MCCI Universal Serial Bus Windows Kernel Bus Interface for USB 3.0 Streams Device Drivers

Revision 1.0rc1

March 22, 2010

MCCI Corporation Document 950001001 rev B

Preliminary
See disclaimer in front matter

Revision History

Rev	Date	Filename	Comments
0.9	2/10/2010		First public release
1.0rc1	3/16/2010		Changes to match results from first implementation. Add GetDeviceOperatingSpeed and GetProviderInfo APIs.

Please send comments or questions to: usb3streams@mcci.com.

Microsoft Word source files for this document are available to interested parties on request.
tact usb3streams@mcci.com.

Contributors

Terry Moore	MCCI
Chris Yokum	MCCI
Trenton Henry	MCCI

Table of Contents

List of figures

List of Tables

No table of figures entries found.

1 Introduction

1.1 Purpose

This specification provides a portable, USB 3.0 host stack independent API for use by Windows kernel drivers for accessing U SB 3.0 features that are not available through standards Windows kernel APIs, including U SB 3.0 streams.

1.2 Scope

This specification defines and specifies a Windows kernel bus interface, as returned by IRP_MN_QUERY_INTERFACE. Class drivers that need binary compatibility across different host stacks may (but are not required to) use this API to isolate themselves from the details of USB 3.0 implementations.

Class driver vendors may (but are not required to) provide filter drivers that implement this API to map to specific host stacks that provide USB 3.0 functions but don't implement this API.

Host stack driver vendors may (but are not required to) implement this API directly in their host stack.

No existing APIs from the Microsoft Windows stack are redefined or overloaded. Because of the use of bus interfaces, no conflicts are possible between various implementations, and the implementation decisions are deferred to the host stack and class driver vendors.

This specification is only applicable to Windows operating systems, and to kernel mode drivers.

1.3 Related Documents

[USB20]	Universal Serial Bus Specification, revision 2.0. http://www.usb.org
[USB30]	Universal Serial Bus Specification, revision 3.0. http://www.usb.org. Unless otherwise specified, any reference to [USB30] includes [USB20] by reference, especially when referring to full- and high-speed devices.
[WDK-INTERFACE]	Microsoft WDK documentation on IRP_MN_QUERY_INTERFACE, http://msdn.microsoft.com/en-us/library/ms806486.aspx
[WDK-FILTER]	Microsoft WDK documentation on filter drivers, http://msdn.microsoft.com/en-us/library/ms795035.aspx and http://msdn.microsoft.com/en-us/library/aa490247.aspx
[WDK-PIPE-INFO]	Microsoft WDK documentation on USBD_PIPE_INFORMATION, http://msdn.microsoft.com/en-us/library/ms793357.aspx

1.4 Terms and Abbreviations

Term	Description
ABI	Application Binary Interface – a binary compatible programming interface, a more specific form of API. ABIs are typically defined in terms of data structures and function pointers.
API	Application Programming Interface – a source (or binary) compatible programming interface. APIs are typically defined in terms of data structures and function calls.
Bus Interface	An abstract, general-purpose application programming interface (API) and application binary interface (AB) mechanism provided for use by Windows kernel-mode drivers. Bus Interfaces are similar to class driver specifications, in that they are defined by a formal specification of behavior at an interface point.
Descriptor	Data structure used to describe a USB device capability or characteristic.
Device	A logical or physical entity that receives a device address during enumeration.
Filter Driver	In the Microsoft kernel driver architecture, a filter driver modifies the behavior of an existing driver, typically by providing additional APIs or functionality. See [WDK-FILTER].
Function	A collection of one or more interfaces in a USB device, which taken together present a specific capability of the device to the host.

1.5 Editorial Notes

In this specification, the word 'shall' is used for mandatory requirements, the word 'should' is used to express recommendations and the word 'may' is used for options.

2 Management Overview

USB 3.0 introduces a number of new features that are visible to software, such as streams, bursting, and additional parameters for periodic endpoints.

Microsoft's USB 2.0 APIs do not support these features.

MCCI does not wish to redefine or extend existing Microsoft APIs, for a number of reasons. In particular, anything MCCI does might conflict with what Microsoft eventually chooses to do.

Instead, MCCI in this document defines a portable API that is highly abstract. Unlike traditional USB APIs, substantial information is hidden from the client. The portable API is intended to be implemented as a library by the host stack or by a filter driver. It uses Microsoft's standard "Bus Interface" feature to allow for dynamic linkage between the client and the API.

Since the API can be implemented in a number of ways, and it is distinct from the host stack, we use the term "provider" to refer to the API implementation itself.

Conceptually, the provider functions as an adapter, or translator, that allows clients employing the MCCI Abstract USB 3.0 API to communicate with a host stack using the native APIs that are specific to that host stack. A specific provider receives requests from its client(s). In response to those requests, the provider allocates, builds and submits native host stack API requests. When the native requests complete, the provider translates the results back to the MCCI Abstract USB 3.0 form expected by the client. A number of library functions allow the client to adjust assumptions in the interval between building and submitting a request, and to allow the client to obtain results.

The design reduces efficiency slightly, but it has a number of benefits.

- Class drivers can be binary compatible with a variety of host stacks.
- Host stacks can arrange their internal data structures for their own convenience; no internal host stack data structures need be exposed by this API.
- Information about the capabilities of the host controller hardware is naturally incorporated into the API. Depending on the host stack, the needed information can be obtained by querying the stack, or it can be obtained from knowledge built into the provider.

The API design allows several implementation approaches. Figure 2-1, shows the simplest case, where the API provider is integrated into the host stack directly.

Figure 2-1 - Native Integration

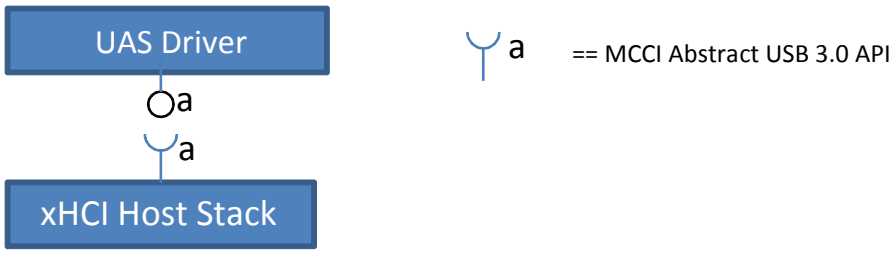


Figure 2-2 shows a more complex case. The UAS driver is unchanged, as it continues to use API "a" (as well as the normal Microsoft USB 2.0 APIs). But now a filter driver provides the translation services. The filter driver might come from the host stack vendor, or it might come from the class driver vendor.

Figure 2-2 - Filter Driver Mapping to Proprietary API

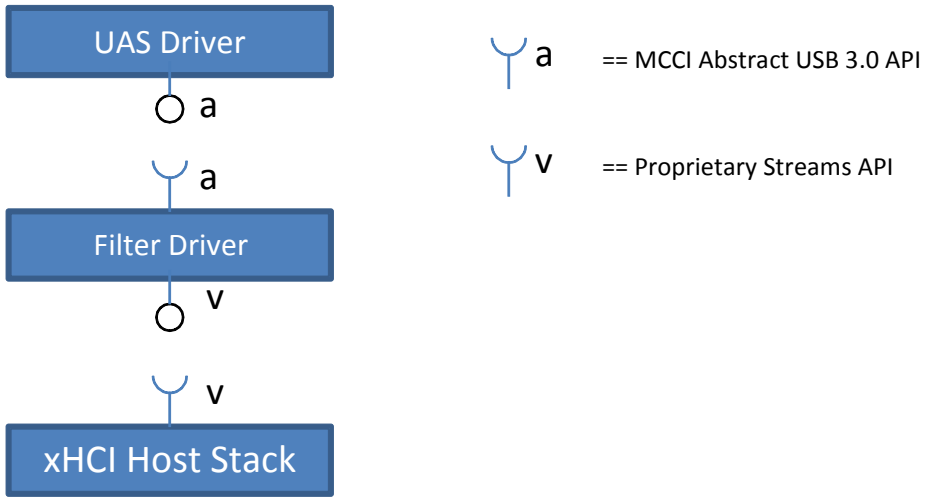
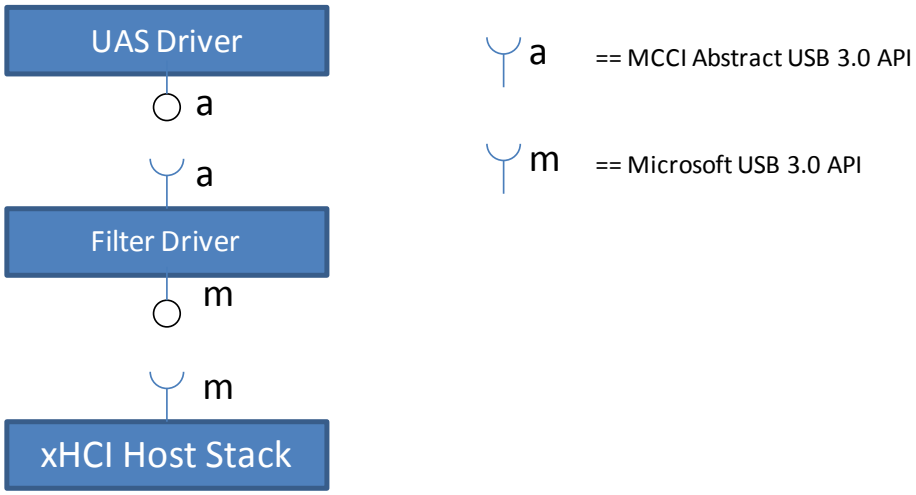


Figure 2-3 shows the situation where Microsoft delivers a stack with a Microsoft-defined USB 3.0 API ("API m"). In this case, the class driver vendor can continue to ship the same driver unmodified for all users, including users who have not yet upgraded to the Microsoft USB 3.0 host stack, by providing a filter driver that maps the abstract USB 3.0 API onto the native Microsoft API.

Figure 2-3 - Filter Driver Mapping to Microsoft USB 3.0 API



In all three cases, the UAS driver can be the same binary: it need not be modified or recompiled. In effect, the adaptation happens dynamically (at runtime).

In the second two cases, no modifications are needed to the host stack, but a filter driver is required.

Finally, it's architecturally easy to provide a single filter driver that detects the underlying host stack at run time, and dynamically selects the appropriate Abstract USB 3.0 API provider needed to support the detected stack.

3 Design Guide

The USB 3.0 specification provides a number of new features beyond those given in the USB 2.0 specification. Many of these features are relevant to class driver writers. This document defines and specifies a Windows kernel bus interface that class driver writers may import, and host driver writers may export, to provide an interim ABI that allows full access to all relevant features of the USB 3.0 specification. Later, when Microsoft ships their USB 3.0 stack, class driver vendors may, as an interim measure, provide filter drivers that adapt from the portable ABI to the final Microsoft API, so that their class drivers can be used without modification with the Microsoft stack, while still being useful with older host stacks until users have migrated to systems using the final Microsoft API.

This section is not a general introduction to the Windows Kernel-Mode Driver Architecture, nor is it an introduction to the theory and use of bus interfaces. Readers should consult the Microsoft documentation [WDK-FILTER], [WDK-INTERFACE] and related pages at www.microsoft.com for more background.

This section includes:

- Querying for the Abstract USB 3.0 interface
- Selecting configurations with USB 3.0 functionality
- Selecting interfaces with USB 3.0 functionality
- Submitting stream I/O operations
- Closing the Abstract USB 3.0 interface

3.1 Querying for the Abstract USB 3.0 Bus Interface

Host stack drivers or filter drivers expose the Abstract USB 3.0 Bus Interface to allow client drivers to use the features of the concrete host stack without knowing which host stack is in use.

Using the Abstract USB 3.0 Bus Interface has several advantages for clients:

- Client driver .sys files don't have to change based on which host stack is present in the system.
- The API allows maximum flexibility to the host stack, so the client driver is more likely to be able to run on any host stack (although perhaps without optimum performance for that stack).

Providing the Abstract USB 3.0 Bus Interface has several advantages for hosts.

- A wider range of client drivers can operate on their host stack.
- The abstract interface still allows host stacks to export higher-performance interfaces that take advantage of special features of the host stack.
- This API is unlikely to collide with the final Microsoft USB 3.0 streams API, and therefore host drivers can readily add the final Microsoft API when it is ready, without breaking existing class drivers.

To access the Abstract USB 3.0 Bus Interface, a client driver must prepare and submit an IRP_MN_QUERY_INTERFACE request to its physical device object, as follows.

First, the client driver allocates an IRP, and initializes it so that the next stack location has major function IRP_MJ_PNP and minor function IRP_MN_QUERY_INTERFACE.

```
pIoStackLocation = IoGetNextStackLocation(pIrp);
pIoStackLocation->MajorFunction = IRP_MJ_PNP;
pIoStackLocation->MinorFunction = IRP_MN_QUERY_INTERFACE;
```

As with all IRP_MN_QUERY_INTERFACE requests, the client must initialize the I/O status block in the IRP prior to calling `IoCallDriver()`:

```
pIrp->IoStatus.Status = STATUS_NOT_SUPORTED;
```

Next, the client driver must allocate memory for the interface structure, and must make the Interface parameter for IRP_MN_QUERY_INTERFACE point to the allocated memory.

```
MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 *pInterface;

pInterface = ExAllocatePool(PagedPool, sizeof(*pInterface));
pIoStackLocation->Parameters.QueryInterface.Interface = pInterface;
```

Next, the client driver may assign a tag (similar to the tag used for `IoAcquireReleaseLock()`) to the `InterfaceSpecificData` parameter for IRP_MN_QUERY_INTERFACE. (If no tag is desired, the client driver should set `InterfaceSpecificData` to NULL. One convenient tag is the device object pointer for the client driver.)

```
pIoStackLocation->Parameters.QueryInterface.InterfaceSpecificData =
      /* arbitrary tag */ pDeviceObject;
```

Next, the client driver must initialize the `InterfaceType` parameter to a pointer to the GUID that identifies the Abstract USB 3.0 Bus Interface.

```
pIoStackLocation->Parameters.QueryInterface.InterfaceType =
      MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_GUID;
```

Next, the client driver must initialize the Version parameter to the version of the interface that is requested. In this case, there is only one version defined, MCCIUSBD_BUSIF_USBD3_ABSTRACT_1.

```
pIoStackLocation->Parameters.QueryInterface.Version =
      MCCIUSBD_BUSIF_USBD3_ABSTRACT_1;
```

The client driver must initialize the Size parameter to the size of the structure allocated for `pInterface` above.

```
pIoStackLocation->Parameters.QueryInterface.Size = sizeof(*pInterface);
```

As with any IRP, the client driver must set a suitable completion routine in order handle IRP completion.

The client driver then sends the IRP to its physical device object.

```
ntStatus = IoCallDriver(pPhysicalDeviceObject, pIrp);
if (ntStatus == STATUS_PENDING)
      // wait for IRP to complete, and set ntStatus to the final
      // status.
```

When the IRP completes, the client driver checks the status. If the status is STATUS_SUCCESS, then the provider has opened an instance of the API for the client, and has placed the API information into the API buffer passed in `Parameters.QueryInterface.Interface`.

3.2 Selecting USB Configurations using the Abstract USB 3.0 API

As with the Microsoft USB 2.0 API, the client driver is responsible for choosing which configuration to enable, and for selecting alternate interface settings. Because most host stack data structures associated with the Abstract USB 3.0 API are opaque to the client, the detailed operations are different, but the concepts are generally the same.

The client driver starts by choosing a configuration, and fetching the associated configuration descriptor bundle. It does this using the normal Microsoft USB 2.0 APIs.

The client then calls `PrepareSelectConfiguration`, passing the relevant configuration descriptor bundle. The Abstract U SB 3.0 API provider parses the bundle, allocating the appropriate IRP and URB structures for the target host stack, allocating a configuration-request handle (`MCCIUSBD_USBD3_HCFGREQUEST`), and associating the handle with the IRP and URB. The provider returns the handle to the client.

Prior to submitting the configuration request, the client can modify the select configuration request. The API provider initializes the Select Configuration handle based on the contents of the client-provided configuration descriptor bundle, assuming certain defaults. Among these assumptions, the provider will initialize the request to select alternate setting zero of each interface; and the provider will set the max packet size for each defined pipe to match the value given in the endpoint descriptor. The client calls CfgRequest_GetHifcInfoCount to find out how many interfaces are defined by the configuration descriptor. The client calls CfgRequest_GetHifcInfo to open an interface information handle. The client uses the interface information handle to call IfcInfo_GetInterfaceInformation to get a copy of the MCCIUSBD_USBD3_INTERFACE_INFORMATION structure for a given interface. The client may change the initial alternate setting and update the stored value by calling IfcInfo_SetInterfaceInformation. The client also uses the interface information handle to call IfcInfo_GetHPipeInfo in order to obtain the pipe information handle for each pipe in the interface.

The client uses a pipe information handle to call PipeInfo_GetUsbd3PipeInformation to fetch a copy of the MCCIUSBD_USBD3_PIPE_INFORMATION for the given pipe. If desired, the client can modify certain values and then update the stored value by calling PipeInfo_SetUsbd3PipeInformation.

After inspecting (and perhaps modifying) the interface and pipe settings, the client submits the request by calling CfgRequest_Submit, passing a completion routine of type MCCIUSBD_HCFGREQUEST_SELECT_CONFIG_DONE_FN and a context pointer. When the host stack has finished processing the request, it calls the client's completion routine.

After a successful completion, the client may call the query functions outlined above to fetch the final values used for each MCCIUSBD_USBD3_INTERFACE_INFORMATION and MCCIUSBD_USBD3_PIPE_INFORMATION.

The client is responsible for closing all handles created above using the appropriate close handle routine (CfgRequest_Close, IfcInfo_Close, or PipeInfo_Close).

3.3 Selecting USB Interfaces using USB 3.0 Functionality

Selecting interfaces is very similar to selecting configurations. PrepareSelectInterface prepares (but does not submit) an appropriate Select Interface request, using information stored during the previous successful PrepareSelectConfiguration/CfgRequest_Submit sequence, and returns an interface request handle. IfcRequest_GetHifcInfo returns the interface information handle for the interface involved in the request. The client calls IfcRequest_Submit to submit the request, passing a pointer to a completion function of type MCCIUSBD_HIFCREQUEST_SELECT_INTERFACE_DONE_FN.

Prior to calling PrepareSelectInterface, the client must have successfully called CfgRequest_Submit, and must provide the configuration request handle to PrepareSelectInterface.

The client is responsible for closing all handles created during this process by calling the appropriate close handle routine (IfcRequest_Close, IfcInfo_Close, or PipeInfo_Close).

3.4 Submitting USB 3.0 Stream I/O Operations

For USB 3.0 streams, the abstract USBD3 API requires two steps for doing I/O.

First, the client must open a stream handle for each stream ID that the client wants to use over a given pipe. This need only be done during initialization, after opening the pipe using CfgRequest_Submit or IfcRequest_Submit, and fetching the pipe handle using PipeInfo_GetUsbd3PipeInformation.

Second, for each I/O operation to be performed, the client must call Stream_PrepareRequest to allocate and initialize an IRP, URB, and any other required data structures needed to perform I/O. Stream_PrepareRequest, if successful, returns a stream request handle.

After preparing the request, the client submits the request using StreamRequest_Submit, passing a pointer to a completion function of type MCCIUSBD_USBD3_HSTREAMRQ_DONE_FN. The request is processed asynchronously by the host stack; when complete, the client's completion function is called in arbitrary thread context.

After a stream request has been submitted, the request cannot be resubmitted. After the client has finished processing, the client must free the request using StreamRequest_Close.

The client can cancel submitted stream requests by calling StreamRequest_Cancel.

3.5 Closing the Abstract USB 3.0 Bus Interface

When done using the Abstract USB 3.0 Bus Interface, the client driver is responsible for closing the interface before by calling

```
(pInterface->InterfaceDereference)(pInterface->Context);
```

Failure to do so may cause the host driver to unload or malfunction when the client driver is unloaded.

Prior to doing so, the client must close any handles allocated by calls to the Abstract USB 3.0 Bus Interface.

3.6 Determining the Maximum Stream ID

Client drivers cannot use the Endpoint Companion Descriptor to determine the maximum stream ID that is available for use. Clients can only use the descriptor to determine the "theoretical maximum stream ID" that the device can support. The actual maximum stream ID must be determined after submitting the HCFGREQUEST or HIFCREQUEST. It will always be less than or equal to the theoretical maximum. USBD may reduce the number of streams if the host controller cannot control as many streams as the device supports. It may also reduce the number of streams to conform to any host stack implementation limitations.

With many host controllers, the hardware limits the maximum stream ID to a value equal to 2^n-1, where n is an integer. On the other hand, the Endpoint Companion Descriptor represents the maximum stream ID (in *bmAttributes*) with the maximum stream ID ranging from 2^1 to 2^{16}; in other words, the maximum stream ID that the device supports is always equal to 2^n. For example, if the Endpoint Companion Descriptor for an endpoint has *bmAttributes* set to 4, it means that the device supports stream IDs [1..16]. However, in this case, the host stack must configure the host controller to support either stream IDs [1..15] or [1..31]; xHCI hardware gives the software no other choices. If the host stack configures the host controller to support stream IDs [1..15], the class driver will get one fewer stream ID than is indicated in the Endpoint Companion Descriptor. If the software configures the host controller to support stream IDs [1..31], then stream IDs (and associated transfer rings, etc) 17..31 allocated by the host controller are wasted (because they represent stream IDs that the device doesn't support).

Furthermore, most real host controllers do not support the full range of stream IDs. They may, for example, be limited to a total of 32 streams for any given endpoint. Again, depending on the hardware design, the host controller may limit the stream ID range to [1..31].

It is very likely that class drivers will encounter situations in which the available maximum stream ID will be less than the maximum stream ID that the device can support. Further, class drivers will encounter situations in which the maximum stream ID will not be an exact power of two.

To accommodate this situation, class drivers must use the *PipeInfo_GetUsbd3PipeInformation* API to obtain the MCCIUSBD_USBD3_PIPE_INFORMATION for each pipe that supports streams. Class drivers must use the value of `TypeSpecificInfo.Bulk.MaxStreamID` to determine the maximum stream ID that is actually available for use, given the host controller hardware and software in use.

4 Reference

4.1 Types Shared between Provider and Client

Three kinds of types are shared between provider and client code

Handles are opaque values that represent state variables that are stored internally by the provider and referenced by the client.

Client callback function types are used by the client to define functions. These functions are then passed by reference from client code to the provider code; later, these functions are called by the provider.

Data structures and enumerations are used by client and provider for input and output parameters. These structures are never used concurrently by client and provider; the client prepares the structure in a client buffer, the provider operates upon the structure (possibly modifying it), copies it if needed for future reference, and then the client uses the results.

API function types are used by the provider to define functions. These functions are then passed by reference from provider code to client code (as a result of IRP_MN_QUERY_INTERFACE).

4.1.1 Handle Types

The following handle types are defined:

- MCCIUSBD_USBD3_HCFGREQUEST
- MCCIUSBD_USBD3_HIFCREQUEST
- MCCIUSBD_USBD3_HIFCINFO
- MCCIUSBD_USBD3_HPIPEINFO
- MCCIUSBD_USBD3_HSTREAM
- MCCIUSBD_USBD3_HSTREAMRQ

These are described in following sections.

4.1.1.1 MCCIUSBD_USBD3_HCFGREQUEST

Function:
Select Configuration Request result handle

Description:
A configuration request handle is returned by a successful call to MCCIUSBD_BUSIFFN_USBD3_PREPARE_SELECT_CONFIGURATION. This handle is opaque to clients. The handle must be closed by the client when it is no longer being used by calling MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_CLOSE. The handle may be duplicated using MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_REFERENCE.

See also:
PrepareSelectConfiguration

4.1.1.2 MCCIUSBD_USBD3_HIFCREQUEST

Function:
Select Interface Request result handle

Description:
An interface-request handle is returned by a successful call to MCCIUSBD_BUSIFFN_USBD3_PREPARE_SELECT_CONFIGURATION. This handle is opaque to clients. The handle must be closed when it is no longer being used by calling MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_CLOSE. The handle may be duplicated using MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_REFERENCE.

See also:
PrepareSelectInterface

4.1.1.3 MCCIUSBD_USBD3_HIFCINFO

Objects of this type are opaque handles for USB 3.0 Interface Information structures.

Description:
An interface-information handle is returned by a successful call to MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_GET_HIFCINFO. This handle is opaque to clients. The handle must be closed when it is no longer being used using MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_CLOSE. The handle may be duplicated using MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_REFERENCE.

See also:
CfgRequest_GetHifcInfo, IfcRequest_GetHifcInfo

4.1.1.4 MCCIUSBD_USBD3_HPIPEINFO

Objects of this type are opaque handles for USB 3.0 Pipe Information structures.

Description:

A pipe-information handle is returned by a successful call to MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_GET_HPIPEINFO. This handle is opaque to clients. The handle must be closed when it is no longer being used using MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_CLOSE. The handle may be duplicated using MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_REFERENCE.

4.1.1.5 MCCIUSBD_USBD3_HSTREAM

Objects of this type are opaque handles for a given stream for a given pipe.

Description:

For USB 3.0 streams, the abstract USBD3 API requires two steps for doing I/O. First, the client must open a stream handle for each stream ID that they want to use. Second, for each I/O request, the client must use the stream handle to prepare a stream request.

A stream handle is returned by a successful call to MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_OPEN_STREAM. The client must release the handle when it is no longer being used using MCCIUSBD_BUSIFFN_USBD3_HSTREAM_CLOSE. The client may duplicate the handle using MCCIUSBD_BUSIFFN_USBD3_HSTREAM_REFERENCE.

4.1.1.6 MCCIUSBD_USBD3_HSTREAMRQ

Objects of this type are opaque handles for a given USB 3.0 stream transfer request.

Description:

For each I/O request, the client must use the stream handle to prepare a stream request using MCCIUSBD_BUSIFFN_USBD3_HSTREAM_PREPARE_REQUEST.

A stream request handle is returned by a successful call to MCCIUSBD_BUSIFFN_USBD3_HSTREAM_PREPARE_REQUEST. The client must release the handle when it is no longer being used, using MCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_CLOSE. The client may duplicate the handle using MCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_REFERENCE

4.1.2 Client Function Types

4.1.2.1 MCCIUSBD_HCFGREQUEST_SELECT_CONFIG_DONE_FN

This type is used to declare completion functions for CfgRequest_Submit.

Definition:

```
typedef VOID
    MCCIUSBD_HCFGREQUEST_SELECT_CONFIG_DONE_FN(
      IN MCCIUSBD_USBD3_HCFGREQUEST hCfgRequest,
      IN NTSTATUS ntStatus,
      IN VOID *pDoneCtx1,
      IN VOID *pDoneCtx2
      );
```

Description:

After a select-configuration request has been submitted and processed to completion, this function is called. `ntStatus` is either `STATUS_SUCCESS` or an error code. `pDoneCtx1` and `pDoneCtx2` are client-supplied context pointers. When this function is called, client regains ownership of the `MCCIUSBD_USBD3_HCFGREQUEST`, and should use the getter methods to obtain the MCCIUSBD_INTERFACE_INFORMATION and MCCIUSBD_USBD3_PIPE_INFORMATION elements.

The client may free the request at any time after it has completed, or may hold the request as long as needed.

The two "done context" pointers are the same values that were provided by the client when submitting the request.

Returns:

No explicit result.

4.1.2.2 MCCIUSBD_HIFCREQUEST_SELECT_INTERFACE_DONE_FN

This type is used by API clients to declare the completion function for IfcRequest_Submit.

Definition:

```
typedef VOID
    MCCIUSBD_HIFCREQUEST_SELECT_INTERFACE_DONE_FN(
      IN MCCIUSBD_USBD3_HIFCREQUEST hIfcRequest,
      IN NTSTATUS ntStatus,
      IN VOID *pDoneCtx1,
      IN VOID *pDoneCtx2
      );
```

Description:

This function is called at the end of processing a SELECT_INTERFACE operation initiated by a call to MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_SUBMIT. The two "done context pointers" are the same pointers that were provided by the client when submitting the request.

Returns:

No explicit result.

4.1.2.3 MCCIUSBD_USBD3_HSTREAMRQ_DONE_FN

This type is used to declare completion routines for StreamRequest_Submit.

Definition:

```
typedef VOID
    MCCIUSBD_USBD3_HSTREAMRQ_DONE_FN(
      IN MCCIUSBD_USBD3_HSTREAMRQ hStreamRq,
      IN USBD_STATUS usbStatus,
      IN NTSTATUS ntStatus,
      IN ULONG TransferBufferLength,
      IN VOID *pDoneCtx1,
      IN VOID *pDoneCtx2
      );
```

Description:

This routine is called in arbitrary thread context when a given stream transfer request has completed. USBD_STATUS is set to the status returned by the host stack, and NTSTATUS is set to the equivalent system status.

TransferBufferLength is set to the actual number of bytes transferred to or from the buffer by this request.

Returns:

No explicit result.

4.1.3 Data Structures and Enumerations

4.1.3.1 MCCIUSBD_USBD_INTERFACE_INFORMATION

Function:

The header portion of a USBD_INTERFACE_INFORMATION.

Description:

It's sometimes inconvenient working with the Microsoft USBD_INTERFACE_INFORMATION, because it also includes one instance of USBD_PIPE_INFORMATION, yet is intended to be variable length. So MCCI defines an equivalent structure without the `Pipes[]` array.

Contents:

Everything in USBD_INTERFACE_INFORMATION except the `Pipes[]` array.

```
USHORT      Length;
UCHAR InterfaceNumber;
UCHAR AlternateSetting;
UCHAR Class;
UCHAR SubClass;
UCHAR Protocol;
UCHAR Reserved;
USBD_INTERFACE_HANDLE   InterfaceHandle;
ULONG NumberOfPipes;
```

See Also:

MCCIUSBD_USBD3_INTERFACE_INFORMATION

4.1.3.2 MCCIUSBD_USBD3_INTERFACE_INFORMATION

Function:

The equivalent of USBD_INTERFACE_INFORMATION, extended to provide USB 3.0 support.

Description:

The standard USBD_INTERFACE_INFORMATION structure used with USB 1.0 and USB 2.0 devices lacks information that is useful for USB 3.0 devices; it also includes an array of USB 2.0 pipe information structures. MCCIUSBD_USBD3_INTERFACE_INFORMATION provides equivalent information, without including the pipe information array.

The provider keeps internal copies of instances of this structure, one for each interface in the configuration bundle supplied by the client to PrepareSelectConfiguration. The client can obtain the current values by calling IfcInfo_GetInterfaceInformation. The client can update some values using IfcInfo_SetInterfaceInformation. For brevity, we indicate that a field can be updated using language "the client can change this value". We emphasize that a field cannot be updated (by the client) using the language "the client cannot change this value". The effect of providing incorrect values when calling IfcInfo_SetInterfaceInformation is implementation-specific.

Contents:

```
USHORT Length;
```

The API provider stets this to the size in bytes of the structure. This is simply `sizeof(MCCIUSBD3_USBD3_INTERFACE_INFORMATION)`.

`UCHAR InterfaceNumber;`
The provider sets this to the `bInterfaceNumber` relevant to this instance.

`UCHAR AlternateSetting;`
PrepareSelectConfiguration initializes this to zero. PrepareSelectInterface initializes this to the `bAlternateSetting` provided by the client. The client may change this value using IfcInfo_SetInterfaceInformation. The client cannot select an alternate setting that was not present in the currently applicable configuration descriptor bundle.

`UCHAR Class;`
Initialized by successful CfgRequest_Submit or IfcRequest_Submit to the `bInterfaceClass` from the associated interface descriptor.

`UCHAR SubClass;`
Initialized by successful CfgRequest_Submit or IfcRequest_Submit to the `bInterfaceSubClass` from the relevant interface descriptor.

`UCHAR Protocol;`
Initialized by successful CfgRequest_Submit or IfcRequest_Submit to the `bInterfaceProtocol` from the relevant interface descriptor.

`USBD_INTERFACE_HANDLE InterfaceHandle;`
PrepareSelectConfiguration and PrepareSelectInterface initialize this field to NULL. After a successful call to CfgRequest_Submit or IfcRequest_Submit, this will be updated to the assigned interface handle.

`ULONG NumberOfPipes;`
PrepareSelectConfiguration and PrepareSelectInterface initialize this field t o the number of pipes in the relevant interface's entry in the configuration descriptor. After ing IfcInfo_SetInterfaceInformation to change `AlternateSetting`, the client should call IfcInfo_GetInterfaceInformation to get the number of pipes in the specified alternate setting.

See Also:
USBD_INTERFACE_INFORMATION, MCCIUSBD_USBD3_PIPE_INFORMATION

4.1.3.3 MCCIUSBD_INTERRUPT_USAGE_TYPE

Function:
Represents the interrupt endpoint usages for USB2 and USB3

Definition:

```
typedef UCHAR MCCIUSBD_INTERRUPT_USAGE_TYPE,
          *PMCCIUSBD_INTERRUPT_USAGE_TYPE;
```

Description:
The following values are defined.

`MCCIUSBD_INTERRUPT_USAGE_TYPE_PERIODIC`
: This value indicates that the pipe is a periodic interrupt pipe -- the pipe must be serviced periodically or the device may not provide good performance. A mouse's INT IN pipe should be marked periodic, for example.

`MCCIUSBD_INTERRUPT_USAGE_TYPE_NOTIFICATION`
: This value indicates that the pipe is a notification pipe. The pipe can be serviced irregularly without any effect on the device's ability to provide good performance. A pipe carrying CDC notifications could be marked as being of this type.

`MCCIUSBD_INTERRUPT_USAGE_TYPE__MAX`
: This constant specifies the maximum legal value for values of type `MCCIUSBD_INTERRUPT_USAGE_TYPE`. All values must be in the range 0≤value≤ `MCCIUSBD_INTERRUPT_USAGE_TYPE__MAX`. This constant can be useful for parameter checking.

Notes:
: The macro `MCCIUSBD_INTERRUPT_USAGE_TYPE__INIT` can be used to initialize a table of debug strings for use when printing this value.

See Also:
: USB 3.0 spec [USB30] section 9.6.6

4.1.3.4 MCCIUSBD_ISOCHRONOUS_USAGE_TYPE

Function:
: Represents an Isochronous Endpoint's usage for USB2 and USB3.

Description:

```
typedef UCHAR MCCIUSBD_ISOCHRONOUS_USAGE_TYPE,
          *PMCCIUSBD_ISOCHRONOUS_USAGE_TYPE;
```

The following values are defined.

`MCCIUSBD_ISOCHRONOUS_USAGE_TYPE_DATA`
: This is a data endpoint.

`MCCIUSBD_ISOCHRONOUS_USAGE_TYPE_FEEDBACK`
: This is a feedback endpoint.

`MCCIUSBD_ISOCHRONOUS_USAGE_TYPE_IMPLICIT_FEEDBACK_DATA`
: This is an "implicit feedback" data endpoint

`MCCIUSBD_ISOCHRONOUS_USAGE_TYPE__MAX`
: The maximum legal value. Useful for parameter checking.

Notes:
: The macro MCCIUSBD_ISOCHRONOUS_USAGE_TYPE__INIT can be used to initialize a table of debug strings for use when printing this value.

See Also:

USB 3.0 spec [USB30] section 9.6.6

4.1.3.5 MCCIUSBD_ISOCHRONOUS_SYNCHRONIZATION_TYPE

Function:

Represents the synchronization type of an isochronous endpoint as declared in the USB2 or USB3 endpoint descriptor.

Definition:

```
typedef     UCHAR MCCIUSBD_ISOCHRONOUS_SYNCHRONIZATION_TYPE,
          *PMCCIUSBD_ISOCHRONOUS_SYNCHRONIZATION_TYPE;
```

Description:

The following values are defined.

`MCCIUSBD_ISOCHRONOUS_SYNCHRONIZATION_TYPE_NONE`
: Endpoint has no synchronization.

`MCCIUSBD_ISOCHRONOUS_SYNCHRONIZATION_TYPE_ASYNCHRONOUS`
: Endpoint is asynchronous.

`MCCIUSBD_ISOCHRONOUS_SYNCHRONIZATION_TYPE_ADAPTIVE`
: Endpoint uses adaptive synchronization.

`MCCIUSBD_ISOCHRONOUS_SYNCHRONIZATION_TYPE_SYNCHRONOUS`
: Endpoint is synchronous.

`MCCIUSBD_ISOCHRONOUS_SYNCHRONIZATION_TYPE__MAX`
: The maximum legal value. Useful for parameter checking.

Notes:

The macro `MCCIUSBD_ISOCHRONOUS_SYNCHRONIZATION_TYPE__INIT` can be used to initialize a table of debug strings for use when printing values of this type.

See Also:

USB 3.0 spec [USB30] section 9.6.6

4.1.3.6 MCCIUSBD_USBD3_PIPE_INFORMATION

Function:

Represents a pipe on a low-speed, full-speed, high-speed or super-speed device.

Description:

This structure is an augmented form of the Microsoft standard USBD_PIPE_INFORMATION. Extra fields are added to allow additional parameters of the USB3 pipe to be observed and controlled.

This structure begins with all the information from USBD_PIPE_INFORMATION; all the extra information is added at the end.

The client obtains information for a given pipe from a configuration request handle or interface request handle by first opening an interface information handle using CfgRequest_GetHifcInfo or IfcRequest_GetHifcInfo, then using that interface information handle to obtain the appropriate pipe information handle using IfcInfo_GetHPipeInfo, and finally ing PipeInfo_GetUsbd3PipeInformation to obtain the pipe information structure for that pipe.

The provider keeps internal copies of instances of this structure, one for each pipe in the configuration bundle supplied by the client to PrepareSelectConfiguration. The client can obtain the current values by calling PipeInfo_GetUsbd3PipeInformation. The client can update some values by modifying fields in its copy of this structure and then calling PipeInfo_SetUsbd3PipeInformation. For brevity, we indicate that a field can be updated using language, "the client can change this value". We emphasize that a field cannot be updated (by the client) using the language, "the client cannot change this value". The effect of providing incorrect values when ing PipeInfo_SetUsbd3PipeInformation is implementation-specific.

Contents:

`USHORT MaximumPacketSize;`

PrepareSelectConfiguration and PrepareSelectInterface set this field to the value given in `wMaxPacketSize` in the Endpoint Descriptor that defines this pipe. For low-speed, full-speed, and super-speed endpoints, for high-speed bulk and control endpoints, and for ordinary high-speed interrupt and isochronous endpoints, this field can be used directly as the maximum packet size. For high bandwidth high-speed interrupt and isochronous endpoints, bits 10..0 specify the maximum packet size in bytes, and bits 12..11 specify the number of additional transactions per micro frame. The client may modify this value using PipeInfo_SetUsbd3PipeInformation after ing PrepareSelectConfiguration (PrepareSelectInterface) and before ing CfgRequest_Submit (or IfcRequest_Submit).

`UCHAR EndpointAddress;`

PrepareSelectConfiguration and PrepareSelectInterface set this field to the value given in `bEndpointAddress` in the Endpoint Descriptor that defines this pipe. Bits 3..0 specify the address, bits 6..4 are reserved, and bit 7 is set to one if this is an IN pipe. The client driver cannot change this value.

`UCHAR Interval;`

PrepareSelectConfiguration and PrepareSelectInterface set this field to the value given in `bInterval` in the Endpoint Descriptor that defines this pipe. The client cannot modify this value. For high-speed and super-speed endpoints, the host controller might not support all possible intervals. If the Interval value is greater than 4 (specifying a polling interval of 16 or greater microframes), the client driver must call PipeInfo_GetUsbd3PipeInformation after a successful CfgRequest_Submit (or IfcRequest_Submit) completion, and must use the value in Interval as the actual polling Interval. The host stack may reduce the Interval in some situations to match the capabilities and actual polling rate of the host controller hardware. Bear in mind, however, that Interval values other than 1, 2, 3 or 4 might not be supported by the concrete host stack.

`USBD_PIPE_TYPE PipeType;`

PrepareSelectConfiguration and PrepareSelectInterface set this field to the value given in bits 1..0 of bInterval in the Endpoint Descriptor that defines this pipe. The client cannot modify this value.

`USBD_PIPE_HANDLE PipeHandle;`
PrepareSelectConfiguration and PrepareSelectInterface set this field to NULL. The client driver must call PipeInfo_GetUsbd3PipeInformation after a ful CfgRequest_Submit (or IfcRequest_Submit) completion to obtain the value assigned as the pipe handle by the host stack. The client cannot modify this value.

`ULONG MaximumTransferSize;`
This value is reserved.

`ULONG PipeFlags;`
PrepareSelectConfiguration and PrepareSelectInterface set this field to zero. The client may modify this value using PipeInfo_SetUsbd3PipeInformation before ing CfgRequest_Submit (or IfcRequest_Submit). Only one flag value is fined, `USBD_PF_CHANGE_MAX_PACKET`. The client must set this flag when changing the maximum packet size using PipeInfo_SetUsbd3PipeInformation.

For more information on the above fields, please see documentation for `USBD_PIPE_INFORMATION` in [WDK-PIPE-INFO].

The structure contains additional fields that are specific to the Abstract USB 3.0 API.

`ULONG Usbd3PipeFlags;`
Flags that can be used to allow clients to override settings from the descriptor in the parameters limited below.

`MCCIUSBD_USBD3_PIPE_INFORMATION_TYPE_SPECIFIC TypeSpecificInfo;`
A union giving information that is specific to the type of pipe being defined.

For interrupt pipes:

`MCCIUSBD_INTERRUPT_USAGE_TYPE TypeSpecificInfo.Interrupt.UsageType;`
For super speed functions, this field represents the usage type of the pipe. PrepareSelectConfiguration and PrepareSelectInterface initialize this field from bits 5..4 of the Endpoint Descriptor. For other speeds, this value is initialized to zero. For Super Speed functions, the client may change this value ing PipeInfo_SetUsbd3PipeInformation before calling CfgRequest_Submit (or IfcRequest_Submit).

`UCHAR TypeSpecificInfo.Interrupt.MaxBurst;`
For super speed functions, this field represents the maximum burst size of the pipe, from 1 to 16. PrepareSelectConfiguration and PrepareSelectInterface initialize this field from the `bMaxBurst` field of the SuperSpeed Endpoint Companion Descriptor. For other speeds, this value is initialized to one. For Super Speed functions, the client may reduce this value using PipeInfo_SetUsbd3PipeInformation before calling CfgRequest_Submit (or IfcRequest_Submit). Note that the host controller might not support all possible max burst values. If the actual maximum burst value is important to the client driver, the

client driver must call PipeInfo_GetUsbd3PipeInformation after a ful CfgRequest_Submit (or IfcRequest_Submit) completion.

`UCHAR TypeSpecificInfo.Interrupt.wBytesPerInterval;`
For super speed functions, this field represents the number of bytes per service val. PrepareSelectConfiguration and PrepareSelectInterface initialize this field from the `wBytesPerInterval` field of the SuperSpeed Endpoint Companion Descriptor. For other speeds, or if the endpoint companion descriptor is not found, this field is reserved and will be set to the effective `wMaxPacketSize`. For Super Speed functions, the client may reduce this value using PipeInfo_SetUsbd3PipeInformation before ing CfgRequest_Submit (or IfcRequest_Submit). For other speeds, clients cannot adjust this value.

For isochronous pipes:

`MCCIUSBD_ISOCHRONOUS_SYNCHRONIZATION_TYPE TypeSpecificInfo.Isochronous.SynchronizationType;`
For high speed and super speed functions, this field represents the synchronization type for this pipe. PrepareSelectConfiguration and PrepareSelectInterface initialize this field from bits 3..2 of the Endpoint Descriptor. For other speeds, this value is initialized to zero. For high speed and super speed functions, the client may change this value ing PipeInfo_SetUsbd3PipeInformation before calling CfgRequest_Submit (or IfcRequest_Submit).

`MCCIUSBD_ISOCHRONOUS_USAGE_TYPE TypeSpecificInfo.Isochronous.UsageType;`
For high speed and super speed functions, this field represents the usage type of the pipe. PrepareSelectConfiguration and PrepareSelectInterface initialize this field from bits 5..4 of the Endpoint Descriptor. For other speeds, this value is initialized to zero. For high speed and super speed functions, the client may change this value ing PipeInfo_SetUsbd3PipeInformation before calling CfgRequest_Submit (or IfcRequest_Submit).

`UCHAR TypeSpecificInfo.Isochronous.MaxBurst;`
For super speed functions, this field represents the maximum burst size of the pipe, from 1 to 16. PrepareSelectConfiguration and PrepareSelectInterface initialize this field from the `bMaxBurst` field of the SuperSpeed Endpoint Companion Descriptor. For other speeds, this value is initialized to one. For super speed functions, the client may reduce this value using PipeInfo_SetUsbd3PipeInformation before calling CfgRequest_Submit (or IfcRequest_Submit). Note that the host controller might not support all possible max burst values. If the actual maximum burst value is important to the client driver, the client driver must call PipeInfo_GetUsbd3PipeInformation after a ful CfgRequest_Submit (or IfcRequest_Submit) completion.

`UCHAR TypeSpecificInfo.Isochronous.MultiplePackets;`
For super speed functions, this field represents the number of packet bursts per service interval (1, 2 or 3). PrepareSelectConfiguration and PrepareSelectInterface initialize this field from bits 1:0 of the `bmAttributes` field of the SuperSpeed Endpoint Companion Descriptor. For other speeds, this field is reserved and will be one. For Super Speed functions, the client may reduce this value using PipeInfo_SetUsbd3PipeInformation before calling CfgRequest_Submit (or IfcRequest_Submit).

`USHORT TypeSpecificInfo.Isochronous.wBytesPerInterval;`

For super speed functions, this field represents the number of bytes per service val. PrepareSelectConfiguration and PrepareSelectInterface initialize this field from the `wBytesPerInterval` field of the SuperSpeed Endpoint Companion Descriptor. For other speeds, or if the endpoint companion descriptor is not found, this field is reserved and will be set to the effective `wMaxPacketSize`. For Super Speed functions, the client may reduce this value using PipeInfo_SetUsbd3PipeInformation before ing CfgRequest_Submit (or IfcRequest_Submit). For other speeds, clients cannot adjust this value.

For bulk pipes:

`UCHAR TypeSpecificInfo.Bulk.MaxBurst;`

For super speed functions, this field represents the maximum burst size of the pipe, from 1 to 16. PrepareSelectConfiguration and PrepareSelectInterface initialize this field from the `bMaxBurst` field of the SuperSpeed Endpoint Companion Descriptor. For other speeds, this value is initialized to one. For super speed functions, the client may reduce this value using PipeInfo_SetUsbd3PipeInformation before calling CfgRequest_Submit (or IfcRequest_Submit). Note that the host controller might not support all possible max burst values. If the actual maximum burst value is important to the client driver, the client driver must call PipeInfo_GetUsbd3PipeInformation after a ful CfgRequest_Submit (or IfcRequest_Submit) completion.

`USHORT TypeSpecificInfo.Bulk.MaxStreamID;`

For super-speed functions, this field represents the maximum number of streams, specified as the maximum stream ID. The value 0 means that no streams are supported. The value 2 indicates that 2 streams (with IDs 1 and 2) are available. The value 4 indicates that 4 streams are available, with IDs 1..4, etc. PrepareSelectConfiguration and PrepareSelectInterface initialize this field based on bits 4..0 of the `bmAttributes` field of the SuperSpeed Endpoint Companion Descriptor Possible initial values based on the Endpoint Companion Descriptor are zero; any power of 2 from 2^1 to 2^{15}; and 65,533 ($2^{16} - 3$).

Note that the value 16 in the endpoint companion descriptor ([USB30], section doesn't mean 65,536 streams, but rather 65,533 streams, numbered from 1 to 65,533. This number is slightly less than 2^{16} because a few stream IDs are reserved. See [USB30], section 8.12.1.4.1 for details.)

The client may reduce this value using PipeInfo_SetUsbd3PipeInformation before ing CfgRequest_Submit (or IfcRequest_Submit). The host stack may allocate fewer streams for a given endpoint, based on resource or other limitations.

After CfgRequest_Submit (or IfcRequest_Submit) completes successfully, the client must check the actual number of streams that were allocated, by fetching the pipe info ing PipeInfo_GetUsbd3PipeInformation. The resulting value will in this field always be less than or equal to the value proposed by the client.

Notes:

There is no `TypeSpecificInfo` for Control pipes.

The following pipe flag mask values are defined for use in `Usbd3PipeFlags`. Note that not every host stack will support all of these flags.

`MCCIUSBD_USBD3_PF_CHANGE_SYNCHRONIZATION_TYPE`
: Change the synchronization type according to the value in `TypeSpecificInfo.Isochronous.SynchronizationType`.

`MCCIUSBD_USBD3_PF_CHANGE_USAGE_TYPE`
: Change the usage type according to the value in `TypeSpecificInfo.*.UsageType`.

`MCCIUSBD_USBD3_PF_CHANGE_MAX_BURST`
: Change the max burst according to the value in `TypeSpecificInfo.*.MaxBurst`.

`MCCIUSBD_USBD3_PF_CHANGE_MULTIPLE_PACKETS`
: Change the "Mult" setting according to the value in `TypeSpecificInfo.*.MultiplePackets`.

`MCCIUSBD_USBD3_PF_CHANGE_BYTES_PER_INTERVAL`
: Change the bytes per interval setting according to the value in `TypeSpecificInfo.*.wBytesPerInterval`.

`MCCIUSBD_USBD3_PF_CHANGE_MAX_STREAMS`
: Change the number of streams according to the value in `TypeSpecificInfo.Bulk.MaxStreams`.

The following combination masks can be used for validating flag values.

`MCCIUSBD_USBD3_PF_CHANGE_VALID_CONTROL`
: The mask of flag values that are permitted for control pipes.

`MCCIUSBD_USBD3_PF_CHANGE_VALID_INTERRUPT`
: The mask of flag values that are permitted for interrupt pipes.

`MCCIUSBD_USBD3_PF_CHANGE_VALID_ISOCHRONOUS`
: The mask of flag values that are permitted for isochronous pipes.

`MCCIUSBD_USBD3_PF_CHANGE_VALID_BULK`
: The mask of flag values that are permitted for bulk pipes.

See Also:

- USB 3.0 specification [USB30] section 9.6.6 and 9.6.7.
- MCCIUSBD_INTERRUPT_USAGE_TYPE
- MCCIUSBD_ISOCHRONOUS_SYNCHRONIZATION_TYPE
- MCCIUSBD_ISOCHRONOUS_USAGE_TYPE

4.1.3.7 MCCIUSBD_USBD3_DEVICE_SPEED

Function:

Represents a USB 3.0 operating speed.

Definition:

```
typedef UINT8 MCCIUSBD_USBD3_DEVICE_SPEED,
      *PMCCIUSBD_USBD3_DEVICE_SPEED;

typedef unsigned ARG_MCCIUSBD_USBD3_DEVICE_SPEED,
      *PARG_MCCIUSBD_USBD3_DEVICE_SPEED;
```

Description:

The following values are defined.

`MCCIUSBD_USBD3_DEVICE_SPEED_LOW`

1.5 megabits/second ("low speed device"). (The value of this symbol is zero.)

`MCCIUSBD_USBD3_DEVICE_SPEED_FULL`

12 megabits/second ("full speed device"). (The value of this symbol is one.)

`MCCIUSBD_USBD3_DEVICE_SPEED_HIGH`

480 megabits/second ("high speed device"). (The value of this symbol is two.)

`MCCIUSBD_USBD3_DEVICE_SPEED_WIRELESS`

Wireless USB device. (The value of this symbol is three.)

`MCCIUSBD_USBD3_DEVICE_SPEED_SUPER`

5 gigabits/second ("super speed device"). (The value of this symbol is four.)

4.2 Interface Structures Exposed by the Abstract API Provider

4.2.1 MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1_GUID

This GUID uniquely identifies the API MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1. In MCCI software, it is defined by the header file "mcciusbd_bus_interface_usbd3_abstract_guid.h". The GUID has the ue {7F2E38BB-70DD-481c-8894-38E29FF2D9E8}.

4.2.2 MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1

A USB host stack driver or a filter driver provides an implementation of MCCIUSBD_BUS_INTERFACE_ABSTRACT_V1, allowing client drivers to use the abstract USBD3 API set. It is identified by the GUID MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1_GUID.

Description:

This structure is returned to clients that open the bus interface identified by MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1_GUID. It provides the method functions needed for accessing this version of the API.

Contents:

```
USHORT Size;
USHORT Version;
PVOID Context;
PINTERFACE_REFERENCE InterfaceReference;
PINTERFACE_DEREFERENCE InterfaceDereference;
```

The structure begins with the normal header of any INTERFACE. The value of `Version` is MCCIUSBD_BUSIF_USBD3_ABSTRACT_1, which is 1.

`InterfaceReference` and `InterfaceDereference` are documented in the Microsoft WDK documentation [WDK-INTERFACE].

```
PMCCIUSBD_BUSIFFN_USBD3_PREPARE_SELECT_CONFIGURATION
PrepareSelectConfiguration;
```

Prepare a "USB3-aware" select-configuration request.

```
PMCCIUSBD_BUSIFFN_USBD3_PREPARE_SELECT_INTERFACE
PrepareSelectInterface;
```

Prepare a "USB3-aware" select-interface request.

```
PMCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_GET_CONFIG_HANDLE
CfgRequest_GetConfigHandle;
```

Getter: returns the USBD_CONFIGURATION_HANDLE from a successful configuration request.

```
PMCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_GET_HIFCINFO
CfgRequest_GetHifcInfo;
```

Getter: return the "interface information" handle for a specific interface instance.

`PMCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_GET_HIFCINFO_COUNT`
`CfgRequest_GetHifcInfoCount`;
: Getter: returns a count of how many "interface information" handles are available for the specified request.

`PMCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_SUBMIT` `CfgRequest_Submit`;
: Submit a "USB3-aware" select-configuration request.

`PMCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_REFERENCE` `CfgRequest_Reference`;
: Create an additional reference to `hCfgRequest`.

`PMCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_CLOSE` `CfgRequest_Close`;
: Retract a reference to `hCfgRequest`.

`PMCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_SUBMIT` `IfcRequest_Submit`;
: Submit a "USB3-aware" select-interface request.

`PMCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_GET_HIFCINFO`
`IfcRequest_GetHifcInfo`;
: Getter: return the "interface information" handle for the interface being modified by SELECT_INTERFACE.

`PMCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_REFERENCE`
`IfcRequest_Reference`;
: Create an additional reference to `hIfcRequest`.

`PMCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_CLOSE`
`IfcRequest_Close`;
: Retract a reference to `hIfcRequest`.

`PMCCIUSBD_BUSIFFN_USBD3_HIFCINFO_GET_IFC_INFORMATION`
`IfcInfo_GetInterfaceInformation`;
: Getter: fetch interface information from `hIfcInfo`

`PMCCIUSBD_BUSIFFN_USBD3_HIFCINFO_SET_IFC_INFORMATION`
`IfcInfo_SetInterfaceInformation`;
: Setter: set interface information to `hIfcInfo`.

`PMCCIUSBD_BUSIFFN_USBD3_HIFCINFO_GET_HPIPEINFO`
`IfcInfo_GetHPipeInfo`;
: Getter: fetch pipe information handle for a specific pipe in `hIfcInfo`.

`PMCCIUSBD_BUSIFFN_USBD3_HIFCINFO_REFERENCE`
`IfcInfo_Reference`;
: Create an additional reference to `hIfcInfo`.

`PMCCIUSBD_BUSIFFN_USBD3_HIFCINFO_CLOSE`
`IfcInfo_Close`;
: Retract a reference to `hIfcInfo`.

`PMCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_OPEN_STREAM`
`PipeInfo_OpenStream`;

Open a handle to a given stream on a given pipe.

`PMCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_GET_USBD3_PIPE_INFORMATION`
`PipeInfo_GetUsbd3PipeInformation;`
Getter: fetch pipe information data from `hPipeInfo`.

`PMCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_SET_USBD3_PIPE_INFORMATION`
`PipeInfo_SetUsbd3PipeInformation;`
Setter: set pipe information data to `hPipeInfo`.

`PMCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_REFERENCE`
`PipeInfo_Reference;`
Create an additional reference to `hPipeInfo`.

`PMCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_CLOSE`
`PipeInfo_Close;`
Retract a reference to `hPipeInfo`.

`PMCCIUSBD_BUSIFFN_USBD3_HSTREAM_PREPARE_REQUEST`
`Stream_PrepareRequest;`
Prepare a "stream-aware" bulk transfer request

`PMCCIUSBD_BUSIFFN_USBD3_HSTREAM_REFERENCE`
`Stream_Reference;`
Create an additional reference to `hStream`.

`PMCCIUSBD_BUSIFFN_USBD3_HSTREAM_CLOSE`
`Stream_Close;`
Retract a reference to `hStream`.

`PMCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_SUBMIT`
`StreamRequest_Submit;`
Submit a prepared stream request for processing by the host stack.

`PMCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_CANCEL`
`StreamRequest_Cancel;`
Cancel a pending stream transfer.

`PMCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_REFERENCE`
`StreamRequest_Reference;`
Create an additional reference to `hStreamRq`.

`PMCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_CLOSE`
`StreamRequest_Close;`
Retract a reference to `hStreamRq`.

`PMCCIUSBD_BUSIFFN_USBD3_GET_DEVICE_OPERATING_SPEED`
`GetDeviceOperatingSpeed;`
Return the current operating speed of the device.

`PMCCIUSBD_BUSIFFN_USBD3_GET_PROVIDER_INFO`
`GetProviderInfo;`
Return a specific property from the API provider.

4.3 Routines Associated with Interface Structures Exposed by the API Provider

Unless otherwise specified, all routines may be called at IRQL <= DISPATCH_LEVEL. Unless otherwise specified, any buffers must be allocated from non-paged pool.

4.3.1 PrepareSelectConfiguration

Type: MCCIUSBD_BUSIFFN_USBD3_PREPARE_SELECT_CONFIGURATION

Function:

Prepare a "USB3-aware" select-configuration request.

Definition:

```
typedef NTSTATUS
    MCCIUSBD_BUSIFFN_USBD3_PREPARE_SELECT_CONFIGURATION(
      IN VOID *pApiContext,
      IN DEVICE_OBJECT *pStackDeviceObject,
      IN CONST USB_CONFIGURATION_DESCRIPTOR *pConfigDesc,
      OUT MCCIUSBD_USBD3_HCFGREQUEST *phCfgRequest
      );

MCCIUSBD_BUSIFFN_USBD3_PREPARE_SELECT_CONFIGURATION
      *pPrepareSelectConfiguration;
```

Description:

This API provides a portable, host-stack-independent way of preparing a USB 3.0 SELECT_CONFIGURATION operation. It takes a configuration descriptor as input, allocates whatever is needed (IRP, URB, etc) by the concrete host stack for the configuration operation, and initializes data structures for the configuration request. It does not send the request downstream. After preparing the request, the client can adjust some parameters by getting and setting the MCCIUSBD_USBD3_PIPE_INFORMATION objects for any pipes that need adjustment. If this routine is successful, it returns a handle that can be used with subsequent get/set operations, and which must be used to submit the SELECT_CONFIGURATION request.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

`pStackDeviceObject` points to the device object returned by `IoAttachToDeviceStack()` or `IoAttachToDeviceStackSafe()`.

`pConfigDesc` points to a configuration bundle that describes the configuration to be selected.

This routine must be called at IRQL PASSIVE_LEVEL.

Returns:

`STATUS_SUCCESS` for success, otherwise an `NTSTATUS` code indicating the reason for the failure.

If successful, `*phCfgRequest` is set to the assigned configuration-request handle. Otherwise `*phCfgRequest` is set to `NULL`.

4.3.2 PrepareSelectInterface

Type: `MCCIUSBD_BUSIFFN_USBD3_PREPARE_SELECT_INTERFACE`

Function:

Prepare a "USB3-aware" select-interface request.

Definition:

```
typedef NTSTATUS
    MCCIUSBD_BUSIFFN_USBD3_PREPARE_SELECT_INTERFACE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HCFGREQUEST hCfgRequest,
      IN UCHAR iInterface,
      IN UCHAR iAltSetting,
      OUT MCCIUSBD_USBD3_HIFCREQUEST *phIfcRequest
      );
```

Description:

This API provides a portable, stack-independent way of preparing a USB 3.0-aware SELECT_INTERFACE operation. It takes a configuration handle as input, allocates whatever is needed (IRP, URB, etc) by the concrete host stack for the SELECT_INTERFACE operation, and initializes data structures for the select-interface request. It does not send the request downstream. After preparing the request, the client can adjust some parameters by getting and setting the MCCIUSBD_USBD3_PIPE_INFORMATION objects for any pipes that need adjustment.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

`hCfgRequest` is the handle obtained by calling PrepareSelectConfiguration. The request must have been successfully submitted prior to issuing this call.

If this routine is successful, it returns a handle that can be used with subsequent get/set operations, and that must be used to submit the SELECT_INTERFACE request by passing it to IfcRequest_Submit.

This routine must be called at IRQL PASSIVE_LEVEL.

Returns:

STATUS_SUCCESS for success, otherwise an NTSTATUS code indicating the reason for the failure.

If successful, `*phIfcRequest` is set to the assigned interface request handle. Otherwise `*phIfcRequest` is set to NULL.

4.3.3 CfgRequest_GetConfigHandle

Type: MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_GET_CONFIG_HANDLE

Function:

Getter: returns the USBD_CONFIGURATION_HANDLE from a successful configuration request.

Definition:

```
typedef USBD_CONFIGURATION_HANDLE
    MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_GET_CONFIG_HANDLE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HCFGREQUEST hCfgRequest
      );
```

Description:

Get the USBD_CONFIGURATION_HANDLE from the result of a submitted SELECT_CONFIGURATION operation.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

Returns a non-NULL configuration handle, or NULL if no configuration handle is available.

4.3.4 CfgRequest_GetHifcInfo

Type: MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_GET_HIFCINFO

Function:

Getter: return the "interface information" handle for a specific interface instance.

Definition:

```
typedef MCCIUSBD_USBD3_HIFCINFO
    MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_GET_HIFCINFO(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HCFGREQUEST hCfgRequest,
      IN ULONG iInterface
      );
```

Description:

A successful SELECT_CONFIGURATION request has an associated collection of interface-information structures, one for each interface descriptor in the configuration descriptor. This function returns a handle for the interface info for a specific interface instance, for use with the HIFCINFO methods of this API.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

Returns a non-NULL interface information handle, or NULL if none is available by the given index.

4.3.5 CfgRequest_GetHifcInfoCount

Type: MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_GET_HIFCINFO_COUNT

Function:

Getter: returns a count of how many "interface information" handles are available for the specified request.

Definition:

```
typedef ULONG
    MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_GET_HIFCINFO_COUNT(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HCFGREQUEST hCfgRequest
      );
```

Description:

A successful SELECT_CONFIGURATION request has an associated collection of interface-information structures, one for each interface descriptor in the configuration descriptor. This function returns the size of the collection.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

Returns the count of HIFCINFOs associated with this MCCIUSBD_USBD3_HCFGREQUEST. Returns 0 if no HCIFINFO is associated with the request.

4.3.6 CfgRequest_Submit

Type: MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_SUBMIT

Function:

Submit a "USB3-aware" select-configuration request.

Definition:

```
typedef NTSTATUS
    MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_SUBMIT(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HCFGREQUEST hCfgRequest,
      IN MCCIUSBD_HCFGREQUEST_SELECT_CONFIG_DONE_FN *pDoneFn,
      IN VOID *pDoneCtx1,
      IN VOID *pDoneCtx2
      );
```

Description:

Submit an URB or other request based on the information in the MCCIUSBD_USBD3_HCFGREQUEST in order to effect the change of configuration. This includes doing any needed bandwidth and/or power allocation, and includes sending the SET_CONFIG command to the device.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

STATUS_PENDING if and only if the request is in flight and the caller must block for completion. Otherwise, the request is already complete (and the user's completion routine has been called).

4.3.7 CfgRequest_Reference

Type: MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_REFERENCE

Function:

Create an additional reference to `hCfgRequest`.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_REFERENCE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HCFGREQUEST hCfgRequest
      );
```

Description:

Each hCfgRequest has a reference count, which is incremented by this API, and decremented by MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_CLOSE. When the reference count goes to zero, the hCfgRequest is disposed.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

No explicit result.

4.3.8 CfgRequest_Close

Type: MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_CLOSE

Function:
Retract a reference to hCfgRequest.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_CLOSE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HCFGREQUEST hCfgRequest
      );
```

Description:
Each hCfgRequest has a reference count, which is incremented by MCCIUSBD_BUSIFFN_USBD3_HCFGREQUEST_REFERENCE, and decremented by this API. When the reference count goes to zero, the hCfgRequest is disposed.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:
No explicit result.

4.3.9 IfcRequest_Submit

Type: MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_SUBMIT

Function:

Submit a "USB3-aware" select-interface request.

Definition:

```
typedef NTSTATUS
    MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_SUBMIT(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HIFCREQUEST hIfcRequest,
      IN MCCIUSBD_HIFCREQUEST_SELECT_INTERFACE_DONE_FN *pDoneFn,
      IN VOID *pDoneCtx1,
      IN VOID *pDoneCtx2
      );
```

Description:

Submit an URB or other request based on the information in the MCCIUSBD3_HIFCREQUEST in order to effect the change of interface setting. This includes doing any needed bandwidth and/or power allocation, and includes sending the SET_INTERFACE command to the device.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

STATUS_PENDING if and only if the request is in flight and the caller must block for completion. Otherwise, the request is already complete (and the user's completion routine has been called).

4.3.10 IfcRequest_GetHifcInfo

Type: MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_GET_HIFCINFO

Function:

Getter: return the "interface information" handle for the interface being modified by SELECT_INTERFACE.

Definition:

```
typedef MCCIUSBD_USBD3_HIFCINFO
    MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_GET_HIFCINFO(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HIFCREQUEST hIfcRequest
      );
```

Description:

A SELECT_INTERFACE request has an associated interface-information structure for the interface being adjusted. This function returns a handle for the interface data, for use in the HIFCINFO methods of this API.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

Returns a non-NULL interface information handle, or NULL if none is available by the given index.

4.3.11 IfcRequest_Reference

Type: MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_REFERENCE

Function:

Create an additional reference to hIfcRequest.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_REFERENCE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HIFCREQUEST hIfcRequest
      );
```

Description:

Each hIfcRequest has a reference count, which is incremented by this API, and decremented by MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_CLOSE. When the reference count goes to zero, the hIfcRequest is disposed.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

No explicit result.

4.3.12 IfcRequest_Close

Type: MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_CLOSE

Function:

Retract a reference to hIfcRequest.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_CLOSE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HIFCREQUEST hIfcRequest
      );
```

Description:

Each hIfcRequest has a reference count, which is incremented by MCCIUSBD_BUSIFFN_USBD3_HIFCREQUEST_REFERENCE, and decremented by this API. When the reference count goes to zero, the hIfcRequest is disposed.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

No explicit result.

4.3.13 IfcInfo_GetInterfaceInformation

Type: MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_GET_IFC_INFORMATION

Function:

Getter: fetch interface information from hIfcInfo

Definition:

```
typedef NTSTATUS
    MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_GET_IFC_INFORMATION(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HIFCINFO hIfcInfo,
      OUT MCCIUSBD_USBD3_INTERFACE_INFORMATION *pIfcInfo
      );
```

Description:

Copy information related to hIfcInfo to the client-supplied MCCIUSBD_USBD3_INTERFACE_INFORMATION structure.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

STATUS_SUCCESS for success, otherwise an NTSTATUS code indicating the reason for the failure.

If successful, the structure at *pIfcInfo is filled in. Otherwise, the contents of *pIfcInfo are not specified.

4.3.14 IfcInfo_SetInterfaceInformation

Type: MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_SET_IFC_INFORMATION

Function:

Setter: set interface information to hIfcInfo.

Definition:

```
typedef NTSTATUS
    MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_SET_IFC_INFORMATION(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HIFCINFO hIfcInfo,
      IN CONST MCCIUSBD_USBD3_INTERFACE_INFORMATION *pIfcInfo
      );
```

Description:

Sets all changeable info for the specified interface from the MCCIUSBD_USBD3_INTERFACE_INFORMATION structure. The caller must to set pipes (and streams) separately from setting the interfaces. The only changeable information is the alternate setting.

Some host stacks may impose further limitations on what you can do with this API.

You can only call this function before you submit the request.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

STATUS_SUCCESS for success, otherwise an NTSTATUS code indicating the reason for the failure.

If successful, the data from structure at `*pIfcInfo` has been copied into the internal interface info. Otherwise, the internal interface info is not modified.

4.3.15 IfcInfo_GetHPipeInfo

Type: MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_GET_HPIPEINFO

Function:

Getter: fetch pipe information handle for a specific pipe in `hIfcInfo`.

Definition:

```
typedef MCCIUSBD_USBD3_HPIPEINFO
    MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_GET_HPIPEINFO(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HIFCINFO hIfcInfo,
      IN UCHAR iPipe
      );
```

Description:

Get a "pipe info handle" for a given pipe index from a given "interface handle". Note that this pipe handle is different than the handle used for submitting URBs.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

Returns a non-NULL pipe information handle, or NULL if none is available by the given index.

4.3.16 IfcInfo_Reference

Type: MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_REFERENCE

Function:

Create an additional reference to `hIfcInfo`.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_REFERENCE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HIFCINFO hIfcInfo
      );
```

Description:

Each `hIfcInfo` has a reference count, which is incremented by this API, and decremented by IfcInfo_Close. When the reference count goes to zero, the `hIfcInfo` is disposed.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

No explicit result.

4.3.17 IfcInfo_Close

Type: MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_CLOSE

Function:

Retract a reference to hIfcInfo.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HIFCINFO_CLOSE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HIFCINFO hIfcInfo
      );
```

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Description:

Each `hIfcInfo` has a reference count, which is incremented by IfcInfo_Reference, and decremented by this API. When the reference count goes to zero, the `hIfcInfo` is disposed.

Returns:

No explicit result.

4.3.18 PipeInfo_OpenStream

Type: MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_OPEN_STREAM

Function:

Open a handle to a given stream on a given pipe.

Definition:

```
typedef MCCIUSBD_USBD3_HSTREAM
    MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_OPEN_STREAM(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HPIPEINFO hPipeInfo,
      IN MCCIUSBD_USBD3_STREAM_ID idStream
      );
```

Description:

For a given stream id, this API prepares an HSTREAM which can be used to prepare I/O to that stream. The API can only be used while the associated configuration is open.

This API is separate from the request launching API in order to allow host stacks maximum flexibility in implementation approaches.

To simplify testing, MCCI's implementation allows this API to be used with any bulk pipe, even if streams have not been configured. If used in this way, `idStream` must be coded as zero.

Returns:

Returns a non-NULL stream handle, or NULL if the given stream ID is not valid for the pipe as configured.

4.3.19 PipeInfo_GetUsbd3PipeInformation

Type: `MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_GET_USBD3_PIPE_INFORMATION`

Function:

Getter: fetch pipe information data from `hPipeInfo`.

Definition:

```
typedef NTSTATUS
    MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_GET_USBD3_PIPE_INFORMATION(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HPIPEINFO hPipeInfo,
      OUT MCCIUSBD_USBD3_PIPE_INFORMATION *pPipeInfo
      );
```

Description:

This function fills in an instance of MCCIUSBD_USBD3_PIPE_INFORMATION with data from the pipe identified by `hPipeInfo`.

MCCIISBD_USBD3_PIPE_INFORMATION is equivalent to the standard USBD_PIPE_INFORMATION structure, but has extra information needed for super speed support.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

STATUS_SUCCESS for success, otherwise an NTSTATUS code indicating the reason for the failure.

If successful, the structure at `*pPipeInfo` is filled in. Otherwise, the contents of `*pPipeInfo` are not specified.

4.3.20 PipeInfo_SetUsbd3PipeInformation

Type: `MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_SET_USBD3_PIPE_INFORMATION`

Function:

Setter: set pipe information data to `hPipeInfo`.

Definition:

```
typedef NTSTATUS
    MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_SET_USBD3_PIPE_INFORMATION(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HPIPEINFO hPipeInfo,
      IN CONST MCCIUSBD_USBD3_PIPE_INFORMATION *pPipeInfo
      );
```

Description:

For a given HPIPEINFO, set attributes from the MCCIUSBD_USBD3_PIPE_INFORMATION. Not all fields can be changed. This only can be done prior to sending down the associated SELECT_CONFIGURATION or SELECT_INTERFACE request.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

STATUS_SUCCESS for success, otherwise an NTSTATUS code indicating the reason for the failure.

If successful, the data from the client structure at `*pPipeInfo` has been copied into the internal pipe information structure. Otherwise, the internal pipe information structure is not modified.

4.3.21 PipeInfo_Reference

Type: MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_REFERENCE

Function:

Create an additional reference to hPipeInfo.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_REFERENCE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HPIPEINFO hPipeInfo
      );
```

Description:

Each `hPipeInfo` has a reference count, which is incremented by this API, and decremented by PipeInfo_Close. When the reference count goes to zero, the `hPipeInfo` is disposed.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

No explicit result.

4.3.22 PipeInfo_Close

Type: MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_CLOSE

Function:

Retract a reference to `hPipeInfo`.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HPIPEINFO_CLOSE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HPIPEINFO hPipeInfo
      );
```

Description:

Each `hPipeInfo` has a reference count, which is incremented by PipeInfo_Reference, and decremented by this API. When the reference count goes to zero, the `hPipeInfo` is disposed.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

No explicit result.

4.3.23 Stream_PrepareRequest

Type: `MCCIUSBD_BUSIFFN_USBD3_HSTREAM_PREPARE_REQUEST`

Function:

Prepare a "stream-aware" bulk transfer request

Definition:

```
typedef NTSTATUS
    MCCIUSBD_BUSIFFN_USBD3_HSTREAM_PREPARE_REQUEST(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HSTREAM hStream,
      IN OPTIONAL PVOID TransferBuffer,
      IN OPTIONAL PMDL TransferBufferMDL,
      IN ULONG TransferBufferLength,
      IN ULONG TransferFlags,
      OUT MCCIUSBD_USBD3_HSTREAMRQ *phStreamRq
      );
```

Description:

This API provides a portable, host-stack-independent way of preparing a stream bulk-in or bulk-out operation. It allocates whatever is needed (IRP, URB, etc) by the concrete host stack for the stream operation. It does not send the request downstream.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

The result is `STATUS_SUCCESS` if successful; otherwise `NT_SUCCESS(result)` will be `FALSE`. If successful, `*phStreamRq` is set to a non-NULL stream-request handle; otherwise `*phStreamRq` is set to `NULL`.

See also:

StreamRequest_Submit, StreamRequest_Cancel, StreamRequest_Reference, StreamRequest_Close

4.3.24 Stream_Reference

Type: MCCIUSBD_BUSIFFN_USBD3_HSTREAM_REFERENCE

Function:

Create an additional reference to hStream.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HSTREAM_REFERENCE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HSTREAM hStream
      );
```

Description:

Each hStream has a reference count, which is incremented by this API, and decremented by MCCIUSBD_BUSIFFN_USBD3_HSTREAM_CLOSE. When the reference count goes to zero, the hStream is disposed.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

No explicit result.

4.3.25 Stream_Close

Type: MCCIUSBD_BUSIFFN_USBD3_HSTREAM_CLOSE

Function:

Retract a reference to `hStream`.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HSTREAM_CLOSE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HSTREAM hStream
      );
```

Description:

Each hStream has a reference count, which is incremented by MCCIUSBD_BUSIFFN_USBD3_HSTREAM_REFERENCE, and decremented by this API. When the reference count goes to zero, the hStream is disposed.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

No explicit result.

See also:

Stream_Reference, PipeInfo_OpenStream

4.3.26 StreamRequest_Submit

Type: MCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_SUBMIT

Function:

Submit a prepared stream request for processing by the host stack.

Definition:

```
typedef NTSTATUS
    MCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_SUBMIT(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HSTREAMRQ hStreamRq,
      IN MCCIUSBD_USBD3_HSTREAMRQ_DONE_FN *pDoneFn,
      IN VOID *pDoneCtx1,
      IN VOID *pDoneCtx2
      );
```

Description:

This request submits a prepared HSTREAMRQ for processing by the host stack.

The current IRQL must be <= DISPATCH_LEVEL.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

STATUS_PENDING if and only if the request is in flight and the caller must block for completion. Otherwise, the request is already complete (and the user's completion routine has been called).

See Also:

Stream_PrepareRequest, MCCIUSBD_USBD3_HSTREAMRQ_DONE_FN

4.3.27 StreamRequest_Cancel

Type: MCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_CANCEL

Function:

Cancel a pending stream transfer.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_CANCEL(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HSTREAMRQ hStreamRq
      );
```

Description:

This request marks an HSTREAMRQ for cancellation. On return from this call, the caller only knows that the HSTREAMRQ will be completed as soon as convenient.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

No explicit result.

See Also:

StreamRequest_Submit

4.3.28 StreamRequest_Reference

Type: MCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_REFERENCE

Function:

Create an additional reference to `hStreamRq`.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_REFERENCE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HSTREAMRQ hStreamRq
      );
```

Description:

Each `hStreamRq` has a reference count, which is incremented by this API, and decremented by StreamRequest_Close. When the reference count goes to zero, the `hStreamRq` is disposed.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

No explicit result.

4.3.29 StreamRequest_Close

Type: MCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_CLOSE

Function:

Retract a reference to `hStreamRq`.

Definition:

```
typedef VOID
    MCCIUSBD_BUSIFFN_USBD3_HSTREAMRQ_CLOSE(
      IN VOID *pApiContext,
      IN MCCIUSBD_USBD3_HSTREAMRQ hStreamRq
      );
```

Description:

Each `hStreamRq` has a reference count, which is incremented by StreamRequest_Reference and decremented by this API. When the reference count goes to zero, the `hStreamRq` is disposed.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

No explicit result.

4.3.30 GetDeviceOperatingSpeed

Type: MCCIUSBD_BUSIFFN_USBD3_GET_DEVICE_OPERATING_SPEED

Function:

Returns the current operating speed of the device.

Definition:

```
typedef MCCIUSBD_USBD3_DEVICE_SPEED
    MCCIUSBD_BUSIFFN_USBD3_GET_DEVICE_OPERATING_SPEED(
      IN VOID *pApiContext
      );
```

Description:

The current operation speed of the device is returned.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

An MCCIUSBD_USBD3_DEVICE_SPEED value is returned

4.3.31 GetProviderInfo

Type: MCCIUSBD_BUSIFFN_USBD3_GET_PROVIDER_INFO

Function:

Returns information about the provider of the API.

Definition:

```
typedef NTSTATUS
    MCCIUSBD_BUSIFFN_USBD3_GET_PROVIDER_INFO(
      IN VOID *pApiContext,
      IN CONST GUID *pPropertyId,
      IN ULONG PropertyVariant,
      OUT VOID *pOutputBuffer,
      IN SIZE_T sizeOutputBuffer,
      OUT SIZE_T *pSizeOutputBufferResult
      );
```

Description:

The provider property identified by `pPropertyID` and the variant value `PropertyVariant` is returned in the buffer given by `pOutputBuffer`. The maximum size returned is `sizeOutputBuffer`. The actual number of bytes returned is written into the variable given by `pSizeOutputBufferResult`.

If `pOutputBuffer` is NULL, `sizeOutputBuffer` must be zero; `pSizeOutputBufferResult` will be set to the number of bytes required for the identified value, and the result will be `STATUS_BUFFER_TOO_SMALL`.

`pApiContext` is the handle returned in the Context member of the MCCIUSBD_BUS_INTERFACE_USBD3_ABSTRACT_V1 structure by the provider in response to an `IRP_MN_QUERY_INTERFACE` request.

Returns:

An appropriate NTSTATUS code is returned.

Notes:

This API is defined for future use. No properties are defined at this time.