IVI-6.2: VISA Interoperability Requirements for USBTMC Specification

November 1, 2018 Edition
Revision 1.0

Important Information

IVI-6.2: VISA Interoperability Requirements for USBTMC Specification is authored by the IVI Foundation member companies. For a vendor membership roster list, please visit the IVI Foundation web site at `www.ivifoundation.org`.

The IVI Foundation wants to receive your comments on this specification. You can contact the Foundation through the web site at `www.ivifoundation.org`.

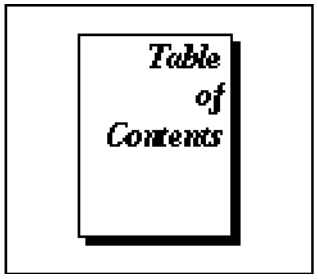
Warranty

The IVI Foundation and its member companies make no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The IVI Foundation and its member companies shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Trademarks

Product and company names listed are trademarks or trade names of their respective companies.

No investigation has been made of common-law trademark rights in any work.



Important Information **2**

Warranty **2**

Trademarks **2**

IVI-6.2 VISA Interoperability Requirements for USBTMC **5**

1. Overview of the VISA Interoperability Requirements for USBTMC Specification **6**

1.1 Introduction 6
1.2 Audience of Specification 6
1.3 Organization of Specification 6
1.4 VISA Interoperability Requirements for USBTMC Overview 6
1.5 References 6
1.6 Definition of Terms and Acronyms 7

2. Windows Specific Information **9**

2.1 CreateFile() 9
2.2 WriteFile() 9
2.3 ReadFile() 10
2.4 DeviceIoControl() 10
2.4.1 IOCTL_USBTMC_GETINFO 11
2.4.1.1 Data structure – USBTMC_DRV_INFO 12
2.4.2 IOCTL_USBTMC_CANCEL_IO 12
2.4.2.1 Data Structures – USBTMC_PIPE_TYPE 13
2.4.3 IOCTL_USBTMC_WAIT_INTERRUPT 13
2.4.4 IOCTL_USBTMC_RESET_PIPE 14
2.4.5 IOCTL_USBTMC_SEND_REQUEST 15
2.4.5.1 Data Structure – USBTMC_IO_BLOCK 16
2.4.6 IOCTL_USBTMC_GET_LAST_ERROR 17
2.5 CloseHandle() 17
2.6 INF files for USBTMC devices 17
2.6.1 Class and ClassGUID for USBTMC devices 17

2.6.2 INF ClassInstall32 section 18
2.6.3 INF DDInstall.Interfaces section 18
2.6.4 Product specific INF files 18
2.6.5 Class, SubClass, Protocol specific INF files 19
2.6.6 Class, SubClass specific INF files and Class specific INF files 19

3. USBTMC include file 20

4. Available USBD Functions 21

Appendix A Linux Specific Information 24

A.1 History 24
A.2 Added Features 24
A.2.1 Changes to Ioctl USBTMC-USB488 READ_STB 24
A.2.2 Support for receiving SRQ notifications via poll/select 24
A.2.3 New for IVI: ioctl USBTMC488_IOCTL_WAIT_SRQ 25
A.2.4 New ioctl USBTMC488_IOCTL_TRIGGER 25
A.2.5 New ioctls USBTMC_IOCTL_GET/SET_TIMEOUT 25
A.2.6 New ioctl USBTMC_IOCTL_CTRL_REQUEST 25
A.2.7 New ioctl USBTMC_IOCTL_EOM_ENABLE 26
A.2.8 New ioctl USBTMC_IOCTL_CONFIG_TERMCHAR 26
A.2.9 New ioctl USBTMC_IOCTL_MSG_IN_ATTR 26
A.2.10 New ioctl USBTMC_IOCTL_WRITE 26
A.2.11 New ioctl USBTMC_IOCTL_WRITE_RESULT 27
A.2.12 New ioctl USBTMC_IOCTL_READ 27
A.2.13 New ioctl USBTMC_IOCTL_CANCEL_IO 28
A.2.14 New ioctl USBTMC_IOCTL_CLEANUP_IO 28
A.2.15 New ioctl USBTMC_IOCTL_AUTO_ABORT 28
A.2.16 New ioctl USBTMC_IOCTL_API_VERSION 28
A.3 Test Functions 28
A.3.1 USBTMC_IOCTL_SET_OUT_HALT 29
A.3.2 USBTMC_IOCTL_SET_IN_HALT 29
A.3.3 USBTMC_IOCTL_ABORT_BULK_IN_TAG 29
A.3.4 USBTMC_IOCTL_ABORT_BULK_OUT_TAG 29

Deleted:
Deleted:
Deleted:
Dele
Dele
De
De
De
De
De
De
De
De
De
De
De
De
De
De
De
Dele
De
De
De
De

IVI-6.2 VISA Interoperability Requirements for USBTMC

VISA Interoperability Requirements for USBTMC Revision History

This section is an overview of the revision history of the VISA Interoperability Requirements for USBTMC specification.

Table 1-1. VISA Interoperability Requirements for USBTMC Specification Revisions

Revision Number	Date of Revision	Revision Notes
Revision 1.0	March 23, 2010	First approved version.
Revision 1.0	November 1, 2018	Editorial change to add the informational appendix describing the Linux USBTMC kernel driver.

1. Overview of the VISA Interoperability Requirements for USBTMC Specification

1.1 Introduction

This section summarizes the *VISA Interoperability Requirements for USBTMC Specification* itself and contains general information that the reader may need to understand, interpret, and implement aspects of this specification. These aspects include the following:

- Audience
- Organization
- Overview
- References
- Terms and Acronyms

1.2 Audience of Specification

The intended readers for this specification are the vendors who wish to interface with the IVI Foundation USBTMC Windows operating systems drivers, and the implementors of the IVI Foundation USBTMC Windows operating systems drivers.

1.3 Organization of Specification

This specification is organized in sections, with each section discussing a particular aspect of the VISA model. Section 3, *Windows Specific Information*, describes the USBTMC Windows operating systems drivers. Section 4, *USBTMC Include File*, presents the USBTMC driver include file. Section 5, *Available USBTMC Functions*, lists the available USBTMC functions.

1.4 VISA Interoperability Requirements for USBTMC Overview

To achieve USBTMC interoperability, a VISA I/O library running in user-space must be able to interact with the USB Host kernel driver stack in a predictable way. The accepted strategy is to specify the kernel API – the system call semantics and behaviors. This enables a VISA I/O library to communicate successfully with the IVI USBTMC kernel driver.

For Windows, specifying an API means specifying the behaviors of CreateFile (), WriteFile (), ReadFile (), DeviceIoControl (), and CloseHandle ().

In addition to the specification of an API, error codes must be specified. This is because a VISA I/O library implementing the USBTMC specification or a USBTMC subclass specification must know when certain error conditions occur.

1.5 References

Several other documents and specifications are related to this specification. These other related documents are the following:

USB Implementers Forum

- USB 2.0 Specification (www.usb.org)
- USBTMC Specification (http://www.usb.org/developers/devclass_docs#approved)

- USBTMC USB488 Specification (http://www.usb.org/developers/devclass_docs#approved)

Microsoft

- Microsoft Platform SDKs for Windows operating systems
- Microsoft DDKs for Windows operating systems

IVI Foundation (www.ivifoundation.org)

- VPP-4.3 & 4.3.x, The VISA Library and detailed VISA and VISA-COM specifications
- VPP-9: Instrument Vendor Abbreviations

1.6 Definition of Terms and Acronyms

The following are some commonly used terms and acronyms used in this document.

API	Application Programmers Interface. The direct interface that an end user sees when creating an application. The VISA API consists of the sum of all of the operations, attributes, and events of each of the VISA Resource Classes.
Host	This is similar to the term "Controller" used in other VISA specifications. From the USB 2.0 specification: "The host computer system where the Host Controller is installed. This includes the host software platform (CPU, bus, etc.) and the operating system in use."
Instrument Driver	Library of functions for controlling a specific instrument
IRP	I/O Request Packet. From the USB 2.0 specification: "An identifiable request by a software client to move data between itself (on the host) and an endpoint of a device in an appropriate direction."
SRQ	IEEE 488 Service Request. This is an asynchronous request from a remote GPIB device that requires service. A service request is essentially an interrupt from a remote device. For USBTMC, this is a notification on the interrupt IN pipe.
USBTMC client software	USBTMC software resident on the host that interacts with the USB System Software to arrange data transfer between a function and the host. The client is often the data provider and consumer for transferred data. A VISA I/O library implementation for USBTMC is the USBTMC client software.
USBTMC device dependent command message	A type of USBTMC command message in which the USBTMC message data bytes are a sequence of bytes defined by the device vendor. Typically a query for a measurement result or a request to change measurement state. Sent from a Host to a device. The VISA method viWrite transfers these messages.
USBTMC interface	A collection of endpoints on a device that conform to the requirements in this USB Test and Measurement Class specification and can be used to provide the physical/signaling/packet connectivity to a Host. The interface descriptor must have bInterfaceClass and bInterfaceSubClass equal to the appropriate values for a USB Test and Measurement Class interface.

USBTMC response message	A type of USBTMC message containing a response to a USBTMC command message. Sent from a device to a Host. The VISA method viRead transfers these messages.

2. Windows Specific Information

The Windows system call behaviors for a USBTMC driver are defined here to resemble the behaviors for a pass-through driver. A pass-through driver allows software to send bulk-OUT and control pipe requests with arbitrary content and to receive bulk-IN and interrupt-IN packets with arbitrary content. The motivation to define system call behaviors in this way are:

- Minimizes the work done by the kernel driver.
- Enables the USBTMC protocol and USBTMC subclass protocols to be implemented in user-space, which is easier to debug and is less likely to crash the system.
- Makes it easy to change the protocol above the kernel driver. The kernel driver does not need to change in order to support changes to the protocol.
- Minimizes the volume of USBTMC interoperability specification material that has to be written and agreed to.
- Multiple protocols may be layered above a pass-through driver.
- Enables using a native pass-through driver if and when it exists.

The sections below describe specific behaviors for each system call. All USB Host USBTMC kernel drivers loaded for Class=0xFE, Subclass=0x03 devices must implement the system call behaviors defined below.

2.1 CreateFile()

Parameters and behaviors are as specified in the Windows documentation.

In addition, for USBTMC, CreateFile() will open transfer pipes to the control endpoint and to each of the USBTMC interface endpoints.

The actual filename of the kernel driver is irrelevant. In other words, user-level client USBTMC code should not be searching for a specific filename. The correct way to find USBTMC resources is to use the class GUID reserved for USBTMC. Windows defines the routines necessary to query kernel filenames that can be passed to CreateFile().

The algorithm for finding all available USBTMC kernel filenames is to first call SetupDiGetClassDevs. This returns a handle to be used with the remaining calls. In a loop, you call SetupDiEnumDeviceInterfaces and SetupDiGetDeviceInterfaceDetail. The SP_INTERFACE_DEVICE_DETAIL_DATA structure gives you the kernel filename to pass to CreateFile(). Finally, release the class handle with the function SetupDiDestroyDeviceInfoList.

A VISA I/O library must always set FILE_FLAG_OVERLAPPED when calling CreateFile(). This is required for waiting on interrupt data while simultaneously performing normal I/O.

In reality, the kernel driver does not know and should not care whether the user calls CreateFile() with or without FILE_FLAG_OVERLAPPED. This text is here mainly for clarification.

The USBTMC kernel driver allows multiple handles to a device to be active at any time.

2.2 WriteFile()

Parameters and behaviors are as specified in the Windows documentation.

In addition, for USBTMC, the data in the buffer passed in is sent unmodified to the bulk-OUT endpoint associated with the file handle.

The implementation of WriteFile() must support transferring data buffers that are larger than the internal maximum transfer size. For example, if the internal USB buffer is 8KB, and the user buffer is 30KB, then the implementation of WriteFile() must transfer the entire user buffer by looping over the buffer (in this case 4 times) and sending a portion from the user buffer each time. The implementation of WriteFile() may send less than the entire transfer count only if an error occurs.

Regardless of whether the user called CreateFile() with FILE_FLAG_OVERLAPPED, the USBTMC kernel implementation of WriteFile() must be able to implement requests asynchronously, in other words, it must be capable of returning STATUS_PENDING. This is not a requirement for all transfer sizes, in other words, it is valid for a USBTMC kernel implementation to perform tiny transfers synchronously. Note that if the user did not call CreateFile() with FILE_FLAG_OVERLAPPED, the operating system will cause the call to WriteFile() to block until the USBTMC kernel driver marks the IRP as complete.

2.3 ReadFile()

Parameters and behaviors are as specified in the Windows documentation.

In addition, for USBTMC, the data received is placed into the specified buffer and is returned unmodified.

ReadFile() must return if a non-maximum length packet is received or the amount of data requested has been received.

The implementation of ReadFile() must support transferring data buffers that are larger than the internal maximum transfer size. For example, if the internal USB buffer is 8KB, and the user buffer is 30KB, then the implementation of ReadFile() must transfer the entire user buffer by looping over the buffer (in this case 4 times) and reading a portion into the user buffer each time. The implementation of ReadFile() may read less than the entire transfer count only if an error occurs or it receives a short packet.

Regardless of whether the user called CreateFile() with FILE_FLAG_OVERLAPPED, the USBTMC kernel implementation of ReadFile() must be able to implement requests asynchronously, in other words, it must be capable of returning STATUS_PENDING. This is not a requirement for all transfer sizes, in other words, it is valid for a USBTMC kernel implementation to perform tiny transfers synchronously. Note that if the user did not call CreateFile() with FILE_FLAG_OVERLAPPED, the operating system will cause the call to ReadFile() to block until the USBTMC kernel driver marks the IRP as complete.

2.4 DeviceIoControl()

Parameters and behaviors are as specified in the Windows documentation.

In addition, for USBTMC, drivers must support the following IOCTL codes. See also section 3.

IOCTL macro	Value	Description
IOCTL_USBTMC_GETINFO	0x8000_2000	Gets information about the Windows USB Host USBTMC driver.
IOCTL_USBTMC_CANCEL_IO	0x8000_2004	Cancels all IRPs on caller-designated pipe.
IOCTL_USBTMC_WAIT_INTERRUPT	0x8000_2008	Waits for data to arrive on interrupt-IN endpoint. If overlapped, the overlapped event is set when interrupt-IN DATA is received.
IOCTL_USBTMC_RESET_PIPE	0x8000_201C	Clears a Halt condition on a pipe.

IOCTL_USBTMC_SEND_REQUEST	0x8000_2080	Sends an arbitrary request to the device control endpoint.
IOCTL_USBTMC_GET_LAST_ERROR	0x8000_2088	Gets the most recent error returned from the lower-level USB driver.

Table 2-1 -- USBTMC IOCTL codes

All DeviceIoControl() requests have the following parameters.

Parameter Type	Parameter Name	Description
HANDLE	hDevice	Device handle, obtained by calling CreateFile.
DWORD	dwControlCode	Operation control code. One of the IOCTL's in Table 1.
LPVOID	lpInBuffer	Input data buffer
DWORD	nInBufferSize	Size of input data buffer
DWORD	lpOutBuffer	Output data buffer
DWORD	nOutBufferSize	Size of output data buffer
LPDWORD	lpBytesReturned	Pointer to a location to receive the number of bytes returned.
LPOVERLAPPED	lpOverlapped	Optional pointer to an OVERLAPPED structure (described in the Windows Platform SDK documentation).

Table 2-2 -- DeviceIoControl parameters

2.4.1 IOCTL_USBTMC_GETINFO

Returns information about the USBTMC driver.

Parameter Name	Description
hDevice	Device handle, obtained by calling CreateFile.
dwControlCode	IOCTL_USBTMC_GETINFO
lpInBuffer	NULL
nInBufferSize	0
lpOutBuffer	Output data buffer. Pointer to a USBTMC_DRV_INFO structure.
nOutBufferSize	Size of output data buffer. Must be sizeof(USBTMC_DRV_INFO).
lpBytesReturned	Pointer to a location to receive the number of bytes returned.
lpOverlapped	Optional pointer to an OVERLAPPED structure (described in the Windows Platform SDK documentation).

When the DeviceloControl function is called with the IOCTL_USBTMC_GETINFO I/O control code, the caller must specify the address of a USBTMC_DRV_INFO structure as the function's lpOutBuffer parameter. The kernel-mode driver fills in the structure members.

Code Example

```
USBTMC_DRV_INFO    drvrInfo;
DWORD              cbRet;

OVERLAPPED      overlapped;
BOOL               bRet;

memset(&overlapped, 0, sizeof(OVERLAPPED));
overlapped.hEvent = CreateEvent(NULL, FALSE,  FALSE, NULL);
bRet = DeviceIoControl( DeviceHandle,
                        (DWORD) IOCTL_USBTMC_GETINFO,
                        NULL,
                        0,
                        &drvInfo,
                        sizeof(USBTMC_DRV_INFO),
                        &cbRet,
                        &overlapped);

if( bRet == TRUE )
    WaitForSingleObject(overlapped.hEvent, INFINITE);

CloseHandle(overlapped.hEvent);
```

- Data structure – USBTMC_DRV_INFO

```
typedef struct {
  DWORD major; // major revision of driver
  DWORD minor; // minor revision of driver
  DWORD build; // internal build number
  WCHAR manufacturer[64]; // unicode manufacturer string
} USBTMC_DRV_INFO, *PUSBTMC_DRV_INFO;
```

As with all Unicode strings, the manufacturer field must be NULL-terminated. This field can contain any valid Unicode character, and this specification does not impose any further restrictions.

Note that this entire data structure (USBTMC_DRV_INFO) and its IOCTL (IOCTL_USBTMC_GETINFO) are for diagnostic purposes. If a future Microsoft pass-through driver does not support this IOCTL or all of its fields, USBTMC client software should not be severely affected. USBTMC client software should not rely on any part of this IOCTL for normal and proper operation.

2.4.2 IOCTL_USBTMC_CANCEL_IO

This IOCTL cancels activity on the specified USB transfer pipe that is associated with the specified device handle.

Parameter Name	Description
hDevice	Device handle, obtained by calling CreateFile.
dwControlCode	IOCTL_USBTMC_CANCEL_IO
lpInBuffer	Pointer to a location containing a USBTMC_PIPE_TYPE-typed value
nInBufferSize	Sizeof(USBTMC_PIPE_TYPE)

lpOutBuffer	NULL
nOutBufferSize	0
lpBytesReturned	0
lpOverlapped	Optional pointer to an OVERLAPPED structure (described in the Windows Platform SDK documentation).

When the DeviceloControl function is called with the IOCTL_USBTMC_CANCEL_IO I/O control code, the caller must specify one of the USBTMC_PIPE_TYPE-typed values as the function's lpInBuffer parameter. This value indicates on which of the transfer pipes (interrupt, bulk IN, bulk OUT) the operation should be performed.

The driver creates an URB with function = URB_FUNCTION_ABORT_PIPE to carry out the request.

Code Example

```
BOOL bState = FALSE;
DWORD cbRet = 0;
OVERLAPPED overlapped;

memset(&overlapped, 0, sizeof(OVERLAPPED));
overlapped.hEvent =
    CreateEvent(NULL, // pointer to security attributes
                FALSE, // automatic reset
                FALSE, // initialize to nosignaled
                NULL); // pointer to the event-object name
bState =
    DeviceIoControl(hdlDevice,
                    (DWORD) IOCTL_USBTMC_CANCEL_IO,
                    (LPVOID)&pipeType,
                    sizeof(USBTMC_PIPE_TYPE),
                    NULL,
                    0,
                    &cbRet,
                    &overlapped);
```

2.4.2.1 Data Structures – USBTMC_PIPE_TYPE

```
typedef enum {
   USBTMC_INTERRUPT_IN_PIPE = 1,
   USBTMC_READ_DATA_PIPE = 2,
   USBTMC_WRITE_DATA_PIPE = 3,
   USBTMC_ALL_PIPES = 4
} USBTMC_PIPE_TYPE;
```

The USBTMC_PIPE_TYPE data type is used as input to the DeviceIoControl function, if the I/O control code is IOCTL_USBTMC_CANCEL_IO or IOCTL_USBTMC_RESET_PIPE. An interrupt pipe, a bulk IN pipe, and a bulk OUT pipe are associated with each device handle supplied to DeviceIoControl. The specified USBTMC_PIPE_TYPE value indicates on which of these pipes the operation should be performed.

2.4.3 IOCTL_USBTMC_WAIT_INTERRUPT

Returns data arriving on a USB interrupt pipe.

Parameter Name	Description
hDevice	Device handle, obtained by calling CreateFile.
dwControlCode	IOCTL_USBTMC_WAIT_INTERRUPT
lpInBuffer	NULL
nInBufferSize	0
lpOutBuffer	Pointer to a buffer that is large enough to receive the largest packet the device is capable of sending on the interrupt pipe. May be large enough to receive several packets.
nOutBufferSize	Size of the output buffer.
lpBytesReturned	Pointer to a location to receive the number of bytes returned.
lpOverlapped	Optional pointer to an OVERLAPPED structure (described in the Windows Platform SDK documentation).

Any application process or thread can issue a DeviceIoControl() with IOCTL_USBTMC_ WAIT_INTERRUPT. The USBTMC kernel driver distributes interrupt-IN DATA to multiple processes. A VISA I/O library may issue a new IOCTL_USBTMC_WAIT_INTERRUPT if one is already outstanding for a given USBTMC interface.

Code Example

```
DWORD dwError;
BOOL bRet;
BYTE InterruptData[64];
OVERLAPPED overlappedIntIn;

memset(&overlappedIntIn, 0, sizeof(overlappedIntIn));
overlappedIntIn.hEvent = CreateEvent( NULL, TRUE, FALSE, NULL );

bRet = DeviceIoControl( hHandle,
                        IOCTL_USBTMC_WAIT_INTERRUPT,
                        NULL,
                        0,
                        &InterruptData,
                        sizeof(InterruptData),
                        &dwError,
                        &overlappedIntIn );
if ( bRet != 0 ) {
   printf("DeviceIoControl err\n");
}

WaitForSingleObject(overlappedIn.hEvent,INFINITE);
```

2.4.4 IOCTL_USBTMC_RESET_PIPE

Resets the specified USB transfer pipe that is associated with the specified device handle. This clears a Halt condition on the pipe.

Parameter Name	Description

hDevice	Device handle, obtained by calling CreateFile.
dwControlCode	IOCTL_USBTMC_RESET_PIPE
lpInBuffer	NULL
nInBufferSize	0
lpOutBuffer	Pointer to a location containing a USBTMC_PIPE_TYPE-typed value.
nOutBufferSize	Size of the output buffer.
lpBytesReturned	Pointer to a location to receive the number of bytes returned.
lpOverlapped	Optional pointer to an OVERLAPPED structure (described in the Windows Platform SDK documentation).

When the DeviceloControl function is called with the IOCTL_USBTMC_RESET_PIPE I/O control code, the caller must specify one of the USBTMC_PIPE_TYPE-typed values as the function's lpInBuffer parameter. This value indicates on which of the transfer pipes (interrupt, bulk IN, bulk OUT) the operation should be performed.

The driver creates an URB with function = URB_FUNCTION_RESET_PIPE to carry out the request.

2.4.5 IOCTL_USBTMC_SEND_REQUEST

Sends a vendor-defined or class-specific request to a USB device, using the control pipe, and optionally sends or receives additional data.

Parameter Name	Description
hDevice	Device handle, obtained by calling CreateFile.
dwControlCode	IOCTL_USBTMC_SEND_REQUEST
lpInBuffer	Pointer to an USBTMC_IO_BLOCK structure.
nInBufferSize	Sizeof(USBTMC_IO_BLOCK)
lpOutBuffer	Pointer to the same buffer identified by the PbyData member of the USBTMC_IO_BLOCK structure, or NULL if a data transfer is not being requested.
nOutBufferSize	Size of the output buffer, or zero if a data transfer is not being requested.
lpBytesReturned	Pointer to a location to receive the number of bytes returned.
lpOverlapped	Optional pointer to an OVERLAPPED structure (described in the Windows Platform SDK documentation).

When the DeviceloControl function is called with the IOCTL_USBTMC_SEND_REQUEST control code, the caller must specify the address of an USBTMC_IO_BLOCK structure as the function's lpInBuffer parameter. The type of request specified with this I/O control code is device-specific and vendor-defined, as are the type and size of any information that might be sent or received.

The USBTMC kernel implementation must support all vendor-specific requests. For class-specific requests, if the Windows operating system implements a given request, then the USBTMC kernel implementation must support that request. If the Windows operating system does not define an implementation for a given request, such as if the bmRequestType or bRequest parameter is an undefined class request, then the USBTMC kernel implementation must return the error STATUS_INVALID_PARAMETER.

The following table shows how input arguments should be specified.

	Read Operation	Write Operation	No data transfer
lpInBuffer	USBTMC_IO_BLOCK pointer.	USBTMC_IO_BLOCK pointer.	USBTMC_IO_BLOCK pointer.
lpOutBuffer	Pointer to buffer that will receive data to be read.	Pointer to buffer containing data to be written.	NULL
lpOutBufferSize	Size of buffer.	Size of buffer.	Zero
pbyData member of USBTMC_IO_BLOCK	Same pointer as lpOutBuffer.	Same pointer as lpOutBuffer.	NULL
wLength member of USBTMC_IO_BLOCK	Same value as lpOutBufferSize.	Same value as lpOutBufferSize.	Zero
fTransferDirectionIn member of USBTMC_IO_BLOCK	TRUE	FALSE	FALSE

2.4.5.1 Data Structure – USBTMC_IO_BLOCK

The USBTMC_IO_BLOCK structure is used as a parameter to DeviceIoControl, when the specified I/O control code is IOCTL_USBTMC_SEND_REQUEST. Values contained in structure members are used to create a USB Device Request (described in the Universal Serial Bus Specification).

```
typedef struct {
  IN unsigned char bmRequestType;
  IN unsigned char bRequest;
  IN unsigned short wValue;
  IN unsigned short wIndex;
  IN unsigned short wLength;
  IN OUT PUCHAR  pbyData;
  IN UCHAR  fTransferDirectionIn;
} USBTMC_IO_BLOCK, *PUSBTMC_IO_BLOCK;
```

USBTMC_IO_BLOCK field	Usage
bmRequestType	Used as the Setup DATA bmRequestType
bRequest	Used as the Setup DATA bRequest
wValue	Used as the Setup DATA wValue
wIndex	Used as the Setup DATA wIndex
wLength	Used as the Setup DATA wLength
pbyData	Pointer to a data buffer with a length of wLength.
fTransferDirectionIn	TRUE for transfers from device to host; FALSE for transfers from host to device.

The following rules apply when using this data structure:

- pbyData must match lpOutBuffer. If it does not match, ensuing behavior is not guaranteed.
- fTransferDirectionIn must be 0 for a write (data transfer direction = OUT) operation and must be 1 for a read (data transfer direction = IN) operation. Must match direction in bmRequestType. If it does not match, ensuing behavior is not guaranteed.

2.4.6 IOCTL_USBTMC_GET_LAST_ERROR

Gets the last error code returned from the lower-level USB driver.

Parameter Name	Description
hDevice	Device handle, obtained by calling CreateFile.
dwControlCode	IOCTL_USBTMC_GET_LAST_ERROR
lpInBuffer	NULL
nInBufferSize	0
lpOutBuffer	Output data buffer. Pointer to a USBD_STATUS.
nOutBufferSize	Size of output data buffer. Must be **sizeof(USBD_STATUS)**.
lpBytesReturned	Pointer to a location to receive the number of bytes returned.
lpOverlapped	Optional pointer to an OVERLAPPED structure (described in the Windows Platform SDK documentation).

When the DeviceloControl function is called with the IOCTL_USBTMC_GET_LAST_ERROR I/O control code, the caller must specify the address of a USBD_STATUS value as the function's lpOutBuffer parameter. The kernel-mode driver fills in the value with the last error code that it received from USBD.

If the USBTMC kernel driver has never encountered a USBD error, this output value must be 0. The USBTMC kernel driver must not change this value when it returns an error code other than a USBD error. Querying the last error does not cause the USBTMC kernel to reset the cached error code value.

2.5 CloseHandle()

Parameters and behaviors are as specified in the Windows documentation.

2.6 INF files for USBTMC devices

INF files determine the kernel driver associated with a USBTMC device. INF files also determine where USBTMC devices show up in the Windows “Device Manager”.

2.6.1 Class and ClassGUID for USBTMC devices

None of the existing Windows device setup classes apply to USBTMC devices. A USBTMC INF file may define a new device setup class for USBTMC devices. The Class and ClassGUID fields appropriate for USBTMC devices are shown below.

```
[Version]
...
Class=%USBTMC_CLASS%
ClassGUID=%USBTMC_GUID%
...
```

```
[Strings]
USBTMC_CLASS="USBTestAndMeasurementDevice"
USBTMC_GUID="{A9FDBB24-128A-11d5-9961-00108335E361}"
```

2.6.2 INF ClassInstall32 section

An INF file for USBTMC devices may have a [ClassInstall] section to add a class description and a class icon to the registry. An example of INF file content to accomplish this is shown below.

```
…
[ClassInstall32]
AddReg=UsbTmcAddReg
…
[UsbTmcAddReg]
HKR,,,%UsbTmcDevClassName%
HKR,,Icon,,-20
…
[Strings]
UsbTmcDevClassName=”USB Test and Measurement Devices”
…
```

The example above uses icon number -20. This is a standard icon for the Windows device manager for USB devices. It has existed since Windows 98 and works on all current Windows WDM operating systems. A vendor is allowed to use a different icon as long as they provide the Windows resource for it.

2.6.3 INF DDInstall.Interfaces section

An INF file for USBTMC devices may have a [DDInstall.Interfaces] section to add the DeviceClasses\{InterfaceClassGUID} to the registry. {InterfaceClassGUID} is the same as the ClassGUID above. Note that the use of the term DDInstall here is a placeholder for an install section name in the vendor’s INF file.

```
[install-section-name.Interfaces]
AddInterface=%USBTMC_GUID%
```

2.6.4 Product specific INF files

Any vendor may supply product specific INF files for USBTMC device(s). An example of part of such an INF file is shown below.

```
…
[Models]
device-description = install-section-name, USB\Vid_XX&Pid_YY
device-description = install-section-name, USB\Vid_XX&Pid_ZZ
…
```

where

- XX is the idVendor in the device descriptor
- YY is the idProduct in the device descriptor for product #1
- ZZ is the idProduct in the device descriptor for product #2

Any software that installs a product specific INF file must also install all of the necessary files required by the INF file.

2.6.5 Class, SubClass, Protocol specific INF files

Any vendor may supply an INF file generic to a set of USBTMC devices with the same bInterfaceClass, bInterfaceSubClass, and bInterfaceProtocol. This mechanism provides a way for a vendor to override operating system vendor supplied INF files for USBTMC devices. An example of part of such an INF file is shown below.

…

```
[Models]
device-description = install-section-name, USB\Class_XX&SubClass_YY&Prot_ZZ
```

…

where

- XX is the bInterfaceClass in the interface descriptor
- YY is the bInterfaceSubClass in the interface descriptor
- ZZ is the bInterfaceProtocol in the interface descriptor

Any software that installs a class, subclass, and protocol specific INF file must also install all of the necessary files required by the INF file.

2.6.6 Class, SubClass specific INF files and Class specific INF files

Only the operating system vendor is allowed to supply an INF file generic to a set of USBTMC devices with the same bInterfaceClass and bInterfaceSubClass. An example of part of such an INF file is shown below.

…

```
[Models]
device-description = install-section-name, USB\Class_XX&SubClass_YY
```

…

where

- XX is the bInterfaceClass in the interface descriptor
- YY is the bInterfaceSubClass in the interface descriptor

Table 2-3 -- INF file syntax permissions

INF file syntax:	Operating System Vendor (eg. Microsoft)	VISA I/O Library Vendor	Instrument Vendor
Class_##&Subclass_##	Yes	No	No
Class_##&Subclass_##&Prot_##	No	Yes	No
Vid_##	No	No	Yes
Vid_##&Pid_##	No	No	Yes

3. USBTMC include file

The following content summarizes the content that must be placed in a header file associated with the USBTMC kernel driver.

```
#ifndef USBTMC_IOCTL_H
#define USBTMC_IOCTL_H

#include <windows.h>
#include <devioctl.h>

//====================================================================
// The following are the I/O control codes that this driver supports.
#define FILE_DEVICE_USBTMC          0x8000
#define IOCTL_INDEX                 0x0800
#define IOCTL_USBTMC(idx,meth)      CTL_CODE(FILE_DEVICE_USBTMC,(idx),(meth),FILE_ANY_ACCESS)

#define IOCTL_USBTMC_GETINFO        IOCTL_USBTMC(IOCTL_INDEX,   METHOD_BUFFERED) // 0x80002000
#define IOCTL_USBTMC_CANCEL_IO      IOCTL_USBTMC(IOCTL_INDEX+1, METHOD_BUFFERED) // 0x80002004
#define IOCTL_USBTMC_WAIT_INTERRUPT IOCTL_USBTMC(IOCTL_INDEX+2, METHOD_BUFFERED) // 0x80002008
#define IOCTL_USBTMC_RESET_PIPE     IOCTL_USBTMC(IOCTL_INDEX+7, METHOD_BUFFERED) // 0x8000201C
#define IOCTL_USBTMC_SEND_REQUEST   IOCTL_USBTMC(IOCTL_INDEX+32,METHOD_BUFFERED) // 0x80002080
#define IOCTL_USBTMC_GET_LAST_ERROR IOCTL_USBTMC(IOCTL_INDEX+34,METHOD_BUFFERED) // 0x80002088

//====================================================================
// The following are required data structures used for DeviceIoControl.
// Applications will pass data to the driver using these data structures.

// This data structure is used for:
// IOCTL_USBTMC_SEND_REQUEST
typedef struct USBTMC_IO_BLOCK
{
  UCHAR  bmRequestType;
  UCHAR  bRequest;
  USHORT wValue;
  USHORT wIndex;
  USHORT wLength;
  PUCHAR pbyData;              // ignore - use lpOutBuffer   instead - usbscan compatible
  UCHAR  fTransferDirectionIn; // ignore - use bmRequestType instead - usbscan compatible
} USBTMC_IO_BLOCK, *PUSBTMC_IO_BLOCK;

// This data structure is used for:
// IOCTL_USBTMC_CANCEL_IO
// IOCTL_USBTMC_RESET_PIPE
typedef enum USBTMC_PIPE_TYPE
{
  USBTMC_INTERRUPT_IN_PIPE = 1,
  USBTMC_READ_DATA_PIPE    = 2,
  USBTMC_WRITE_DATA_PIPE   = 3,
  USBTMC_ALL_PIPES         = 4
} USBTMC_PIPE_TYPE, *PUSBTMC_PIPE_TYPE;

// This data structure is used for:
// IOCTL_USB_GETINFO
typedef struct USBTMC_DRV_INFO
{
  USHORT major;
  USHORT minor;
  USHORT build;
  WCHAR  manufacturer[64];
} USBTMC_DRV_INFO, *PUSBTMC_DRV_INFO;

//====================================================================
// Class GUID for all USBTMC devices is {A9FDBB24-128A-11D5-9961-00108335E361}
#define USBTMC_CLASS_GUID  (GUID)\
   { 0xa9fdbb24, 0x128a, 0x11d5, { 0x99, 0x61, 0x00, 0x10, 0x83, 0x35, 0xe3, 0x61 } }

#endif
```

4. Available USBD Functions

Not all “Standard Request Codes” listed in Table 9.4 of the Universal Serial Bus Specification are supported by USBD. Following is an excerpt from usbdi.h, the header file that defines function codes and error codes for USBD:

```
#define URB_FUNCTION_SELECT_CONFIGURATION           0x0000
#define URB_FUNCTION_SELECT_INTERFACE               0x0001
#define URB_FUNCTION_ABORT_PIPE                     0x0002
#define URB_FUNCTION_TAKE_FRAME_LENGTH_CONTROL      0x0003
#define URB_FUNCTION_RELEASE_FRAME_LENGTH_CONTROL   0x0004
#define URB_FUNCTION_GET_FRAME_LENGTH               0x0005
#define URB_FUNCTION_SET_FRAME_LENGTH               0x0006
#define URB_FUNCTION_GET_CURRENT_FRAME_NUMBER       0x0007
#define URB_FUNCTION_CONTROL_TRANSFER               0x0008
#define URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER     0x0009
#define URB_FUNCTION_ISOCH_TRANSFER                 0x000A
#define URB_FUNCTION_RESET_PIPE                     0x001E

// These functions correspond to the standard commands on
// the default pipe.  The direction is implied.

#define URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE     0x000B
#define URB_FUNCTION_GET_DESCRIPTOR_FROM_ENDPOINT   0x0024
#define URB_FUNCTION_GET_DESCRIPTOR_FROM_INTERFACE  0x0028

#define URB_FUNCTION_SET_DESCRIPTOR_TO_DEVICE       0x000C
#define URB_FUNCTION_SET_DESCRIPTOR_TO_ENDPOINT     0x0025
#define URB_FUNCTION_SET_DESCRIPTOR_TO_INTERFACE    0x0029

#define URB_FUNCTION_SET_FEATURE_TO_DEVICE          0x000D
#define URB_FUNCTION_SET_FEATURE_TO_INTERFACE       0x000E
#define URB_FUNCTION_SET_FEATURE_TO_ENDPOINT        0x000F
#define URB_FUNCTION_SET_FEATURE_TO_OTHER           0x0023

#define URB_FUNCTION_CLEAR_FEATURE_TO_DEVICE        0x0010
#define URB_FUNCTION_CLEAR_FEATURE_TO_INTERFACE     0x0011
#define URB_FUNCTION_CLEAR_FEATURE_TO_ENDPOINT      0x0012
#define URB_FUNCTION_CLEAR_FEATURE_TO_OTHER         0x0022

#define URB_FUNCTION_GET_STATUS_FROM_DEVICE         0x0013
#define URB_FUNCTION_GET_STATUS_FROM_INTERFACE      0x0014
#define URB_FUNCTION_GET_STATUS_FROM_ENDPOINT       0x0015
#define URB_FUNCTION_GET_STATUS_FROM_OTHER          0x0021

// Direction is specified in TransferFlags.

#define URB_FUNCTION_RESERVED0                      0x0016

// These are for sending vendor and class commands on the
// default pipe.  The direction is specified in TransferFlags.

#define URB_FUNCTION_VENDOR_DEVICE                  0x0017
#define URB_FUNCTION_VENDOR_INTERFACE               0x0018
#define URB_FUNCTION_VENDOR_ENDPOINT                0x0019
#define URB_FUNCTION_VENDOR_OTHER                   0x0020

#define URB_FUNCTION_CLASS_DEVICE                   0x001A
#define URB_FUNCTION_CLASS_INTERFACE                0x001B
#define URB_FUNCTION_CLASS_ENDPOINT                 0x001C
#define URB_FUNCTION_CLASS_OTHER                    0x001F

// Reserved function codes.

#define URB_FUNCTION_RESERVED                       0x001D
#define URB_FUNCTION_GET_CONFIGURATION              0x0026
#define URB_FUNCTION_GET_INTERFACE                  0x0027
#define URB_FUNCTION_LAST                           0x0029
```

Upon mapping Table 9-4 onto the above function codes defined for USBD, the following "Required USBD Function Code Support" table becomes apparent. These mappings of bmRequestType and bmRequest to USB's are required:

Table 4-1: Required USBD Function Code Support

Required USBD Function Code Support			
bmRequestType (with value)	**bRequest (with value)**	**Recipient (with value)**	
Class (0x20)	NA	DEVICE (0)	URB_FUNCTION_CLASS_DEVICE (0x001A)
		INTERFACE (1)	URB_FUNCTION_CLASS_INTERFACE (0x001B)
		ENDPOINT (2)	URB_FUNCTION_CLASS_ENDPOINT (0x001C)
		OTHER (3)	URB_FUNCTION_CLASS_OTHER (0x001F)
Vendor (0x40)	NA	DEVICE (0)	URB_FUNCTION_VENDOR_DEVICE (0x0017)
		INTERFACE (1)	URB_FUNCTION_VENDOR_INTERFACE (0x0018)
		ENDPOINT (2)	URB_FUNCTION_VENDOR_ENDPOINT (0x0019)
		OTHER (3)	URB_FUNCTION_VENDOR_OTHER (0x0020)
Reserved (0x60)	NA	URB_FUNCTION_RESERVED (0x001D)	
Standard (0x00)	GET_STATUS (0)	DEVICE (0)	URB_FUNCTION_GET_STATUS_FROM_DEVICE (0x0013)
		INTERFACE (1)	URB_FUNCTION_GET_STATUS_FROM_INTERFACE (0x0014)
		ENDPOINT (2)	URB_FUNCTION_GET_STATUS_FROM_ENDPOINT (0x0015)
		OTHER (3)	URB_FUNCTION_GET_STATUS_FROM_OTHER (0x0021)
	CLEAR_FEATURE (1)	DEVICE (0)	URB_FUNCTION_CLEAR_FEATURE_TO_DEVICE (0x0010)
		INTERFACE (1)	URB_FUNCTION_CLEAR_FEATURE_TO_INTERFACE (0x0011)
		ENDPOINT (2)	URB_FUNCTION_CLEAR_FEATURE_TO_ENDPOINT (0x0012)
		OTHER (3)	URB_FUNCTION_CLEAR_FEATURE_TO_OTHER (0x0022)
	SET_FEATURE (3)	DEVICE (0)	URB_FUNCTION_SET_FEATURE_TO_DEVICE (0x000D)
		INTERFACE (1)	URB_FUNCTION_SET_FEATURE_TO_INTERFACE (0x000E)
		ENDPOINT (2)	URB_FUNCTION_SET_FEATURE_TO_ENDPOINT (0x000F)
		OTHER (3)	URB_FUNCTION_SET_FEATURE_TO_OTHER (0x0023)
	GET_DESCRIPTOR (6)	DEVICE (0)	URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE (0x000B)
		INTERFACE (1)	URB_FUNCTION_GET_DESCRIPTOR_FROM_INTERFACE (0x0028)
		ENDPOINT (2)	URB_FUNCTION_GET_DESCRIPTOR_FROM_ENDPOINT (0x0024)
		OTHER (3)	None
	SET_DESCRIPTOR (0x07)	DEVICE (0)	URB_FUNCTION_SET_DESCRIPTOR_TO_DEVICE (0x000C)
		INTERFACE (1)	URB_FUNCTION_SET_DESCRIPTOR_TO_INTERFACE (0x0029)
		ENDPOINT (2)	URB_FUNCTION_SET_DESCRIPTOR_TO_ENDPOINT (0x0025)
		OTHER (3)	None
	GET_CONFIGURATION (0x08)	DEVICE (0)	URB_FUNCTION_GET_CONFIGURATION (0x0026)
		INTERFACE (1)	None
		ENDPOINT (2)	None
		OTHER (3)	None

	GET_INTERFACE (0x0A)	DEVICE (0)	URB_FUNCTION_GET_INTERFACE (0x0027)
		INTERFACE (1)	None
		ENDPOINT (2)	None
		OTHER (3)	None

Appendix A: Linux Specific Information

The IVI Foundation is both contributing to and using the USBTMC kernel driver that is part of the Linux kernel open source project.

This appendix is informational only. Because the Linux open source USBTMC kernel driver is not owned by the IVI Foundation, this specification does not specify the driver. However, this section documents changes contributed by the IVI Foundation to the Linux open source USBTMC kernel driver to make sure that IVI members can properly evaluate future proposed changes to the driver.

A.1 History

As part of the initial effort to migrate VISA standards to Linux in 2018, the IVI Foundation agreed that the best USBTMC solution on Linux was to use the existing Linux open source USBTMC kernel driver, if it could be modified to provide essential functionality from the Windows version of the driver. As a result, member companies contributed engineering time to modify and test the driver, and submitted their changes to the Linux kernel open source.

A.2 Added Features

In contrast to the Windows driver, the I/O operations of the initial Linux USBTMC kernel driver already include the USBTMC protocol header. Thus all contributed Linux kernel patches should still support current Linux applications and shall meet the following requirements needed for a VISA Library:

- Synchronous and asynchronous I/O operations
- Vendor specific and generic USB request operations
- Multiple applications can share access to the same instruments
- The driver handles SRQ conflicts
- Simplify definition of device access rules (udev) for USBTMC devices

A.2.1 Changes to ioctl USBTMC488_IOCTL_READ_STB

The ioctl function USBTMC488_IOCTL_READ_STB reads the Status Byte of the connected instrument. The Linux kernel driver supports USB488 interfaces with or without an Interrupt IN endpoint.

This function was modified to return the correct Status Byte with RQS bit set (Bit 6) when the instrument has issued a service request via Interrupt IN endpoint. If more file handles are opened to the same instrument, all file handles will receive the same status byte with RQS bit set. Note that instruments without Interrupt IN endpoint do not support SR1 device capabilities and will just return a Status Byte without RQS bit set.

A.2.2 Support for receiving SRQ notifications via poll/select

In many situations operations on multiple instruments need to be synchronized. The poll() and select() functions provide a convenient way of waiting on a number of different instruments and other peripherals simultaneously. When the instrument sends an SRQ notification the file descriptors (fd) watching for exceptional conditions become readable. To reset the exceptional condition a USBTMC488_IOCTL_READ_STB ioctl must be performed.

With the new asynchronous functions the behavior of the poll function was extended.

- POLLPRI is set when the interrupt pipe receives a status byte with SRQ.
- POLLIN | POLLRDNORM signals that asynchronous URBs are available on IN pipe.
- POLLOUT | POLLWRNORM signals that no URBS are submitted to IN or OUT pipe. It is safe to write.
- POLLERR is set when any submitted URB fails.

Note that POLLERR cannot be masked out. That means waiting only for POLLPRI does not work when asynchronous operations are used. In this case using the next ioctl USBTMC488_IOCTL_WAIT_SRQ is recommended.

A.2.3 New ioctl USBTMC488_IOCTL_WAIT_SRQ

The new ioctl offers an alternative way to wait for a Service Request. In contrast to the poll() function (see above) the ioctl does not return when asynchronous operations fail.

The given parameter of type __u32 specifies the maximum timeout in milliseconds to wait for the next Service Request.

The ioctl returns 0 or -1 with errno set:

- 0 when an SRQ is received
- errno = ETIMEDOUT when timeout (in ms) is elapsed.
- errno = ENODEV when file handle is closed or device disconnected
- errno = EFAULT when device does not have an interrupt pipe.

A.2.4 New ioctl USBTMC488_IOCTL_TRIGGER

This ioctl was added to send a TRIGGER Bulk-OUT header according to the Subclass USB488 Specification.

A.2.5 New ioctls USBTMC_IOCTL_GET/SET_TIMEOUT

These ioctls were added to set/get the I/O timeout in milliseconds for a specific file handle. The I/O timeout is used for write(), read(), and USBTMC488_IOCTL_READ_STB operations.

By default the timeout is set to 5000 milliseconds for compatibility with current Linux applications. VISA implementations should change this timeout to the VISA default of 2000 milliseconds.

USBTMC_IOCTL_SET_TIMEOUT will return with an error EINVAL if timeout is set to less than 100 milliseconds.

A.2.6 New ioctl USBTMC_IOCTL_CTRL_REQUEST

This new ioctl USBTMC_IOCTL_CTRL_REQUEST allows sending arbitrary requests on the control pipe. VISA implementations will use it to implement the VISA API functions: viUsbControlIn/Out.

The given parameter is of type:

```
struct usbtmc_ctrlrequest {
    struct usbtmc_request req;
    void __user *data; /* pointer to data */
} __attribute__ ((packed));
```

where struct usbtmc_request defines the standard setup control request:

```
struct usbtmc_request {
    __u8 bRequestType;
    __u8 bRequest;
    __u16 wValue;
    __u16 wIndex;
    __u16 wLength;
} __attribute__ ((packed));
```

A.2.7 New ioctl USBTMC_IOCTL_EOM_ENABLE

By default the EOM bit is set on the last transfer of a write() operation. This ioctl enables or disables setting the EOM bit for the next write() operation.

Will return with error EINVAL if given parameter eom is not 0 or 1.

A.2.8 New ioctl USBTMC_IOCTL_CONFIG_TERMCHAR

Allows enabling/disabling terminating a read on reception of a termination character. The parameters are passed with the struct:

```
struct usbtmc_termchar {
    __u8 term_char;
    __u8 term_char_enabled;
} __attribute__ ((packed));
```

`This` ioctl controls the field *TermChar* and Bit 1 of field *bmTransferAttributes* of the REQUEST_DEV_DEP_MSG_IN BULK-OUT header. By default *TermCharEnabled* is false and *TermChar* is '\n' (0x0a).

Will return with error EINVAL if *TermCharEnabled* is not 0 or 1 or if attempting to enable *TermCharEnabled* when the device does not support terminating a read when a byte matches the specified *TermChar*.

A.2.9 New ioctl USBTMC_IOCTL_MSG_IN_ATTR

The ioctl function returns the specific field `bmTransferAttributes` of the last DEV_DEP_MSG_IN Bulk-IN header. This header is received by the read() function. The meaning of the (u8) bitmap `bmTransferAttributes` is:

- Bit 0 = EOM flag is set when the last of a USBTMC message is received.
- Bit 1 = Is set when the last byte is a termchar (e.g. '\n').
  Note that this bit is always zero when the device does not support termchar feature or when termchar detection is not enabled (see ioctl USBTMC_IOCTL_CONFIG_TERMCHAR).

A.2.10 New ioctl USBTMC_IOCTL_WRITE

The ioctl function uses the following struct to send generic OUT bulk messages for synchronous and asynchronous write operation:

```
#define USBTMC_FLAG_ASYNC  0x0001
#define USBTMC_FLAG_APPEND 0x0002

struct usbtmc_message {
    __u32 transfer_size; /* size of bytes to transfer */
    __u32 transferred; /* size of received/written bytes */
    __u32 flags; /* bit 0: 0 = synchronous; 1 = asynchronous */
    void __user *message; /* pointer to header and data in user space */
} __attribute__ ((packed));
```

In synchronous mode (flags=0) the generic write function sends the `message` with a size of `transfer_size`. The `message` is split into chunks of 4k (=page size) and submitted (by usb_submit_urb) to the Bulk Out. A semaphore limits the number of flying urbs. The function waits for the end of transmission or returns on error, for example when a single chunk exceeds the timeout. The member `usbtmc_message.transferred` returns the number of transferred bytes.

In asynchronous mode (flags=USBTMC_FLAG_ASYNC) the generic write function is non-blocking. The ioctl clears the current error state and the `internal transfer counter`. The member `usbtmc_message.transferred` returns the number of submitted bytes, however less data can be sent to the device in case of error. The `internal transfer counter` holds the number of total transferred bytes.

With flag USBTMC_FLAG_APPEND additional urbs are submitted without clearing the current error state or internal transfer counter.

The function returns -1 and sets errno = EAGAIN when the semaphore does not allow submitting any urb.

POLLOUT | POLLWRNORM are signaled when all submitted urbs are completed. POLLERR is set when any urb fails. See poll() function above.

A.2.11 New ioctl USBTMC_IOCTL_WRITE_RESULT

The ioctl function copies the current `internal transfer counter` to the given __u32 pointer and returns the current error state of the last (asynchronous) USBTMC_IOCTL_WRITE call. The error state and `internal transfer counter` is not cleared by this ioctl.

A.2.12 New ioctl USBTMC_IOCTL_READ

The ioctl function uses the following struct to get generic IN bulk messages:

```
#define USBTMC_FLAG_ASYNC          0x0001
#define USBTMC_FLAG_IGNORE_TRAILER 0x0004

struct usbtmc_message {
    __u32 transfer_size; /* size of bytes to transfer */
    __u32 transferred; /* size of received/written bytes */
    __u32 flags; /* bit 0: 0 = synchronous; 1 = asynchronous */
    void __user *message; /* pointer to header and data in user space */
} __attribute__ ((packed));
```

In synchronous mode (flags=0) the generic read function copies maximum `transfer_size` bytes of received data from Bulk IN to the `usbtmc_message.message` pointer. Depending on `transfer_size` the read function submits one (<=4kB) or more urbs (up to 16) to Bulk IN. For best performance, the read function copies bytes from one urb to the `message` buffer while other urbs still can receive data from the T&M device concurrently. The function waits for the end of transmission or returns on error or timeout. The member `usbtmc_message.transferred` returns the number of received bytes.

For best performance, the requested transfer size should be a multiple of 4 kB. Please note that the driver has to round down the transfer_size to a multiple of 4 kB when you use more than 4kB, since the driver does not cache or save unread data. The flag USBTMC_FLAG_IGNORE_TRAILER can be used when the transmission size is already known. Then the driver does not round down the transfer_size to a multiple of 4 kB, but does reserve extra space to receive the final short or zero length packet. Note that the instrument may send up to wMaxPacketSize - 1 bytes at the end of a message to avoid sending a zero-length packet.

In asynchronous mode (flags=USBTMC_FLAG_ASYNC) the generic read function is non-blocking. When no received data is available, the read function submits urbs as many as needed to receive `transfer_size` bytes. However, the number of flying urbs (=4kB) is limited to 16 even with subsequent calls of this ioctl.

The message pointer can be NULL when no receiving data shall be returned. The function returns -EAGAIN when no data is available. -EINVAL is returned when data is available but the message pointer is NULL.

When available data is copied to a valid `usbtmc_message.message` pointer the member `usbtmc_message.transferred` returns the number of received bytes.

POLLIN | POLLRDNORM are signaled when at least one urb has completed with received data. POLLOUT | POLLWRNORM are signaled when all submitted urbs IN/OUT are completed. POLLERR is set when any urb fails. See poll() function above.

The function returns:

- 1 when a short or zero length packet is detected.
- 0 is returned when the transferred size is a multiple of wMaxPacketSize
- -1 and errno = EAGAIN: when no data can be read asynchronous.
- -1 and errno = EINVAL: when message pointer is invalid and data could be read.
- -1 and errno = ETIMEDOUT: when no data can be read synchronous (see USBTMC_IOCTL_SET_TIMEOUT) Otherwise the ioctl always returns the very first error of submitted urbs.

A.2.13 New ioctl USBTMC_IOCTL_CANCEL_IO

This ioctl function cancels USBTMC_IOCTL_READ/USBTMC_IOCTL_WRITE functions. Internal error states are set to -ECANCELED. A subsequent call to USBTMC_IOCTL_READ or USBTMC_IOCTL_WRITE_RESULT will return -ECANCELED with information about current transferred data.[1]

A.2.14 New ioctl USBTMC_IOCTL_CLEANUP_IO

This ioctl function kills all submitted urbs to OUT and IN pipe, and clears all received data from IN pipe. The `Internal transfer counters` and error states are reset. An application should use this ioctl after an asynchronous transfer was canceled and/or error handling has finished.

A.2.15 New ioctl USBTMC_IOCTL_AUTO_ABORT

Enable/Disable the auto_abort feature. Auto_abort is disabled by default.

A.2.16 New ioctl USBTMC_IOCTL_API_VERSION

Returns current API version of usbtmc driver.

This is to allow an instrument library to determine whether the driver API is compatible with the implementation.

The API may change in future versions. Therefore, the macro USBTMC_API_VERSION should be incremented when changing tmc.h with new flags, ioctls or when changing a significant behavior of the driver.

A.3 Test Functions

These ioctls are implemented in the intermediate github driver[2] to simulate error conditions while testing. Because these ioctls are not part of the core driver functionality, they are not included in the USBTMC Linux kernel driver code.

[1] For examples on the proper way to use this ioctl, refer to examples on the IVI git repository.

[2] https://github.com/GuidoKiener/linux-usbtmc

A.3.1 USBTMC_IOCTL_SET_OUT_HALT

For testing: This ioctl sends a SET_FEATURE(HALT) request to the OUT endpoint. The ioctl is useful for test purposes, to simulate a device that cannot receive any data due to an error condition.

A.3.2 USBTMC_IOCTL_SET_IN_HALT

For testing: This ioctl sends a SET_FEATURE(HALT) request to the IN endpoint. The ioctl is useful for test purposes to simulate a device that cannot send any data due to an error condition.

A.3.3 USBTMC_IOCTL_ABORT_BULK_IN_TAG

For testing: The ioctl tries to abort a BULK IN transfer with a given tag.

A.3.4 USBTMC_IOCTL_ABORT_BULK_OUT_TAG

For testing: The ioctl tries to abort a BULK OUT transfer with a given tag.