for the different request types

5.3.1 Request Layout

The SET requests are used to set an attribute of a Control inside an entity of the debug function, and the GET requests read the attribute inside the entity. Table 5-2 shows the fields for a USB request as used by the Debug class.

Table 5-2: SET and GET Requests

bmRequestType	bRequest	wValue	wIndex	wLength
00100001	SET_CONFIG_DATA SET_CONFIG_DATA_ADDRESS SET_ ALT_STACK SET_OPERATING_MODE SET_TRACE SET_BUFFER SET_RESET	See following paragraphs. Typically use wValue<7:0> to select between Global and	See following paragraphs. Typically used to select between Local and Specific	Length of the Data Paramet er block

bmRequestType	bRequest	wValue	wIndex	wLength
10100001	GET_CONFIG_DATA GET_CONFIG_DATA_ADDRESS GET_ALT_STACK GET_OPERATING_MODE GET_TRACE GET_ BUFFER GET_INFO GET_ERROR	Local entity (see Table 5-5) wValue<15:8> typically used for sub-commands	unit (see Table 5-5)	

The bmRequestType field specifies that this is a SET request (D7=0) or a GET request (D7=1). It is a class-specific request (D6..5=01), directed to a Debug interface (D4..0=00001).

The bRequest field contains a constant that identifies which attribute of the addressed Control is to be modified. Possible attributes for a Control are:

- Write and Read to the Debug Data Structure/Configuration register (SET_CONFIG_DATA, GET_CONFIG_DATA)
- Write and Read the Power Mode control (SET_OPERATING, GET_OPERATING)
- Set a Trace configuration (SET_TRACE, GET_TRACE)
- Access Buffer size information in the Debug Function (SET_BUFFER, GET_BUFFER)
- Restore the Debug function to its default state (SET_RESET)
- Read the Error state pertaining to the USB command (GET_ERROR)
- Read the Information state pertaining to the available USB commands (GET_INFO)

If the addressed Control or entity does not support modification of a certain attribute, the control pipe shall gracefully ignore the request by indicating a *Stall* when an attempt is made to modify that attribute.

The wValue field interpretation is qualified by the value in the wIndex field. These two fields are typically used to select between the TS (Global), DIC (Local) or a specific Debug unit. See Table 5-3. Depending on what entity is addressed, the layout of the wValue field changes. Later sections describe the contents of the wValue field for each entity separately.

The low byte of the wIndex field specifies the interface to be addressed, and the high byte specifies the Unit ID of the debug unit or zero. If the wIndex is addressing an interface, then the virtual entity "interface" can be addressed by specifying zero in the high byte (i.e., Unit ID = 0). In general, one can make a request to a global structure (e.g., the configuration registers for the complete TS), a local structure within a DIC, and a specific structure within a debug unit (e.g., see Table 5-5).

Table 5-3: Debug Command Selection of TS, DIC, and Unit

Command Target	wValue	wIndex
TS	00 00	00 ii Where ii is the interface number of any arbitrary Debug-Control Interface in any DIC in the TS.
DIC interface #ii	00 01	00 ii Where ii is the interface number of the desired Debug-Control Interface.
Unit #uu of DIC #ii	xx xx (Don't care) Note: Unit ID = 0 is *reserved* for accessing the complete TS or a	uu ii

	DIC. Hence, the wValue is ignored if uu ≠ 0	

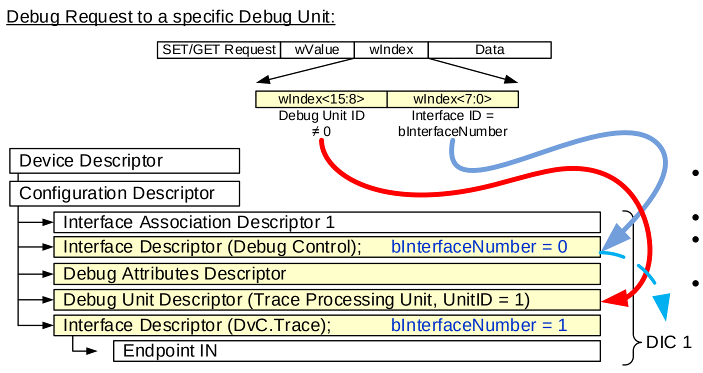
Section 5.3.2 gives examples to clarify the addressing capabilities.

The values in wIndex shall be appropriate to the recipient. Only existing entities in the debug function can be addressed, and only appropriate interface numbers may be used. If the request specifies an unknown unit ID or an unknown interface then the control pipe shall indicate a stall.

The actual parameter(s) for the SET/GET request are passed in the data stage of the control transfer. The length of the parameter block is indicated in the wLength field of the request. The layout of the parameter blocks are given later.

5.3.2 Request Examples

Figure 5-3 and Figure 5-4 illustrate and explain how the Debug commands access the TS, a DIC, or a specific debug unit. Figure 5-3 illustrates how to access a specific Debug unit, while Figure 5-4 illustrates how to access a specific DIC. The explanation within Figure 5-4 also explains how to access the complete TS.


Debug Request to a specific Debug Unit:
SET/GET Request
wValue
wIndex
Data
wIndex<15:8>
Debug Unit ID ≠ 0
wIndex<7:0>
Interface ID = bInterfaceNumber
Device Descriptor
Configuration Descriptor
Interface Association Descriptor 1
Interface Descriptor (Debug Control); bInterfaceNumber = 0
Debug Attributes Descriptor
Debug Unit Descriptor (Trace Processing Unit, UnitID = 1)
Interface Descriptor (DvC.Trace); bInterfaceNumber = 1
Endpoint IN
DIC 1

- wIndex<7:0> is the Interface ID (i.e., bInterfaceNumber) of the Debug-Control Descriptor within the desired DIC
- Thus, the Interface ID indirectly defines the desired DIC
- A non-zero Debug-Unit ID defines the debug unit within this DIC
- E.g., to access the Trace Processing unit of DIC1, the wIndex = 0x0100 (i.e., Debug-unit ID = 1 & Interface ID = 0)

Figure 5-3: Debug Control accessing a specific Debug Unit

A non-zero wIndex<15:8> field defines the desired debug unit that the debug command is targeting. If wIndex<15:8> = 0 (i.e., Unit ID = 0), then the command is targeting the TS or the DIC, where the wValue<1> bit defines whether the command is targeting the complete TS or a specific DIC. When targeting the TS, then the command is free to point to any of the DICs within the TS.

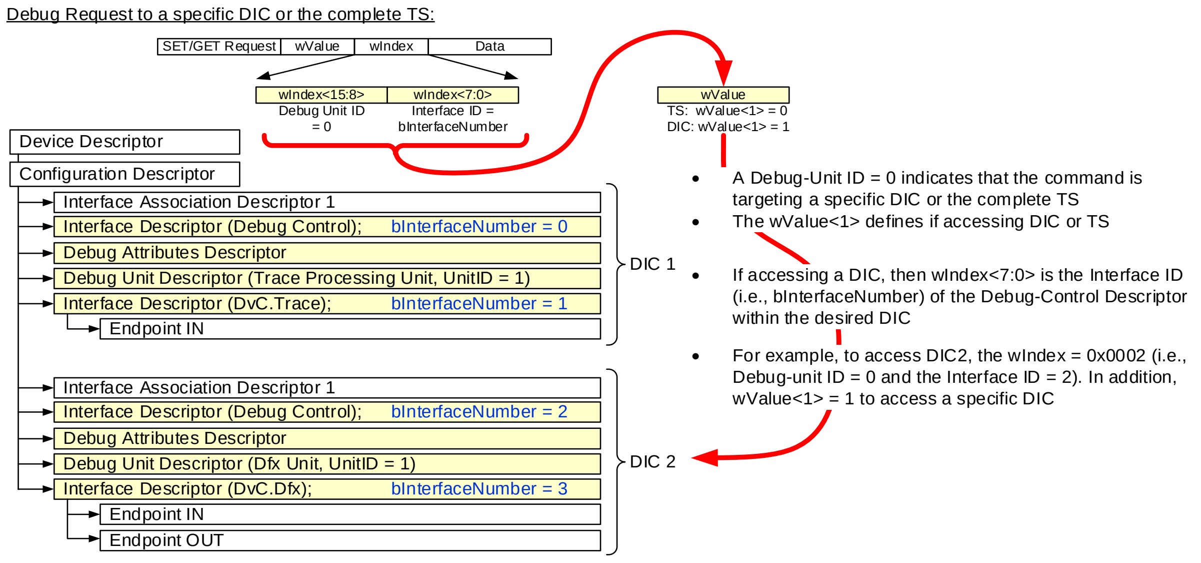

Debug Request to a specific DIC or the complete TS:
SET/GET Request
wValue
wIndex
Data
wIndex<15:8> Debug Unit ID = 0
wIndex<7:0> Interface ID = bInterfaceNumber
wValue TS: wValue<1> = 0 DIC: wValue<1> = 1
Device Descriptor
Configuration Descriptor
Interface Association Descriptor 1
Interface Descriptor (Debug Control); bInterfaceNumber = 0
Debug Attributes Descriptor
Debug Unit Descriptor (Trace Processing Unit, UnitID = 1)
Interface Descriptor (DvC.Trace); bInterfaceNumber = 1
Endpoint IN
DIC 1
Interface Association Descriptor 1
Interface Descriptor (Debug Control); bInterfaceNumber = 2
Debug Attributes Descriptor
Debug Unit Descriptor (Dfx Unit, UnitID = 1)
Interface Descriptor (DvC.Dfx); bInterfaceNumber = 3
Endpoint IN
Endpoint OUT
DIC 2
A Debug-Unit ID = 0 indicates that the command is targeting a specific DIC or the complete TS
The wValue<1> defines if accessing DIC or TS
If accessing a DIC, then wIndex<7:0> is the Interface ID (i.e., bInterfaceNumber) of the Debug-Control Descriptor within the desired DIC
For example, to access DIC2, the wIndex = 0x0002 (i.e., Debug-unit ID = 0 and the Interface ID = 2). In addition, wValue<1> = 1 to access a specific DIC

Figure 5-4: Debug Control accessing a specific DIC or the complete TS

Figure 5-5 is an example of a GET_CONFIG_DATA request to a global data structure. This is the highest-level data structure corresponding to the complete debug entity (i.e., the TS, which may consist of a collection of chips if the debug topology spans multiple chips).

Note that an access to the global data structure via any of the Debug-Control Interface descriptors (e.g., interface 1 and 3 in Figure 5-5) aliases to the same data structure.

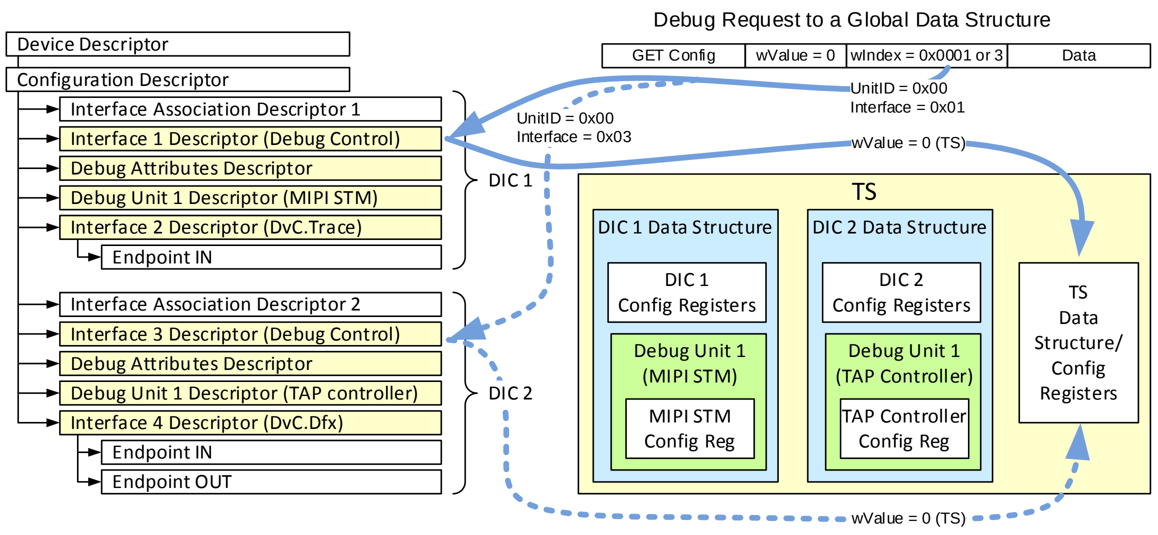

Debug Request to a Global Data Structure
GET Config
wValue = 0
wIndex = 0x0001 or 3
Data
UnitID = 0x00 Interface = 0x01
UnitID = 0x00 Interface = 0x03
wValue = 0 (TS)
Device Descriptor
Configuration Descriptor
Interface Association Descriptor 1
Interface 1 Descriptor (Debug Control)
Debug Attributes Descriptor
Debug Unit 1 Descriptor (MIPI STM)
Interface 2 Descriptor (DvC.Trace)
Endpoint IN
DIC 1
Interface Association Descriptor 2
Interface 3 Descriptor (Debug Control)
Debug Attributes Descriptor
Debug Unit 1 Descriptor (TAP controller)
Interface 4 Descriptor (DvC.Dfx)
Endpoint IN
Endpoint OUT
DIC 2
TS
DIC 1 Data Structure
DIC 1 Config Registers
Debug Unit 1 (MIPI STM)
MIPI STM Config Reg
DIC 2 Data Structure
DIC 2 Config Registers
Debug Unit 1 (TAP Controller)
TAP Controller Config Reg
TS Data Structure/ Config Registers
wValue = 0 (TS)

Figure 5-5: Debug Request to Global Configuration Registers

Note that one can also target a Debug Command directly if there is a Debug-Control Interface & Debug-Attributes descriptor pair dedicated to the TS, as per the example in Figure 4-7. In this case, we specifically address that Debug-Control Interface using the scheme shown below in Figure 5-6 (i.e., wValue = 0x0001 and wIndex points to the Debug-Control Interface associated with the TS).

Figure 5-6 is an example of a GET_CONFIG_DATA request to a local (DIC) data structure. Unlike the previous global request, an access to a specific Debug-Control Interface descriptor (e.g., interface 1 or 3 in Figure 5-6) access the data structure for that corresponding DIC. This figure shows two examples of GET_CONFIG_DATA: one to interface 1 and a second example (in dotted arrows) to interface 3.

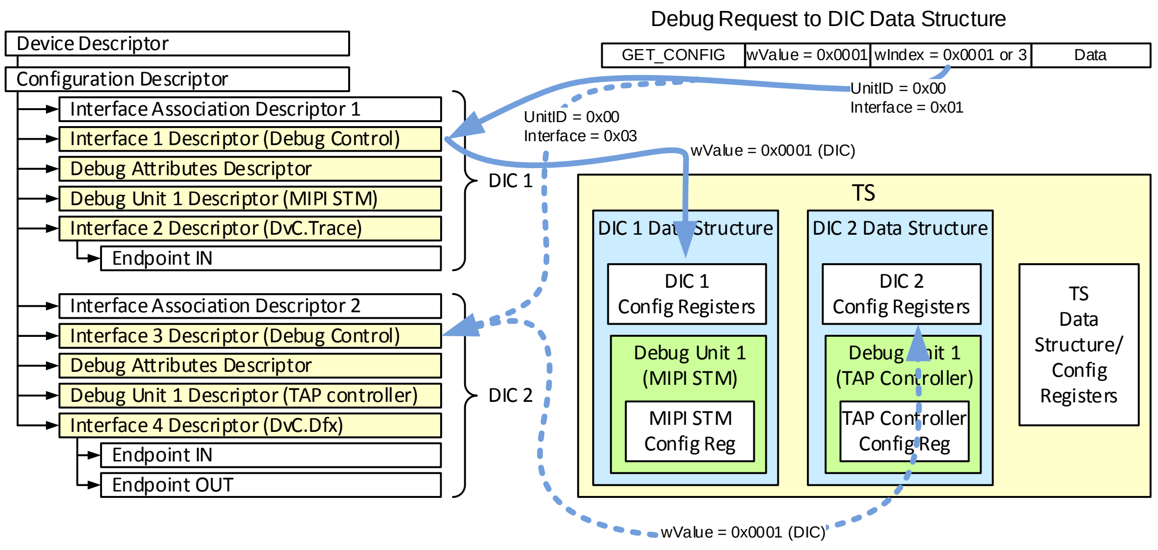

Debug Request to DIC Data Structure
GET_CONFIG
wValue = 0x0001
wIndex = 0x0001 or 3
Data
UnitID = 0x00
Interface = 0x01
UnitID = 0x00
Interface = 0x03
wValue = 0x0001 (DIC)
Device Descriptor
Configuration Descriptor
Interface Association Descriptor 1
Interface 1 Descriptor (Debug Control)
Debug Attributes Descriptor
Debug Unit 1 Descriptor (MIPI STM)
Interface 2 Descriptor (DvC.Trace)
Endpoint IN
DIC 1
Interface Association Descriptor 2
Interface 3 Descriptor (Debug Control)
Debug Attributes Descriptor
Debug Unit 1 Descriptor (TAP controller)
Interface 4 Descriptor (DvC.Dfx)
Endpoint IN
Endpoint OUT
DIC 2
TS
DIC 1 Data Structure
DIC 1 Config Registers
Debug Unit 1 (MIPI STM)
MIPI STM Config Reg
DIC 2 Data Structure
DIC 2 Config Registers
Debug Unit 1 (TAP Controller)
TAP Controller Config Reg
TS Data Structure/ Config Registers
wValue = 0x0001 (DIC)

Figure 5-6: Debug Request to DIC Configuration Registers

Figure 5-7 is an example of a GET_CONFIG_DATA request to a specific data structure within a debug unit (e.g., a MIPI STM unit). This is the lowest-level data structure corresponding to a specific Unit ID. In this case, the wIndex defines the Unit ID within a particular DIC.

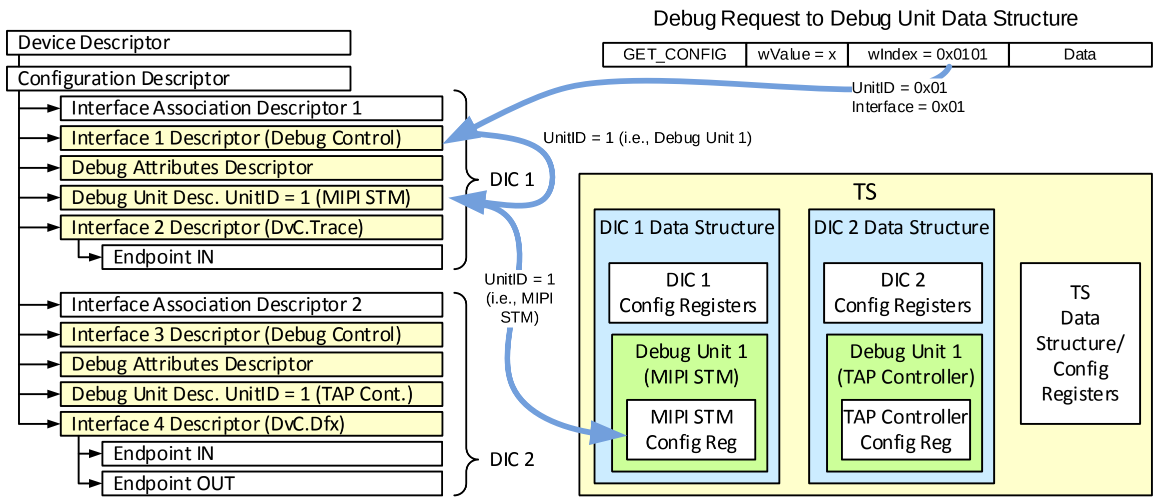

Debug Request to Debug Unit Data Structure
GET_CONFIG
wValue = x
wIndex = 0x0101
Data
UnitID = 0x01
Interface = 0x01
UnitID = 1 (i.e., Debug Unit 1)
UnitID = 1 (i.e., MIPI STM)
Device Descriptor
Configuration Descriptor
Interface Association Descriptor 1
Interface 1 Descriptor (Debug Control)
Debug Attributes Descriptor
Debug Unit Desc. UnitID = 1 (MIPI STM)
Interface 2 Descriptor (DvC.Trace)
Endpoint IN
DIC 1
Interface Association Descriptor 2
Interface 3 Descriptor (Debug Control)
Debug Attributes Descriptor
Debug Unit Desc. UnitID = 1 (TAP Cont.)
Interface 4 Descriptor (DvC.Dfx)
Endpoint IN
Endpoint OUT
DIC 2
TS
DIC 1 Data Structure
DIC 1 Config Registers
Debug Unit 1 (MIPI STM)
MIPI STM Config Reg
DIC 2 Data Structure
DIC 2 Config Registers
Debug Unit 1 (TAP Controller)
TAP Controller Config Reg
TS Data Structure/ Config Registers

Figure 5-7 Debug Request to Debug-Unit Configuration Registers

5.4 Debug Control Requests

5.4.1 SET_CONFIG_DATA and GET_CONFIG_DATA Overview

Figure 5-8 shows an example of a SET_CONFIG_DATA, GET_CONFIG_DATA, and a SET_CONFIG_DATA_SINGLE command. These commands provide read and write access to the memory space.

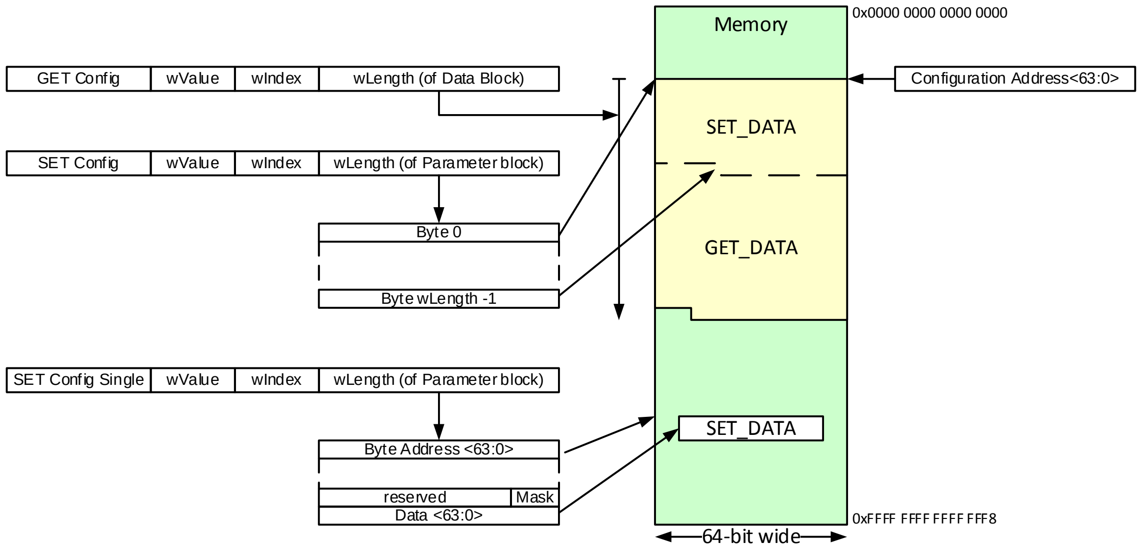


Figure 5-8: Example of SET_CONFIG_DATA, SET_CONFIG_DATA_SINGLE, and GET_CONFIG_DATA access of Memory space

The SET_CONFIG_DATA_SINGLE command writes a contiguous, byte-masked data value, from 0 to 8 bytes in size, to a 64-bit byte address. The address, data and byte mask are contained within the 32B Parameter block.

The SET_CONFIG_DATA command writes N contiguous data bytes to the 32-bit aligned, 64-bit byte address specified by the Configuration Address register. The SET_CONFIG_ADDRESS and the GET_CONFIG_ADDRESS commands read and write this register. The wLength field in the SET_CONFIG_DATA command defines the number of bytes to write.

The GET_CONFIG_DATA command reads N contiguous bytes starting at the 32-bit aligned, 64-bit byte address specified by the Configuration Address register. The wLength field in the GET_CONFIG_DATA command defines the number of bytes to read.

5.4.2 Debug Commands and Operating Modes

Debug commands are intended for configuration of the debug units at the start of a debug session and for “housekeeping” tasks during a debug session. They are not intended for general debug. Of course, the Get/Set Configuration data can read/write memory, but there are no debug commands for stop-mode or run-mode operations (e.g., Set breakpoint, Single-step, etc.). These should be done via DxC.Dfx or DxC.GP as appropriate.

The UBS device hardware controllers support either software or hardware decode of the debug control. The simplest implementation uses software decode, where the device hardware controller interrupts the USB stack, and software performs the operation. Hardware decode, on the other hand, performs the debug request in hardware without disturbing the OS. This is preferable for debug.

Note that a DbC is a standard USB device, in the sense that it supports a Default Control endpoint, which responds to standard USB requests, e.g. SET_ADDRESS, GET_DESCRIPTOR, GET_CONFIGURATION, etc. Typical DbC implementations provide hardware support for these commands and thus implementations may choose to extend this hardware to include the additional debug commands. From a debug perspective, this is the preferred implementation.

If a debug function does not support a certain request, it must indicate this by stalling the control pipe when that request is issued to the function.

5.4.3 SET_CONFIG_DATA_SINGLE

The SET_CONFIG_DATA_SINGLE command writes a contiguous, byte-masked data value, from 0 to 8 bytes in size, to a 64-bit byte address. The address, data and byte mask are contained within the 32B Parameter block.

Table 5-4: SET_CONFIG_DATA_SINGLE Control Request

Offset (Byte)	**Field**	**Size (Bytes)**	**Value**	**Description**

0	bmRequestType	1	0x21	D7 = 0 → Host to device
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x02	SET_CONFIG_DATA_SINGLE
2	wValue	2	Number	wValue<15:8>: *reserved* wValue<7:0>: See Table 5-5
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	2	Number	Parameter Block Length: • 32B for Debug Class specific parameter block • Variable-length for Vendor specific. Table 5-5 defines the selection between these 2 options

The wIndex and wValue fields define whether the access is to the Global, Local, or a Specific unit (see Table 5-5). In addition, these fields define whether the access is a 64-bit access using an architected 32B data parameter block, or if the data-parameter block is vendor specific.

Table 5-5: SET & GET_CONFIG_DATA Debug Data Structure Selection

wIndex<15:8>	wValue<7:0>	Debug Data Structure
UnitID = 0	0x00 (Debug Class defined Parameter Block) 0x01 (Vendor defined Parameter Block)	Global (i.e., the complete TS, for example the SoC)
	0x02 (Debug Class defined Parameter Block) 0x03 (Vendor defined Parameter Block) *Otherwise: reserved*	Local (i.e., DIC)
UnitID ≠ 0	0x00 (Debug Class defined Parameter Block) 0x01 (Vendor defined Parameter Block) *Otherwise: reserved*	Specific (i.e., Debug unit)

The Debug Class specific SET_CONFIG_DATA Parameter block (see Table 5-5) is 32 Bytes in size and consists of the following, little-endian format:

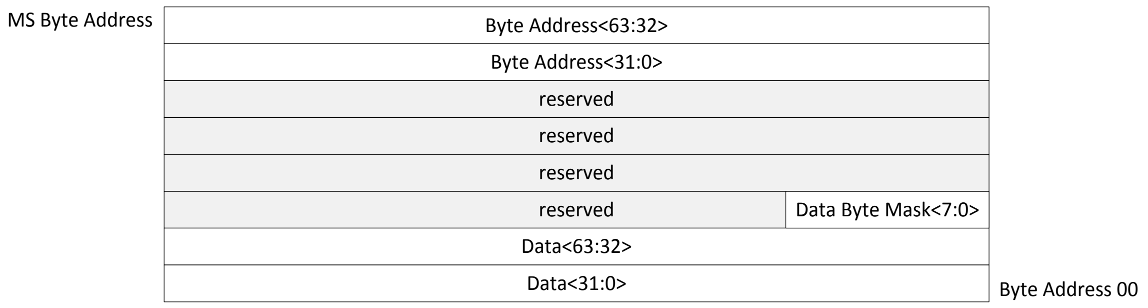


Figure 5-9: Debug Class Specific Parameter Block for SET_CONFIG_DATA Request

The address within the parameter block is a quad-word aligned 64-bit address and thus address<2:0> are ignored.

5.4.4 SET_CONFIG_DATA

The SET_CONFIG_DATA command writes N contiguous data bytes to the 32-bit aligned, 64-bit byte address specified by the Configuration Address register. The SET_CONFIG_ADDRESS and the GET_CONFIG_ADDRESS commands read and write this Configuration-Address register. The wLength field in the SET_CONFIG_DATA command defines the number of bytes to write.

Table 5-6 defines the SET_CONFIG_DATA command.

Table 5-6: SET_CONFIG_DATA Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0x21	D7 = 0 → Host to device
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x01	SET_CONFIG_DATA
2	wValue	2	Number	wValue<15:8>: *reserved* wValue<7:0>: See Table 5-5
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	2	Number	Parameter Block Length: Number of data bytes in the parameter block. The number of bytes equals wLength+1. Thus, the SET_CONFIG_DATA can write from 1 to 64KB.

The wIndex and wValue fields define whether the access is to the Global, Local, or a Specific unit (see Table 5-5).

5.4.5 GET_CONFIG_DATA

The GET_CONFIG_DATA command reads N contiguous bytes starting at the 32-bit aligned, 64-bit byte address specified by the Configuration Address register. The wLength field in the GET_CONFIG_DATA command defines the number of bytes to read, from 1 to wLength+1. The SET_CONFIG_ADDRESS and the GET_CONFIG_ADDRESS commands read and write this register. The data is in little-endian format.

Table 5-7: Get Config Data Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0xA1	D7 = 1 → Device to Host
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x81	GET_CONFIG_DATA
2	wValue	2	Number	wValue<15:8>: *reserved* wValue<7:0>: See Table 5-5
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	2	Number	Parameter Block Length. Size of the data to be fetched in bytes, from 1 byte up to a maximum, of 64KB (i.e., wLength +1)

5.4.6 SET_CONFIG_ADDRESS

This command writes to the 64-bit, Dword aligned byte Configuration Address register. The architecture defines a Configuration Address per Global, Local, or per Specific debug unit via the wIndex and wValue fields (see Table 5-5). However, an implementation may choose to alias these three memory spaces into a single Configuration Address registers, thus defining a single, unified address space.

If the TS does not support this command then the Configuration Address defaults to the value 0.

Table 5-8: SET_CONFIG_ADDRESS Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0x21	D7 = 0 → Host to device
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x03	SET_CONFIG_ADDRESS
2	wValue	2	Number	wValue<15:8>: *reserved* wValue<7:0>: See Table 5-5
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	2	0x0008	Parameter Block Length = 8 The parameter block contains the 64-bit, Dword aligned address to place in the Configuration Address register

5.4.7 GET_CONFIG_ADDRESS

This command reads from the 64-bit Configuration Address register.

Table 5-9: GET_CONFIG_ADDRESS Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0xA1	D7 = 1 → Device to device
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x83	GET_CONFIG_ADDRESS
2	wValue	2	Number	wValue<15:8>: *reserved* wValue<7:0>: See Table 5-5
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	2	0x0008	Parameter Block Length = 8 bytes Read all 8 bytes of the 64-bit, Dword-aligned address in the Configuration Address register.

5.4.8 SET_ ALT_STACK

Typical SoC's contain many cores, where one or more may support the USB stack. For example, during initial boot, a secondary core may provide basic USB device support for downloading new firmware or Operating System images. Later, the main OS takes over and provides the USB stack after it has booted. It is advantageous to allow the DTS to switch between these various USB stacks as necessary. For example, if the main OS hangs, then switching the USB stack to a secondary core will allow debug to continue.

The SET_ALT_STACK command is optional, and selects between the various options defined by the GET_ALT_STACK vendor specific. See the GET_ALT_STACK section.

Table 5-10: Set_ Alt_Stack Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0x21	D7 = 0 → Host to device
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x04	SET_ALT_STACK
2	wValue	2	0x0000	*Not used (This command only applies to the complete TS and not to a DIC or debug unit)*
4	wIndex	2	Number	This numbers selects one of the options supported by the GET_ALT_STACK
6	wLength	2	0x0000	No Parameter Block

5.4.9 GET_ALT_STACK

This command returns information on the state of the various “cores” within the SoC that support a USB stack. Because SoC designs vary considerably, then this capability is vendor specific. However, we provide an example usage as a guide.

Table 5-11: GET_ALT_STACK Control Request

Offset (Byte)	**Field**	**Size (Bytes)**	**Value**	**Description**

0	bmRequestType	1	0xA1	D7 = 1 → Device to Host
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x84	GET_ALT_STACK
2	wValue	2	0	*Not used (This command only applies to the complete TS and not to a DIC or debug unit)*
4	wIndex	2	0	*Not used*
6	wLength	2	Number	Parameter Block from 1 byte up to a maximum, of 64KB (i.e., wLength +1). This is vendor specific, but see Table 5-12 for an example.

The Debug Class specific GET_ALT_STACK Parameter block is a 2-Byte bitmask:

Table 5-12: Example GET_ALT_STACK Parameter Block

Bit	Description
<0>	Running on USB Stack 1
<1>	USB Stack 1 is available
<2>	Running on USB Stack 2
<3>	USB Stack 2 is available
<4>	Running on USB Stack 3
<5>	USB Stack 3 is available
<6>	Running on USB Stack 4
<7>	USB Stack 4 is available

Table 5-12 is for an implementation that supports four alternate USB stacks, labelled 1 to 3. The even bits <0, 2, 4, and 6> are mutually exclusive and indicate which Stack is active. For example, the main core could be USB Stack 0, while the boot core could be USB Stack 1. Clearly, only 1 USB stack can be running at a time.

The odd bits <1, 3, 5, & 7> indicate if the associated core that can run the USB stack is available. For example, it has booted and is not in a powered-down, inactive state.

Table 5-13 defines the corresponding values for wIndex in the SET_ALT_STACK. Each pair of consecutive bits corresponds to the different USB stacks. The DTS selects which USB Stack it wishes to use next by setting one of the even bits <0, 2, etc.> The DTS host then performs a USB reset, forcing the TS to use the new USB Stack for enumeration. Consequently, the TS must preserve the information on which USB Stack to use when a USB Reset occurs.

The odd bits of Table 5-13 allow the DTS to tell a core it can go back to sleep if it wishes to – in other words, the DTS no longer need the core to be active. For example, an SoC may go into a sleep state where the USB hardware logic is alive but the rest of the chip is in a low-power state. In this case, the "USB Stack N is available" bit of the active USB Stack of Table 5-12 will be 0 indicating that the core is asleep. The DTS this asserts the corresponding even bit of Table 5-13 when it needs to wake up the core in order to perform a debug operation; and then later sets the odd bit, when it has finished with the debug operation.

Table 5-13 Example SET_ALT_STACK wIndex

Bit	Description
<0>	Use USB stack 1 for the debug capabilities
<1>	Allow USB Stack 1 to go to sleep if required.
<2>	Use USB stack 2 for the debug capabilities
<3>	Allow USB Stack 2 to go to sleep if required.
<4>	Use USB stack 3 for the debug capabilities
<5>	Allow USB Stack 3 to go to sleep if required.
<6>	Use USB stack 4 for the debug capabilities
<7>	Allow USB Stack 5 to go to sleep if required.

5.4.10 SET_OPERATING_MODE

The SET_OPERATING_MODE is used to control the voltage, clocks, initialization, etc. of the TS, DIC, or Debug Unit. This command thus defines whether the debug logic is available for use or not. The SET_OPERATING_MODE selects a specified debug operating mode, and then the software or hardware in the TS, DIC, or Debug Unit places the debug hardware in the desired mode.

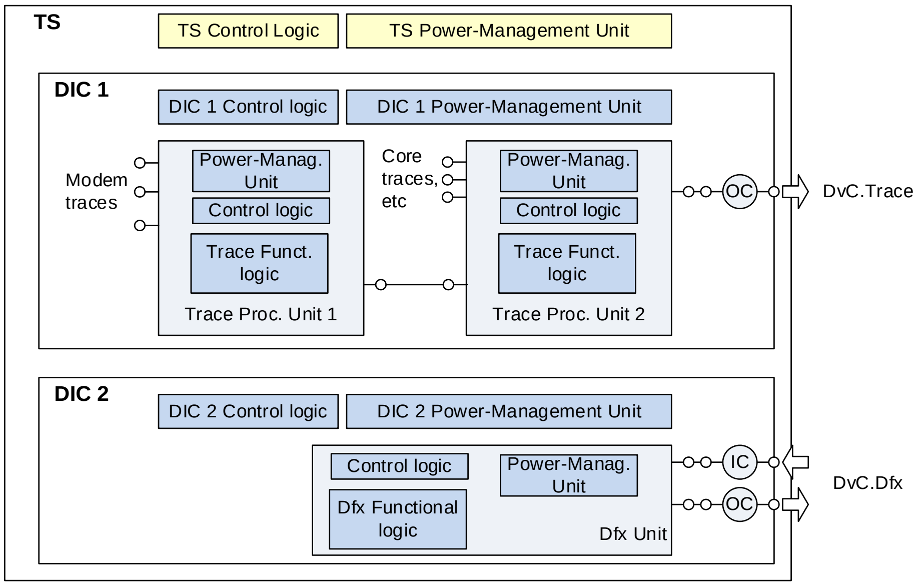


Figure 5-10: SET_OPERATING_MODE Example

Figure 5-10 shows an example of a TS consisting of two DICs: one for the DvC.Trace capability and one for the DvC.Dfx capability. In this example, each of the TS, DICs, and Debug units have an associated Power-Management unit, allowing fine-grain control of which unit(s) are powered-on. An actual implementation may provide less capability, but this example shows extensive capability for the purpose of illustration.

Note that each entity (TS, DIC or Debug unit) can allow power-management of the debug control logic (and any associated configuration registers) as well as of the “main” debug functional hardware (e.g., the trace-merge hardware). Thus, for example, one can power down the debug control capability while keeping the trace hardware powered on.

Consider, for example, the scenario where the DTS is only capturing traces from the core in Figure 5-10, and the DTS wishes to power down all the remaining debug logic to avoid excessive battery drain of the TS. In this case, the DTS will do the following:

- Disable the power being applied to the Debug-command logic for the TS, DIC1, DIC2, Trace-Processing unit 1, Trace-Processing unit 2, and the Dfx Unit. This assumes that the debug control logic and associated configuration registers are in a separate sub-power well that is distinct from the trace and Dfx logic in the debug units. Admittedly, this is unlikely in current-generation parts, but is being considered here for the purpose of illustration.
- Disable the Trace Processing unit 1 hardware associated with the Modem traces, since the DTS is not interested in these traces.
- Disable the Dfx unit hardware logic since the DTS is only interested in core traces and is thus not using the Dfx interface.

The SET_OPERATING_MODE (Debug-Operating mode) request can target the TS Control Interface, or the Control/capability interfaces within a DIC. Thus, for example, the SET_OPERATING_MODE (Debug-Operating mode) can power off the Debug-Control logic associated with the Debug trace but still keep the debug trace hardware powered-on.

At the other extreme, a TS may need to turn on/off all of the debug logic, and it would be inconvenient to have to individually enable/disable each DIC and Debug Unit. For this reason, the SET_OPERATING_MODE supports a Debug-All mode, which turns all of the debug logic on or off.

The SET_OPERATING_MODE request configures the operating mode in the Global (TS), Local (DIC), or Specific debug unit as specified by the wIndex and wValue fields (see Table 5-16). Thus, if we are only interested in capturing traces from a specific unit (e.g., the modem) then this command allows us to enable the power and clocks in the specific unit or DIC that contains the relevant logic. An implementation may choose to not support this level of granularity. For example, an implementation may choose to enable the power for all of the debug logic in a TS, even though the SET_OPERATING_MODE command targeted a particular DIC within the TS.

A device may choose to not provide the SET_OPERATING_MODE control. In this case, it shall default to the equivalent of the Debug-All-mode = ON for all of the supported debug logic within the TS.

The operating modes are defined in the Table 5-14:

Table 5-14: SET_OPERATING_MODE Definitions

Mode	Description
Debug All	0: Debug-All mode = OFF 1: Debug-All mode = ON Certain SoC will only provide the capability to enable/disable all of the debug hardware (i.e., Traces, Dfx, Control, etc.) and nothing less. In this case, the SET_OPERATING_MODE (Debug-Logic mode = ON/OFF) enables/disables all of the debug logic within the TS. Note that the other modes allow fine-grain enabling/disabling of the power to individual debug components. An SoC may choose to use the Debug-All = ON mode similarly to a UART Wakeup. In this case, if the SoC has gone to sleep, then it will awaken when the DTS issues a Debug-All mode = ON. If the TS supports preservation of debug state across a wakeup, then it set the Graceful-Wakeup-Supported bit in the parameter block of Table 5-17. This mode is optional.

Mode	Description
Debug Operating	0: Debug-Operating mode = OFF 1: Debug-Operating mode = ON The SET_OPERATING_MODE (Debug-Operating) request can target the TS Control Interface, or the Control or Capability interfaces within a DIC. In this way, this request can power on/off the Trace, Dfx, or GP hardware capability or the Debug Control logic associated with the TS, DIC or debug unit. The DTS uses the SET_OPERATING_MODE (Debug-Operating = ON) command to place the Command or Functional component of the TS, DIC or Debug Unit into a fully functional mode with regards the supported Debug capability. Thus, to the DTS, the specified debug command or capability appears to be fully functional even though portions of the SoC (including the debug logic) could be asleep. For example, when the Debug-Operating Mode = ON, a TS may keep the debug logic always powered up; or it may choose to power-down the debug logic, but awaken it automatically and transparently whenever it receives a command from the DTS. In either case, the DTS will be under the impression that the debug commands or capability associated with the TS, DIC, or Debug unit is seamlessly, always available. Debug-Operating Mode = OFF means that the targeted hardware is powered down, and thus not operational. However, in this mode, the SET_OPERATING_MODE command should provide support for the DTS to set Debug-Operating mode = ON, thus allowing subsequent re-enabling of the debug operations. If the TS does not provide this support, then it will STALL the request. This mode is optional.
Debug Emulate Low-Power	This command affects the debug logic within the complete TS, within the DIC, or within a specific Debug unit as defined by Table 5-16. In this mode, the device emulates the steps it takes to transition between low-power states, but does not actually power down the hardware. Thus, the device goes through the motions, but does not actually change the power state. This avoids the TS having to save debug state that would be lost during an actual power transition. This mode is optional.
Graceful Degrade	In this mode, the TS, DIC or Debug Unit are allowed to autonomously disable functionality. For example, a DIC may stop generating traces when the device’s battery power becomes too low. The TS, DIC, or Debug unit should issue an interrupt to notify the DTS whenever it disables any debug functionality. This mode is optional.

Mode	Description
Force Debug	There are situations where the DTS needs to inform the TS not to disable the debug logic. For example, enabling all of the debug logic in a SoC may place it outside the safe operating mode of the device. A lab debugger may require the device to be in this potentially damaging state, because this may be the only way to debug a sighting. In addition, forcing a debug unit to be on, prevents the power-management logic from powering it off when in the Graceful-Degrade mode (e.g. when the battery is running low). Presumably, the power-management logic will instead power down some other logic in this scenario. This mode is optional, and will probably be fused off in a production part.
Close Debug	At the end of a debug session, the DTS may need to send a command to the TS, DIC or Debug Unit to gracefully disable the debug logic. Otherwise, for example, a trace buffer may overflow and hang the SoC if it is not gracefully disconnected prior to a USB3 link disconnect.

Table 5-15 defines the SET_OPERATING_MODE Control request.

Table 5-15: SET_OPERATING_MODE Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0x21	D7 = 0 → Host to device
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x05	SET_OPERATING_MODE
2	wValue	2	Number	wValue<15:8>: *reserved* wValue<7:0>: See Table 5-16
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	2	0x0004	*Device Operating Mode* Parameter Block. 4–byte Bitmap. See Table 5-17

Table 5-16: SET & GET OPERATING MODE Debug Structure

wIndex<15:8>	wValue<7:0>	Debug Structure
UnitID = 0	0x00	Global (i.e., TS)
	0x02 *Otherwise: reserved*	Local (i.e., DIC)
UnitID ≠ 0	0x00 *Otherwise: reserved*	Specific (i.e., Debug unit)

The Debug Class specific SET OPERATING Parameter block is a 2-Byte bitmask Table 5-17.

Table 5-17: Device Operating-Mode Parameter Block

Bit	Description	Read/Write
<0>	Debug-All mode	RW
<1>	Debug-All supported	R
<2>	Graceful-Wakeup-Supported	R
<3>	*reserved*	
<4>	Debug-Operating mode	RW
<5>	Debug-Operating mode supported	R
<6>	Debug Emulate low-power	RW
<7>	Debug Emulate low-power mode supported	R
<8>	Graceful-Degrade	RW
<9>	Graceful-Degrade mode supported	R
<10>	Force-Debug	RW
<11>	Force-Debug mode supported	R
<12>	Device uses power supplied by USB.	RW-opt
<13>	Device uses power supplied by Battery	RW-opt
<14>	Device uses power supplied by AC	RW-opt
<15>	Close Debug	RW
<16>	Close Debug mode supported	R
<31:17>	Vendor-specific operating modes	RW-opt

The bit fields in the parameter block provides Information regarding operating modes and power sources:

- D16, D11, D9, D7, D5, D2, D1 define whether the TS, DIC, or the Debug unit supports the specified operation mode
- D15, D10, D8, D6, D4, D0 enable/disabled the specified operation mode in the TS, DIC, or Debug unit. These bits are mutually exclusive. The behavior if multiple bits are enabled is undefined
- D14..D12 indicates which power source is currently used in the USB device. These bits are mutually exclusive. The behavior if multiple bits are enabled is undefined.

  The bits D14..D12 are set by the device and are usually informational. However, it is recommended that the TS allows the debugger to force the device to use a different power source, which is why these bits are designated as RW-opt (Read & Write-optional). For example, when debugging over USB, the device will normally be charging over the USB cable instead of using its internal battery. However, during a battery-consumption test, one needs to force the device to use its internal battery.
- D31..D17 are vendor-specific operating modes

5.4.11 GET_OPERATING_MODE

This Control request reads the parameter block in the specified debug unit.

Table 5-18: GET_OPERATING_MODE Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0xA1	D7 = 1 → Device to Host
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x85	GET_OPERATING_MODE
2	wValue	2	Number	wValue<15:8>: *reserved* wValue<7:0>: See Table 5-16
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	2	0x0004	*Device Power Mode* Parameter Block. 4–byte Bitmap. See Table 5-17

5.4.12 GET_INFO

The GET_INFO request queries the capabilities and status of the specified control. This command is mandatory if a device supports at least one Debug Command.

Table 5-19: GET_INFO Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0xA1	D7 = 1 → Device to Host
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x87	GET_INFO
2	wValue	2	Number	wValue<15:8>: *reserved* wValue<7:0>: See Table 5-16
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	4	0x0004	GET_INFO Parameter Block. See Table 5-17

When issuing the GET_INFO request, the wLength field shall always be set to a value of 1 byte. The result returned is a bit mask reporting the capabilities of the control. The bits are defined as:

Table 5-20: GET_INFO Parameter Block

Bit Field	Description	Bit state
<0>	1 = Supports SET_CONFIG_DATA_SINGLE	Capability
<1>	1 = Supports Set_Config_Data	Capability
<2>	1 = GET_CONFIG_DATA	Capability
<3>	1 = SET_CONFIG_ADDRESS	Capability
<4>	1 = GET_CONFIG_ADDRESS	Capability

Bit Field	Description	Bit state
<5>	1 = SET_ALT_STACK	Capability
<6>	1 = GET ALT STACK	Capability
<7>	1 = SET OPERATING MODE	Capability
<8>	1 = GET OPERATING MODE	Capability
<9>	1 = SET TRACE CONFIGURATION	Capability
<10>	1 = GET TRACE CONFIGURATION	Capability
<11>	1 = SET BUFFER	Capability
<12>	1 = GET BUFFER	Capability
<13>	1 = SET RESET	Capability
<28:14>	*reserved (Set to 0)*	--
<29>	1=Disabled due to automatic mode (under device control)	State
<30>	1= Self-Generated Control (see Section 3.6.7.2, "Status Interrupt Endpoint")	Capability
<31>	1= Slow Control (see Section 3.6.7.2, "Status Interrupt Endpoint")	Capability

5.4.13 GET_ERROR

This read-only control indicates the status of each host-initiated request to a Unit or interface of the debug function. If the device is unable to fulfill the request, it will indicate a Stall on the control pipe and update this control with the appropriate code to indicate the cause. This control will be reset to 0 (i.e., No Error) upon the successful completion of any control request (including requests to this control). Slow control requests are a special case, where the initial request will update this control, but the final result is delivered via the Status Interrupt Endpoint (see sections 3.6.7.2, "Status Interrupt Endpoint"). This command is mandatory if a device supports at least one Debug Command.

Table 5-21: GET_ERROR Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0xA1	D7 = 1 → Device to Host
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x88	GET_ERROR
2	wValue	2	Number	wValue<15:8>: *reserved* wValue<7:0>: See Table 5-16
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	2	0x1	ERROR CODE Parameter Block. See Table 5-22

Table 5-22: Error Code Parameter Block

Offset (Byte)	Field	Size (Bytes)	Value	Description
0	bRequestErrorCode	1	Number	0x00: No error 0x01: Not ready 0x02: Wrong state 0x03: Insufficient Power Available 0x04: Max Power Violation 0x05: Operating-mode unavailable 0x06: Out of range 0x07: Invalid unit 0x08: Invalid control 0x09: Invalid Request 0x0A: Permission denied 0x0B-0xFE: *reserved* 0xFF: Unknown

The bit field in the parameter block provides status information regarding error conditions:

- No error: The request succeeded.
- Not ready: The device has not completed a previous operation. The device will recover from this state as soon as the previous operation has completed.
- Wrong State: The device is in a state that disallows the specific request. The device will remain in this state until a specific action from the host or the user is completed.
- Insufficient Power available: The actual Operating Mode of the device is insufficient to complete the Request. For example:
    - The TS may be able to support a request when AC powered but not when battery powered (or when the battery is almost dead).
    - The TS is in an operating mode where the Trace DIC is powered-off thus preventing access to the configuration register via the SET_CONFIG_DATA command. The debugger may need to first change the operating mode and then resubmit the request.
- Max Power Violation: The requested debug operation would exceed the maximum operating conditions of the TS and was denied
- Operating-mode unavailable: The operating mode requested is not available
- Out of Range: Result of a SET_CONFIG_DATA Request when attempting to write to a non-existent/out-of-range address.
- Invalid Unit: The Unit ID addressed in this Request is not assigned.
- Invalid Control: The Control addressed by this Request is not supported.
- Permission Denied: The request was denied by the security state.
- Invalid Request: This Request is not supported by the Control.

A request to an unauthenticated debug unit, DIC or TS stalls the command and reports an *Invalid Control* or *Permission Denied* depending on the implementation.

5.4.14 SET_TRACE

The SET_TRACE request enables one of 255 trace configurations. SET_TRACE (0) disables traces. The actual traces enabled is vendor specific and beyond the scope of this document.

This requests sets or reads the vendor-specific trace configuration. The vendor can define one of 255 possible trace configurations. For example, Trace Configuration 1 may enable all traces within the TS, while Trace configuration 2 only enables the modem traces. This register is not a bit mask but a number corresponding to a set of allowed traces.

Enabling debug power mode may automatically start a vendor-defined trace configuration. The debugger can use the GET_TRACE_CONFIGURATION to determine which configuration is enabled.

Table 5-23: SET_TRACE Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0x21	D7 = 0 → Device to Host
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x06	SET_TRACE
2	wValue	2	0x0000	*Not used (This command only applies to the complete TS and not to a DIC or debug unit)*
4	wIndex	2	Number	Trace Configuration: 0: Disable Trace 1-255: Vendor-specific trace configuration *Otherwise: reserved*
6	wLength	2	0x0000	No Parameter Block

5.4.15 GET_TRACE

The GET_TRACE returns a number indicating which trace configuration is currently active.

Table 5-24: GET_TRACE Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0xA1	D7 = 1 → Host to device
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x86	GET_TRACE
2	wValue	2	0x0000	*Not used (This command only applies to the complete TS and not to a DIC or debug unit)*

Offset (Byte)	Field	Size (Bytes)	Value	Description
4	wIndex	2	Number	Trace Configuration: 0: There are no Trace configurations 1-255: Vendor-specific trace configuration *Otherwise: reserved*
6	wLength	2	0x2	Returns 2-byte Parameter Block containing the number of the active Trace Configuration

5.4.16 SET_BUFFER

The Set Buffer command performs basic operations on buffer(s) within a TS, DIC, or Debug Unit. Table 5-25 defines the command.

Table 5-25: SET_BUFFER Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0x21	D7 = 0 → Host to device
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x09	SET_BUFFER
2	wValue	2	Number	wValue<15:8>: *reserved* wValue<7:0>: See Table 5-16
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	8	0x0008	*Buffer* Parameter Block. 8–byte structure. See Table 5-26.

Table 5-26 defines the parameter block for the SET_BUFFER_INFO command.

Table 5-26: Buffer Parameter Block

Bit	Description
<7:0>	Command: 0: Flush Buffer 1: Initialize Buffer 3: Set Buffer Size (size in bits <31:16>) Otherwise: *reserved*
<15:8>	Vendor-Specific Buffer Modes
<31:16>	Buffer Size in bytes
<64:32>	*reserved*

5.4.17 GET_BUFFER

The GET_BUFFER command reads the buffer size corresponding to a TS, DIC, or Debug Unit. Table 5-25 defines the command.

Table 5-27: GET_BUFFER Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0xA1	D7 = 1 → Host to device
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x89	GET_BUFFER
2	wValue	2	Number	wValue<15:8>: Bit Map <0>: Buffer was Flushed <1>: Buffer was Initialized <2>: Get Buffer Size wValue<7:0>: See Table 5-16
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	2	0x0004	4-byte Parameter Block. Only applicable for wValue<15:8> = Get_Buffer_Size <31:0>: Buffer Size

5.4.18 SET_RESET

The SET_RESET request resets and initializes the TS, DIC, or Debug unit logic as specified by the wValue field. This class-specific request shall return the function to its default, initialized state with all buffers cleared, and the configuration registers in their default state. The DTS can use this request to resume debug functionality when the TS, DIC, or debug unit has unexpectedly stopped functioning.

Table 5-28: SET_RESET Control Request

Offset (Byte)	Field	Size (Bytes)	Value	Description

0	bmRequestType	1	0x21	D7 = 0 → Device to Host
				D6..5 = 01 → Class request
				D4..0 = 00001 → Recipient is interface

1	bRequest	1	0x0A	SET_RESET
2	wValue	2	Number	wValue<15:8>: *reserved* wValue<7:0>: See Table 5-16
4	wIndex	2	Number	wIndex<15:8> = Debug Unit ID wIndex <7:0> = Interface ID
6	wLength	2	0x0000	No Parameter Block

A reset recovery consists of performing the following steps in order:

1. Set Reset
2. Clear Feature (ENDPOINT_HALT) standard USB request for the IN endpoint for the debug interface that has stopped functioning.
3. Clear Feature (ENDPOINT_HALT) standard USB request for the OUT endpoint for the debug interface that has stopped functioning

6 Debug Payload

6.1 Debug Trace Overview

The Debug Class driver simply passes the data payload for all three capabilities (DxC.Trace, DxC.Dfx, DxC.GP) up to the software stack, and is oblivious of their content. These are vendor-specific or maybe defined by another standards body. However, Appendix C: provides information on a suggested debug-trace format.

7 USB 3.1 Debug Security

7.1 Overview

The Debug Class specification does not address security issues. The requirement is that the TS will provide implementation-specific security features on the USB 3.1 debug hooks. The USB 3.1 interface is essentially a “virtual” debug port allowing access to the debug hooks within the device. Any security features that the device uses to protect access via physical debug ports (e.g., JTAG) are equally applicable to the debug access via the USB “virtual” debug port.

Note that it is possible to conceal debug features: For example, via alternate interfaces as described in section 3.6.1.

8 USB 3.1 Debug Data Structures

8.1 Overview

The Debug Class specification has defined a number of mechanisms to access debug data structures within the TS, within a DIC, or within an actual debug unit. These mechanisms are:

- GET_CONFIG_DATA and SET_CONFIG_DATA commands (see Section 5 for more details)
- Data structures contained within the Debug-Attributes and Debug-Unit descriptors (see Sections 4.4.3 and 4.4.6)

These data structures are vendor specific and do not necessarily point to the same data structure. An implementation may choose to alias the data structures contained within the descriptors to those accessed via GET_CONFIG_DATA and SET_CONFIG_DATA; but this is not required.

There is no BOS descriptor defined for debug data.

Appendix A: Debug-Device-Class Codes

Figure 4-4 shows the information in the following tables as a diagram.

Table 8-1: Debug Interface Class Code (CC_DEBUG)

Debug Interface Class Code	Value
CC_DEBUG	0xDC

Table 8-2: Debug Interface Sub-Class Code (SC_DEBUG)

Debug Interface Sub-Class Code (SC_DEBUG)	Value
SC_DbC	0x02
SC_DbC_DFX	0x03
SC_DbC_TRACE	0x04
SC_DvC_GP	0x05
SC_DvC_DFX	0x06
SC_DvC_TRACE	0x07
SC_DEBUG_CONTROL	0x08

Table 8-3: Debug Interface Protocol Code (PC_DEBUG)

SC_DbC Interface Protocol Code	Value
Protocol code (see xHCI specification)	DCDDI1 DbC Protocol Field
SC_DbC_Dfx Interface Protocol Code	Value
PC_PROTOCOL_CODE_UNDEFINED	0x00
PC_PROTOCOL_TARGET_VENDOR	0x01
SC_DbC_Trace Interface Protocol Code	Value
PC_PROTOCOL_CODE_UNDEFINED	0x00
PC_PROTOCOL_TARGET_VENDOR	0x01
SC_DvC_GP Interface Protocol Code	Value
PC_PROTOCOL_TARGET_VENDOR	0x00
PC_PROTOCOL_GNU	0x01
SC_DvC_Dfx Interface Protocol Code	Value
PC_PROTOCOL_CODE_UNDEFINED	0x00
PC_PROTOCOL_TARGET_VENDOR	0x01
SC_DvC_Trace Interface Protocol Code	Value
PC_PROTOCOL_CODE_UNDEFINED	0x00
PC_PROTOCOL_TARGET_VENDOR	0x01

Debug Class-Specific Descriptor Types

Table 8-4: Debug Class-Specific Descriptor Types

Debug Class-Specific Descriptor Type	Value
SC_DvC_Trace Interface Protocol Code	Value
CS_UNDEFINED	0x20
CS_DEVICE	0x21
CS_CONFIGURATION	0x22
CS_STRING	0x23
CS_INTERFACE	0x24
CS_ENDPOINT	0x25

Table 8-5: Debug Class-Specific Commands bRequest

Debug Class-Specific Commands (bRequest)	Value
SET_CONFIG_DATA	0x01
SET_CONFIG_DATA_SINGLE	0x02
SET_CONFIG_ADDRESS	0x03
SET_ ALT_STACK	0x04
SET_OPERATING_MODE	0x05
SET_TTRACE	0x06
reserved	0x07, 0x08
SET_BUFFER	0x09
SET_RESET	0x0A
GET_CONFIG_DATA	0x81
GET_CONFIG_DATA_SINGLE	0x82
GET_CONFIG_ADDRESS	0x83
GET_ALT_STACK	0x84
GET_OPERATING_MODE	0x85
GET_TRACE	0x86
GET_INFO	0x87
GET_ERROR	0x88
GET_BUFFER	0x89

Debug Class-Specific Descriptor Sub-Types

Table 8-6: Debug Class-Specific Descriptor SubTypes

Debug Class-Specific Descriptor SubType	Value
DC_UNDEFINED	0x00
DC_INPUT_CONNECTION	0x01
DC_OUTPUT_CONNECTION	0x02
DC_DEBUG_UNIT	0x03
DC_DEBUG _ATTRIBUTES	0x04

Appendix B: Descriptor Examples

Overview

This Appendix gives a few examples of USB 3.1 debug descriptors for common scenarios. The examples in Figure 8-1 show corresponding descriptors aligned horizontally, to help highlight the differences and similarities between the various examples.

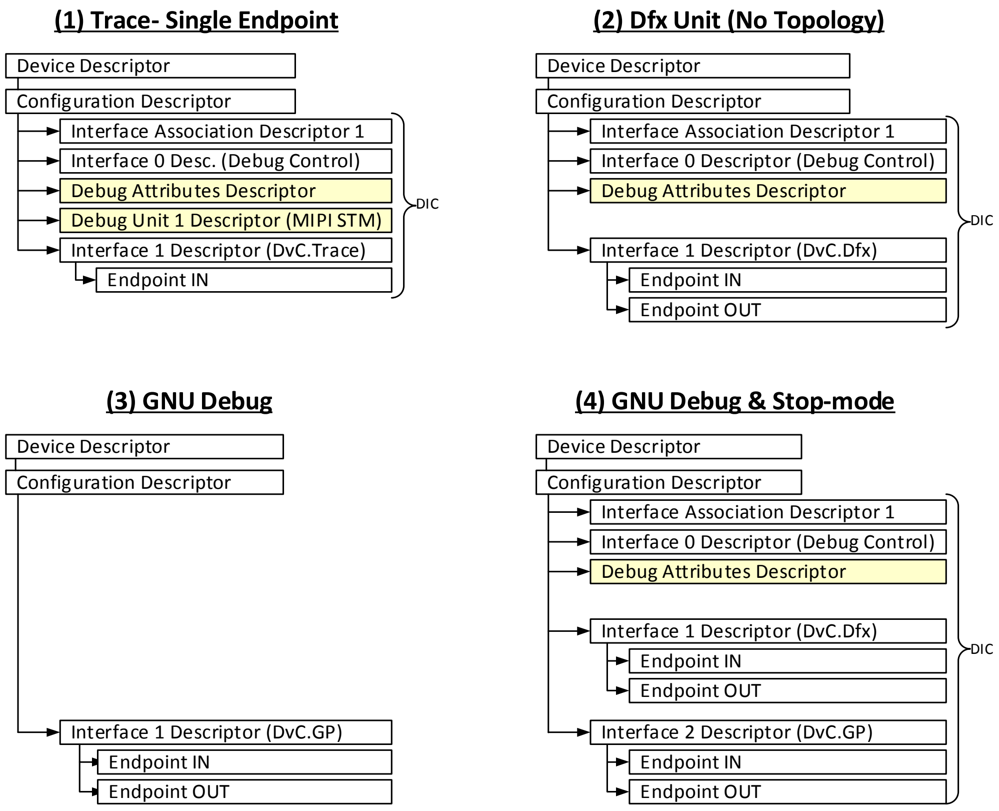

(1) Trace- Single Endpoint
Device Descriptor
Configuration Descriptor
Interface Association Descriptor 1
Interface 0 Desc. (Debug Control)
Debug Attributes Descriptor
Debug Unit 1 Descriptor (MIPI STM)
Interface 1 Descriptor (DvC.Trace)
Endpoint IN
DIC
(2) Dfx Unit (No Topology)
Device Descriptor
Configuration Descriptor
Interface Association Descriptor 1
Interface 0 Descriptor (Debug Control)
Debug Attributes Descriptor
Interface 1 Descriptor (DvC.Dfx)
Endpoint IN
Endpoint OUT
DIC
(3) GNU Debug
Device Descriptor
Configuration Descriptor
Interface 1 Descriptor (DvC.GP)
Endpoint IN
Endpoint OUT
(4) GNU Debug & Stop-mode
Device Descriptor
Configuration Descriptor
Interface Association Descriptor 1
Interface 0 Descriptor (Debug Control)
Debug Attributes Descriptor
Interface 1 Descriptor (DvC.Dfx)
Endpoint IN
Endpoint OUT
Interface 2 Descriptor (DvC.GP)
Endpoint IN
Endpoint OUT
DIC

Figure 8-1: USB 3.1 Debug Class Descriptor Examples

Example (1) in Figure 8-1 is for a low-cost system with only one bulk endpoint available for trace. The debugger uses the default endpoint 0 for the configuration and enabling of the traces. An Interface Association descriptor is required to “group” the control and DvC.Trace interfaces into a single DIC, if the control is associated with the Trace unit. Otherwise, see Figure 8-4 and associated text.

Example (2) in Figure 8-1 is for an interface to a standard Dfx unit such as the TAP. This example shows a Debug-Control Interface associated with the Dfx interface, and thus requires an IAD to form a DIC.

Example (3) in Figure 8-1 is for an interface to a kernel debugger using the DvC.GP capability. This is the similar to Example (2) except that there is no Debug-Attributes and Debug Control descriptors, and thus no IAD/DIC.

Example (4) in Figure 8-1 is one possible combination of examples (2) and (3). In this example, the Interface Association descriptor spans the DvC.Dfx and the DvC.GP capabilities creating a single DIC. If alternatively, we wished for separate DICs for each capability, then we would require an additional IAD together with the two debug control descriptors for each debug capability, as shown in Figure 8-2. The intention behind Example (4), is a debug tool that is primarily used for kernel debug via the GP interface, but if the kernel debugger hangs, then the debug tool can use TAP commands via the Dfx interface to debug the bug scenario.

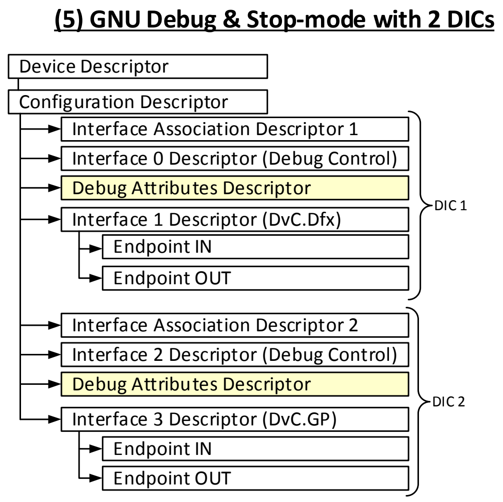

(5) GNU Debug & Stop-mode with 2 DICs
Device Descriptor
Configuration Descriptor
Interface Association Descriptor 1
Interface 0 Descriptor (Debug Control)
Debug Attributes Descriptor
Interface 1 Descriptor (DvC.Dfx)
Endpoint IN
Endpoint OUT
DIC 1
Interface Association Descriptor 2
Interface 2 Descriptor (Debug Control)
Debug Attributes Descriptor
Interface 3 Descriptor (DvC.GP)
Endpoint IN
Endpoint OUT
DIC 2

Figure 8-2: USB 3.1 Debug Class Descriptor example of two DICs

Figure 8-3 shows an implementation that only supports debug commands. There is no DIC in this case because there is only the singe Debug-Control Interface (i.e., an IAD requires 2 or more interfaces, which is why this example is not a DIC). An implementation that simply saves traces to an internal Sink (e.g., memory) does not requires a DxC.Trace interface, and can simply use the Debug-Control Interface to control the trace generation and then the extraction from the Sink.

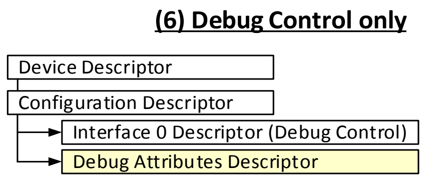

(6) Debug Control only
Device Descriptor
Configuration Descriptor
Interface 0 Descriptor (Debug Control)
Debug Attributes Descriptor

Figure 8-3: USB 3.1 Debug Class Descriptor example of Debug Control only

Figure 8-4 shows three debug interfaces – Debug Control, DvC.Trace and DvC.Dfx. There is NO IAD/DIC in this example because the Debug-Control Interface is for the TS and not for the Trace of Dfx interfaces. For example, the TS may only support the command to enable/disable all of the debug logic. Thus, the Debug Control is not associated specifically with the Trace or Dfx interfaces, and thus no IAD/DIC is required.

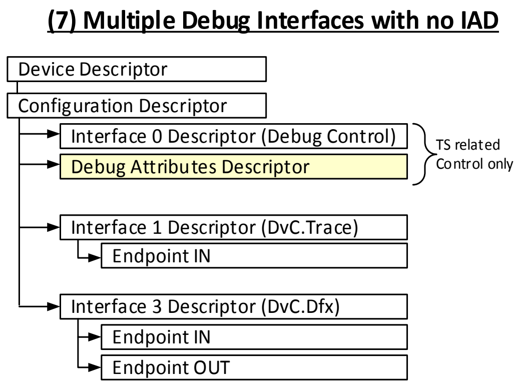

(7) Multiple Debug Interfaces with no IAD
Device Descriptor
Configuration Descriptor
Interface 0 Descriptor (Debug Control)
Debug Attributes Descriptor
TS related Control only
Interface 1 Descriptor (DvC.Trace)
Endpoint IN
Interface 3 Descriptor (DvC.Dfx)
Endpoint IN
Endpoint OUT

Figure 8-4: Example of Multiple Debug interface without a DIC

Appendix C: Debug Trace Payload Format

This Appendix describes three suggested debug trace formats. These formats apply to DxC.Trace and DxC.Dfx. The USB 3.1 Debug Class is oblivious of the payload format and simply pipes the payload upstream. However, debug does not always have to be robust – for example, receiving a corrupted trace very infrequently could be acceptable. This allows debug to “cheat” and violate rules. This Appendix describes a trace format that is tolerant of such cheating, including avoiding some USB specific requirements.

This Appendix provides specific details of a payload structure suitable for SuperSpeed and HighSpeed, and is tolerant of dropped packets under bulk retries.

Debug Trace Payload

A debug trace refers to a stream of debug data of an arbitrary byte length encapsulated in an integer multiple of 1KB segments for SuperSpeed and optionally 512B or 1KB segments for HighSpeed. Each 1KB/512B segment of the debug trace contains either a header at the start of the payload, or a Footer at the end of the payload. The figure below shows an example using a Footer.

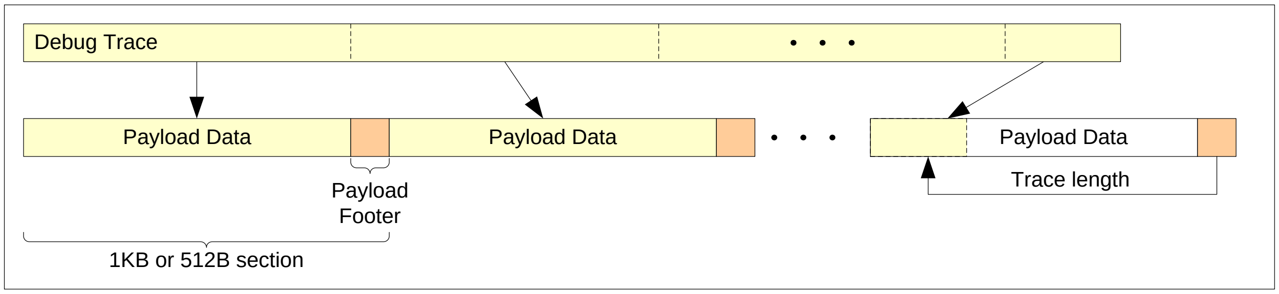


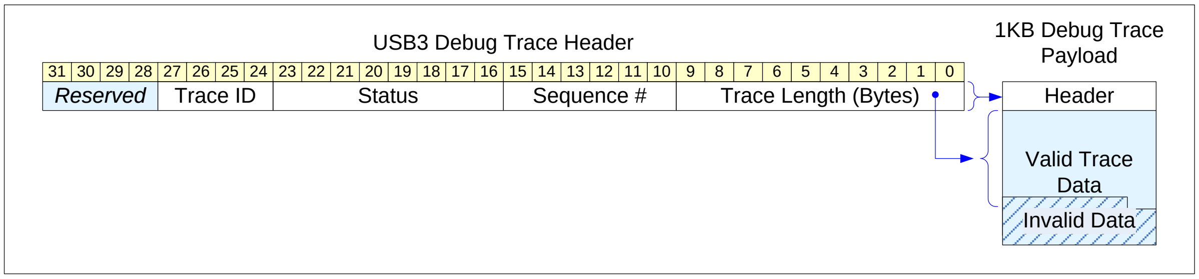
Figure 8-5: Example of a Debug Trace

Debug Trace Payload Size

USB 2.0 HighSpeed allows a max-packet-size of 512B for bulk transfers and 1KB for isochronous transfers. SuperSpeed allows 1KB for both isochronous and bulk transfers. Depending on the hardware implementation, the TS may send the trace as either 512B or 1KB segments in either HighSpeed or SuperSpeed. In other words, a TS may send a header/footer per 1KB data payload, even in HighSpeed.

Debug Trace Header / Footer

A Figure 8-6 shows two formats, one with a header and one with a footer. The header and footer are identical and are each 4 bytes in size. Table 4-11 “dTraceFormat” field of the Debug Unit descriptor of the Input Connector descriptor selects between these two trace formats.



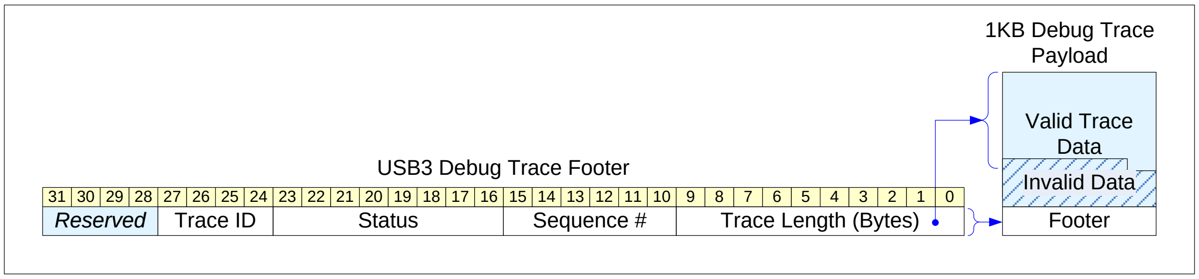


Figure 8-6: Debug Trace Footer Formats

Table 8-7 defines the fields in the trace header/footer.

Table 8-7: Header Fields

Field	Size	Description
Trace Length	9:0	Length of Debug trace in bytes for the data portion of a 1KB/512B region. Typically, this equals 1020/508 bytes when the data portion of the 1KB/512B region is full of trace data. However, at the end of a debug session, the data portion of the 1KB/512B region could be partially filled. The size of the data portion (1KB/512B) is defined in the Status & Information field of this footer.
Debug Sequence Number	5:0	The TS increments the sequence number whenever it completes writing to a 1KB region of the debug trace buffer. This field wraps. NB: This is not the same as the USB 3.1 Sequence number.
Status	7:0	The definition of the field <7:0> bits are: <0>: No trace data (i.e., “”NULL” packet) <1>: Bulk Retry occurred (optional, informational for the debugger) <2>: Backpressure occurred (optional, informational for the debugger) <3>: All Trace Data Flushed out of TS <4:5>: *reserved* <6>: Size of the Trace Payload 0: 1KB 1: 512B <7>: *reserved*
Trace ID	3:0	0x0 – 0x7F: Vendor Specific *Otherwise: reserved*

The Trace ID field is vendor specific and allows the device to intermix multiple different traces together at a 1KB/512B granularity.

Debug Trace Sequence Number

A USB3 Device-hardware-controller typically buffers a local copy of the data in case it needs it for a link retry. This is common for the typical USB 2.0 device controller. See Figure 8-7.

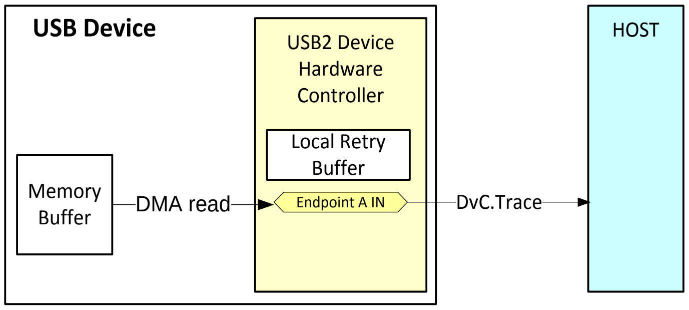


Figure 8-7: Local Retry Buffer within USB3 Device Controller

However, the high bandwidth of SuperSpeed makes such buffering expensive, and some implementations of the device controller do not provide this local retry buffering. Instead, the device controller re-fetches the data from the memory buffer if a link retry is necessary. Typically, buffers in memory are very large (many MB in size) and thus large enough to cover the retry latency. Thus the application is unlikely to have overwritten the data necessary for the retry. Furthermore, the software manages the buffer pointers such that overwriting of the data in memory is impossible.

The situation for debug trace buffers is different. Because of cost constraints, typical trace buffers are small in size, and in addition the hardware state machine only provides elementary pointer management logic that cannot handle overflows. Consequently, in such a simple implementation, the data required for the USB-link retry may have been overwritten. Figure 8-8 shows a typical debug scenario, with the USB3 device controller making a retry request from the trace buffer.

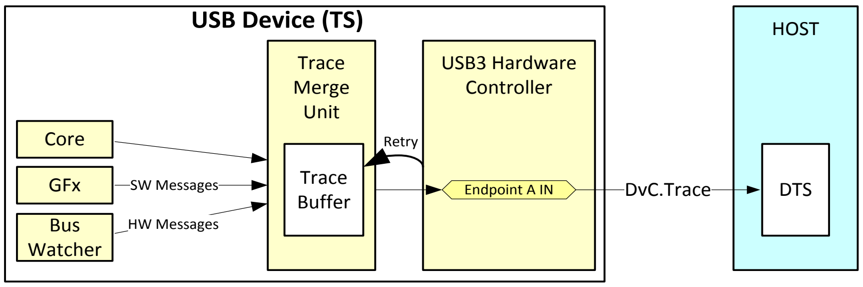


Figure 8-8: No Local Retry Buffer within the USB3 Device Controller

For some debug traces this is of no concern – they can tolerate occasional dropped trace data. The Trace Payload format described here is designed to handle such scenarios.

Bulk transfers provide guaranteed delivery across the USB link by retrying a failed transfer. However, for the situation described above, the retry will deliver the wrong data if the required data was overwritten in the trace buffer. The debug trace header/footer provides a sequence number that increments whenever the TS dispatches a trace packet from the trace buffer. Under normal operation, the debugger will receive an incrementing sequence number. If a retry resulted in trace data being lost, then the sequence number will have a gap in the sequence. The debugger running on the host thus knows when there is a gap in the trace data, and act accordingly.

Bulk retry errors are expected to be rare, and thus loosing data occasionally in a debug trace is an acceptable tradeoff for a simpler and cheaper implementation. Note that since most USB 2.0 device controllers contain local retry buffers, and thus no trace data will be lost because of retries. Hence, initial debug, when the USB link is possibly unreliable, should use HighSpeed transfers for guaranteed delivery of traces (albeit at a lower bandwidth).

Note that an implementation may be aware of the fact that it is unable to supply the correct data for a link retry from the trace buffer. For example, an implementation may have a 4KB trace buffer. Under normal operations, the USB3 device controller will DMA read consecutive 1KB SuperSpeed packets. However, for a retry, the DMA address will be for an earlier location. For example, the DMA may be requesting the first 1KB buffer entry instead of the expected third 1KB buffer entry. In other words, the DMA read sequence was for payloads 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 1,…

In this case, the hardware can mark the “Bulk-Retry occurred” bit in the Status field of the header/footer, thus giving the debugger additional information that a Bulk-retry occurred.

Figure 8-9 shows an example of such a bulk retry. In this example, the Host issues IN requests with NumP = 4, and thus each IN fetches four Data Packets, each of 1KB max-packet size. There are two different sequence numbers shown in the figure. The Sequence number within the data payload (i.e., 21, 22, 23, etc.) corresponds to the sequence number in the Debug Header. The other sequence number is the sequence number used by the USB 3.1 protocol layer to signal which Data Packet to resend for a Bulk retry. This Sequence number starts at 0 in the figure.

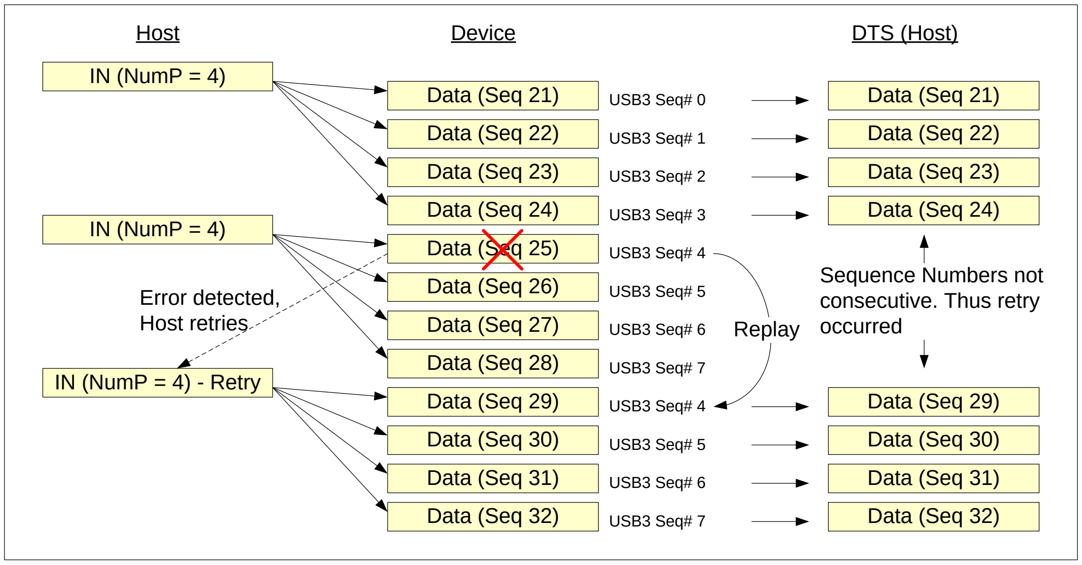


Figure 8-9: Bulk Retry example

Suppose that the Data packet corresponding to USB 3.1-Sequence number 4 has errors. The host xHC will thus restart the requests from sequence 4 and continue consecutively from that point onward. Note that although the host received the data packets corresponding to USB 3.1 Sequence number 5, 6, and 7 correctly, it will still refetch these. Hence, in this example, the host refetched the data packets corresponding to USB 3.1 sequence number 4, 5, 6, and 7.

However, in this example, the retry request arrived late at the debug-trace buffer logic, after the TS has overwritten the trace buffer. Thus, the debug-trace buffer has progressed as far as the data packet corresponding to Debug-Sequence number 29.

Consequently, when the USB 3.1 Device hardware controller attempts to refetch the retry data, the trace buffer will return data corresponding to Debug-Sequence number 29. The debugger will thus notice there is a discontinuity in the Debug Sequence numbers (see right-hand side of Figure 8-9). In this example, they jump from 24 to 29. The debugger thus knows that a retry occurred on data 25-28, and that it did not receive this data.

Isochronous Operations and “NULL” data

A Debug trace data is sporadic and bursty. Consequently, the debugger running on the host has no idea how much trace data the TS has generated at any given time; instead, the debugger shall assume the worse-case scenario, and issue IN packets corresponding to the maximum trace bandwidth.

The USB architecture does provide NRDY and Zero-length packets to deal with such bursty traffic, but not all host controllers handle these operations efficiently. Thus, although the TS could inform the host that it has no data (via a NRDY or zero-length packet), the xHC devices could take a long time before it re-requests the trace data. For example, for isochronous transfers, the xHC will wait until the next service interval, which could be 256uS later or even longer. Such a long delay could result in a small debug trace buffer (e.g., 48KB) overflowing.

For this reason, we allow the TS to supply Null packets if it has nothing to send, thus avoiding the need for NRDY or zero-length packets. Thus, the debugger is constantly requesting debug trace data at the maximum rate appropriate for the particular trace type (e.g., 30MB/s for printf-type software messages, or 400MB/s for processor traces). The debug logic in the TS will satisfy these requests by either supplying the actual trace data if it is available, or otherwise supplying a Null trace packet (i.e., Status [No Trace data] = 1).

Figure 8-10 shows an example with interspersed “NULL” data packets when the debug Trace buffer does not have debug trace-data available. The debugger simply discards the NULL packets.

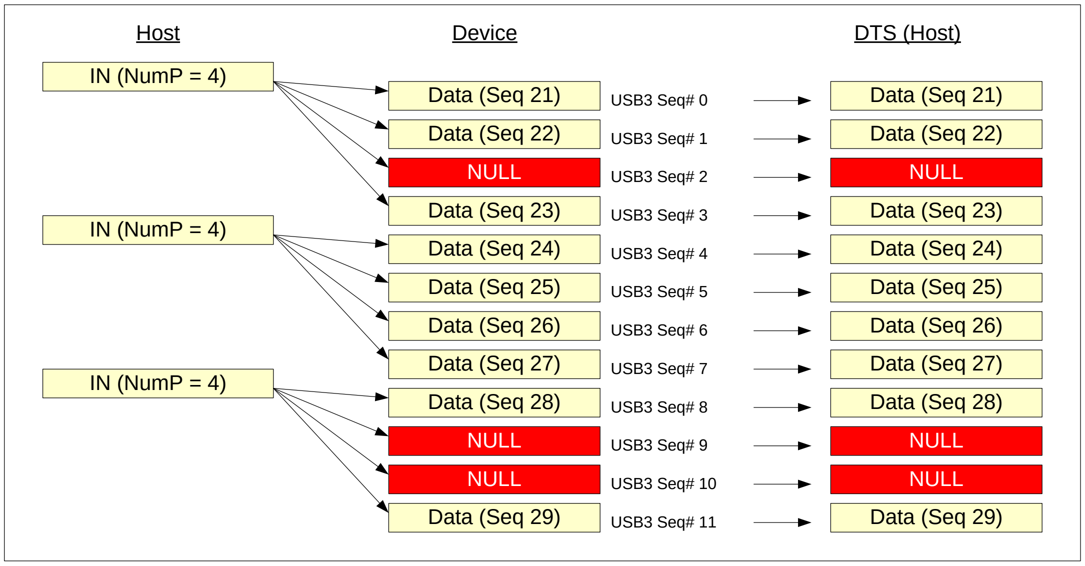


Figure 8-10: Isochronous Example showing interspersed “NULL” data

Note that the Debug Sequence number does not increment for a Null packet. All valid trace data has a monotonically increasing Debug sequence number, modulo the sequence field width.

Appendix D: Power Management

The Debug Class does not address the USB 3.1 Power management features and capabilities. These are vendor specific – some implementations may allow debug to coexist with the link-power management, while other will simply ignore the U1 and U2 link power state change requests and remain in U0. This is most likely the case with DbC, since the DbC implementation does not cost many gates.

The DbC or DvC cannot refuse a U3 request. However, during a debug session, the debugger is likely to configure the host software to not evoke U3.

Appendix E: Example Debug Scenarios

Software Stack Model

Figure 8-11 shows an example of a Software stack creating Software instrumentation messages. The traces then drive a Hardware MIPI STM Trace-Processing unit. The traces use the DvC.Trace interface while the configuration and control of the software stack is done via the Debug-Control Interface.

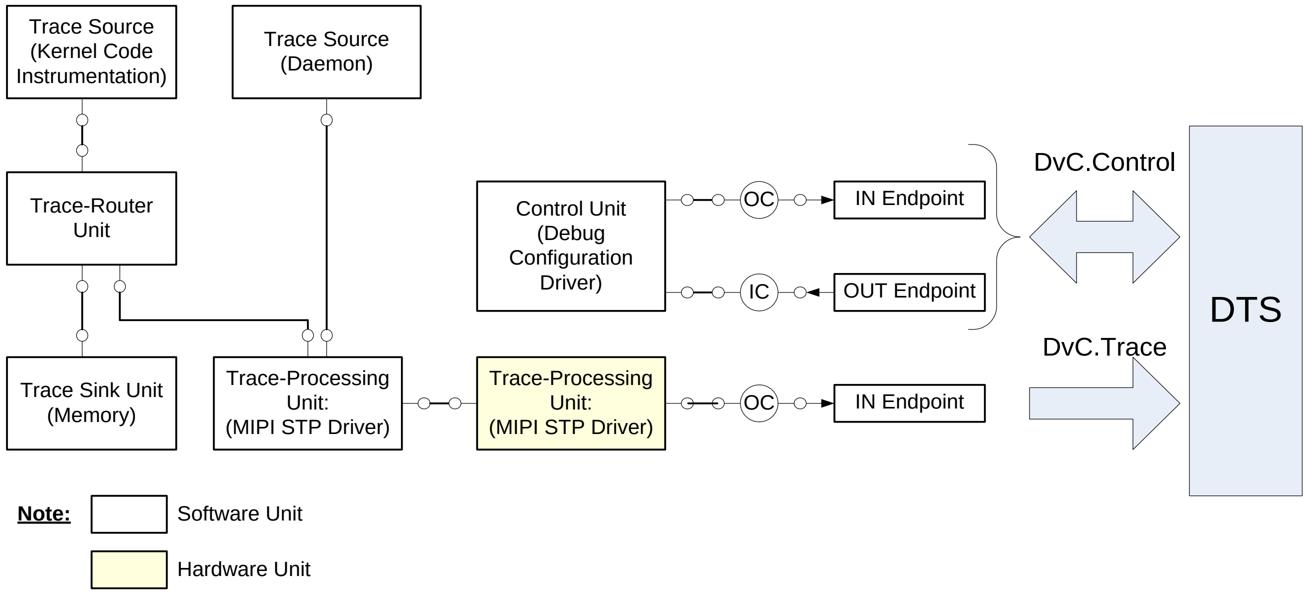

Trace Source (Kernel Code Instrumentation)
Trace Source (Daemon)
Trace-Router Unit
Control Unit (Debug Configuration Driver)
OC
IN Endpoint
IC
OUT Endpoint
DvC.Control
DTS
DvC.Trace
Trace Sink Unit (Memory)
Trace-Processing Unit: (MIPI STP Driver)
Trace-Processing Unit: (MIPI STP Driver)
OC
IN Endpoint
Note:
Software Unit
Hardware Unit

Figure 8-11: Example of a Software Stack driving traces to a Hardware Trace-Processing unit

TS as Host

Figure 8-12 shows two examples where the TS is a USB host and is streaming traces to an external USB device that captures the trace. Example 1 is a simple mass-storage device. In this case, the TS has a driver that is streaming the traces to a mass-storage device. This is out of scope of the USB 3.1 Debug Class specification.

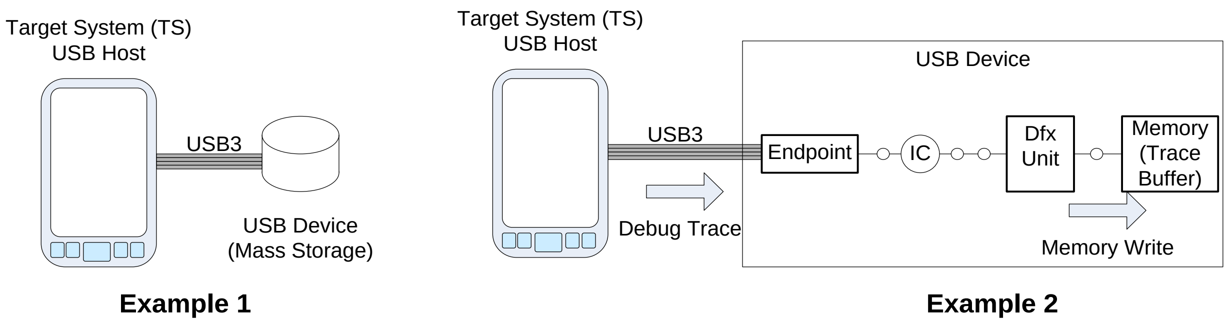


Figure 8-12: Two examples showing the TS streaming traces to external devices

Example 2 is similar to example 1, except that the TS is now streaming traces to a custom device that supports the USB 3.1 Debug Class. In particular, this example supports the DvC.Dfx capability, which provides access to a trace buffer in memory. The debugger could be an application running on the TS device itself, and could thus configure the external debug device. This detail is not shown in the figure.

Alternatively, this external debug device could be a debug probe that an external DTS configures via a private connection, and the DTS configures the TS via a JTAG connection to enable tracing:

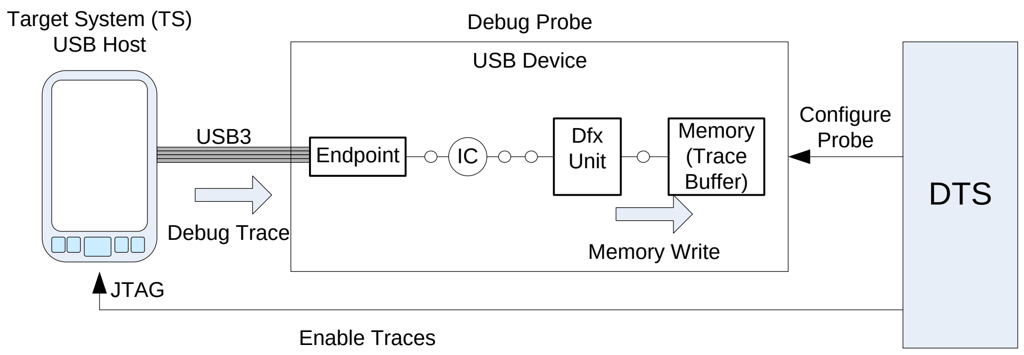


Figure 8-13: Debug Probe providing DvC.Dfx support

Appendix F: Software Stack Overview

Figure 8-14 shows an example of a host connected to a device, showing the different hardware and software layers. Note that normal (non-debug) USB 3.1 applications and debug applications can co-exist and run concurrently. This document defines the Debug Class driver running on the host and the descriptors that the debug driver on the TS supplies during enumeration.

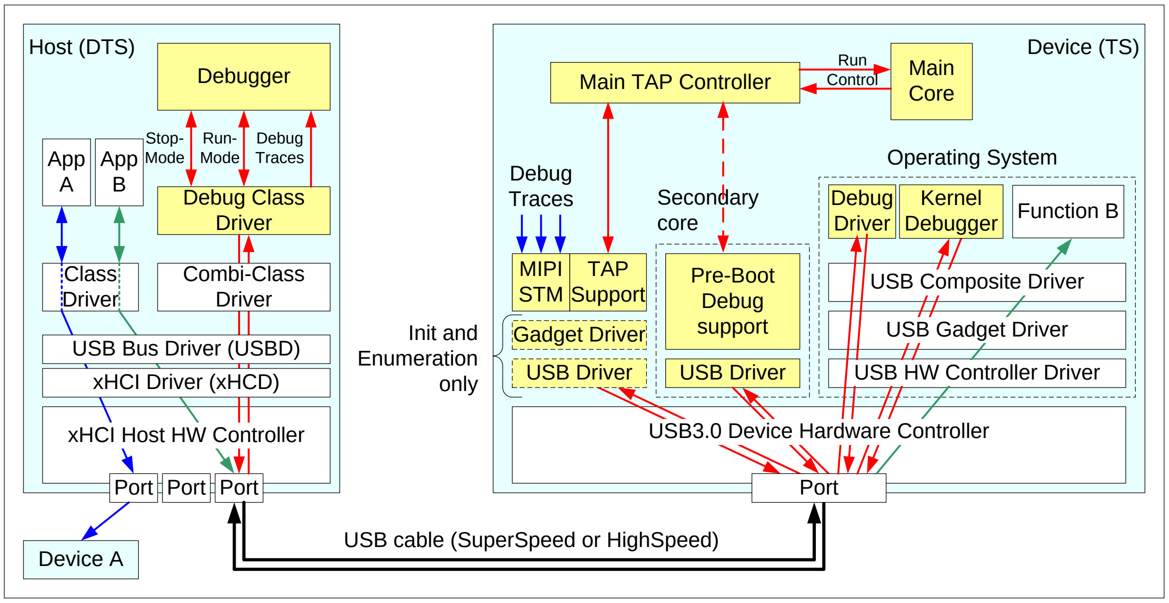


Figure 8-14: DvC Debug Mode S/W stack example

For comparison, Figure 8-15 shows the DbC stack on a multi-port TS. This example has a single DbC. From the host's perspective, the DbC appears similar to the DvC. For example, an OTG device may support both DbC and DvC. The host DTS shall support both, depending on which USB cable links the device to the host.

The USB stack on the device consists of four components:

1) USB Device Hardware Controller Driver: this directly communicates with the USB controller hardware. It is a hardware abstraction layer that exports the hardware functionalities to the layers above.
2) USB Gadget driver: this provides the basic USB framework support, such as managing the USB state transitions (Attached, Powered, Default, Addressed, and Configured), endpoint 0 enumeration, and so on.
3) USB Composite Driver: This provides support for composite (multi-function) USB devices. Note that debug <u>requires</u> a composite driver when using a DIC, because this involves two or more interfaces (e.g., Debug Control together with DvC.Trace). In addition, debug may run at the same time as a normal USB interface (e.g., mass storage), which is a further reason for a composite driver.
4) USB Class drivers: these implement the application functionality of the device, which is generally independent of the USB protocol. These drivers provide the descriptor information (i.e., Interface type, endpoint type, etc.) for a given function. In a multi-function device. The composite driver blends these descriptors together so that they represent the multi-function device. Thus, a GET_DESCRIPTOR command received by the USB device goes all the way up the USB stack (USB Controller driver → USB Gadget driver → USB Composite driver → Class driver) to gather and assemble the appropriate data.

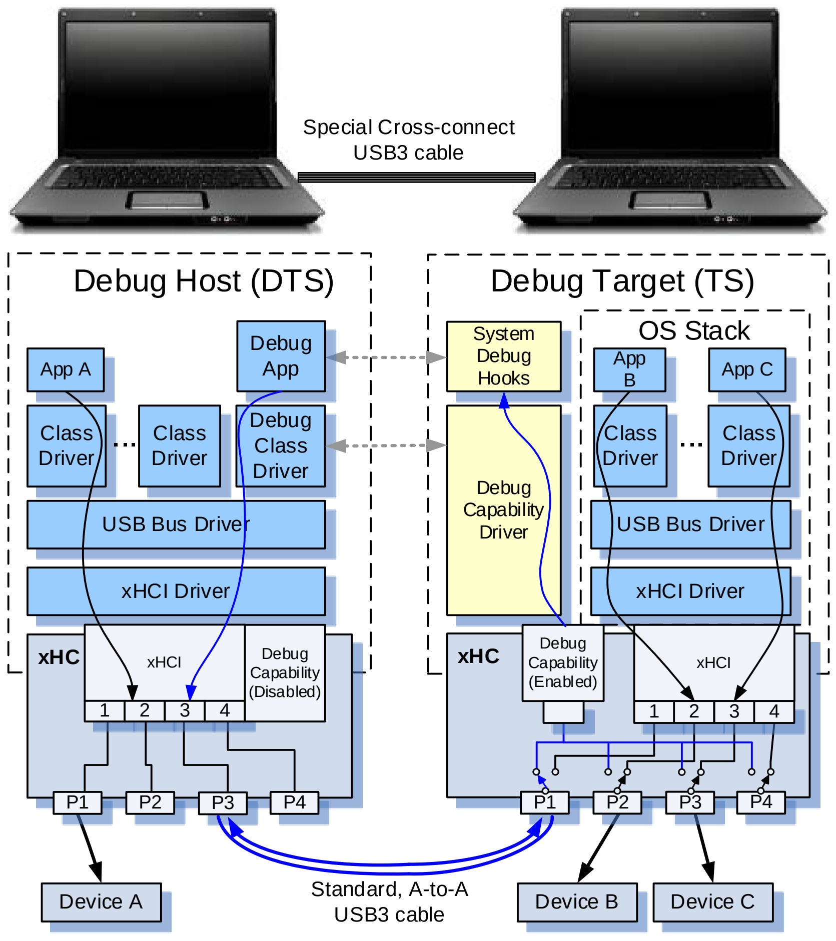


Figure 8-15: The xHCI DbC Software stack

The Debug Class driver thus provides the descriptor information for enumerating the debug capabilities, as defined by the USB 3.1 Debug Class specification. A typical USB Class driver provides three capabilities:

- USB Characteristics: this routine provides the descriptor information and the interaction with the composite driver. A debug use case may change the descriptors between sessions (e.g., to use a different debugger via new descriptors that define an Alternate Setting), which then becomes active when the host resets or reconfigures the USB device.
- Class Operations: the USB 3.1 Debug class allows for basic control operations, such as writing to a configuration register in a debug unit, such as the Trace Processing unit, and thus enabling trace output. See Section 5 for more details.
- File system: this allows the device to interact with the OS file system. DvC.Dfx and DvC.Trace typically use hardware buffers and so do not access the OS file system, whereas the DvC.GP may.

In addition, Figure 8-14 shows that some secondary core may also provide USB stack, which will provide debug support prior to the OS boot. This is implementation specific.