Embedded Systems Project
USBTMC support to the Atmel's UC3 Software Framework

Massimiliano Gentile

September 3, 2010

Contents

1 Introduction

1.1 Purpose

The goal of this project is to add the support to the USB Test and Measurement Class (USBTMC) standard to the Atmel's UC3 software framework. This implementation will be useful for everyone looking at the usage of a UC3 device as a measurement instrument (i.e. low sample rate ADCs or digital signal analyzers).

USBTMC, introduced in 2002, is a vendor-independent standard for programmatic control of USB-based test instruments. The standard defines the protocol that is used to send command messages to an instrument and read back responses but it does not define the format of the messages. Most USB-based instruments available today adheres to the USBTMC standard.

The UC3 Software framework provides software, drivers and libraries to help designing and building any application for AVR32 UC3 devices. The software framework is written in C code and it contains drivers for each AVR32 UC3 peripheral, software libraries optimized for AVR32, hardware components drivers and RTOS-ready source code. Furthermore, high level user documentation including examples, getting started and tutorials are provided. This framework already provides support for many USB classes (i.e. mass storage, audio, DFU, CDC and HID) but it still does not include the support to the USBTMC protocol.

1.2 Organization

This report is organized as follows.

Chapter 2 provides a more detailed analysis of the USBTMC protocol and of the Atmel's Software Framework. Chapter 3 describes the details of the implementation of the USBTMC protocol driver. The document is completed by an appendix with a synthesis of the USBTMC specification for the devices.

1.3 Related Documents

- Universal Serial Bus Specification, Revision 2.0, April 27, 2000, http://www.usb.org
- USB Test and Measurement Class (USBTMC) specification, Revision 1.0, http://www.usb.org
- USB Test and Measurement Class USB488 subclass specification, Revision 1.0, http://www.usb.org
- AVR276: USB Software Library for AT90USBxxx Microcontrollers, http://www.atmel.com

2 Analysis

2.1 USBTMC and USBTMC-USB488

USBTMC is a protocol built on top of USB that allows GPIB-like communication with USB devices. From the user's point of view, the USB device behaves just like a GPIB device. USBTMC allows instrument manufacturers to upgrade the physical layer from GPIB to USB while maintaining software compatibility with existing software, such as instrument drivers and any application that uses Virtual Instrument Software Architectures (VISA).

USBTMC-USB488 is a subclass of the USBTMC protocol that can be used with devices that wish to communicate using messages that are based on the IEEE-488.1 and IEEE-488.2 standards, adding items such as triggering, remote/local signaling, service requests, and communication protocols to properly detect IEEE-488.2 errors such as unterminated and interrupted conditions.
Therefore, USBTMC and its subclass define the protocol for exchanging messages between hosts and devices without defining a new format for the commands.
As it is possible to see in Figure 1, a USBTMC-USB488 device typically contains four endpoints. These endpoints are the default control endpoint, the bulk-out endpoint, the bulk-in endpoint and the interrupt-in endpoint.

The control endpoint is required by the USB 2.0 specification. The Bulk-OUT endpoint is required and is used to provide a high performance, guaranteed delivery data path from the Host to the device. The Host must use the Bulk-OUT endpoint to send USBTMC command messages to the device and to set up all transfers on the Bulk-IN endpoint, and the device must process the USBTMC command messages in

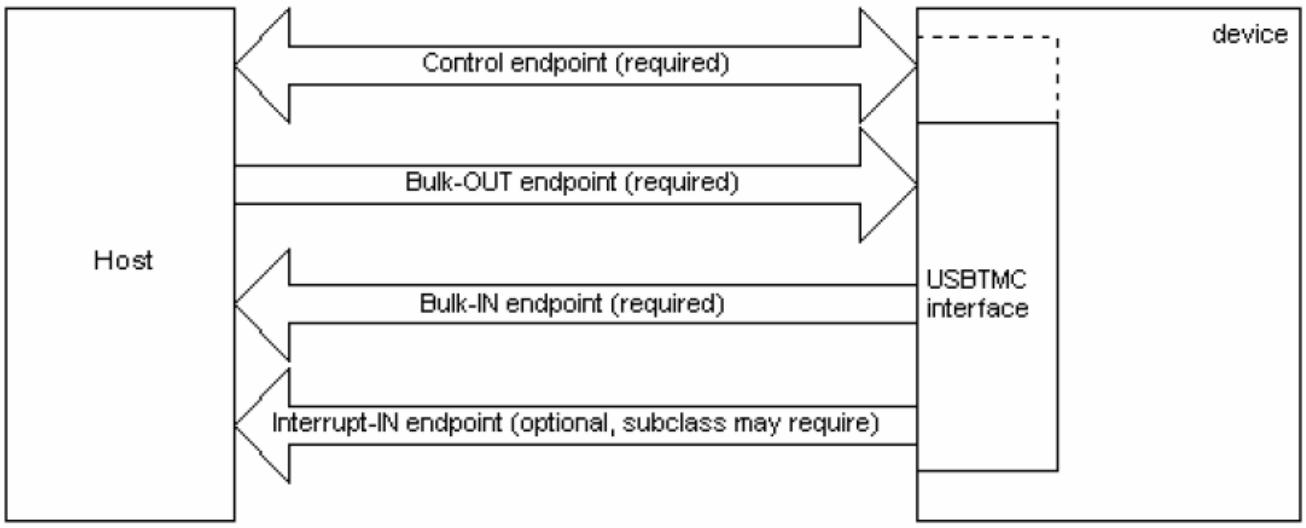

Figure 1: USBTMC communication model

the order they are received.
The Bulk-IN endpoint is required and is used to provide a high performance, guaranteed delivery data path from the device to the Host. The Host must use the Bulk-IN endpoint to receive USBTMC response messages from the device.
The Interrupt-IN endpoint is an optional endpoint and is used by the device to send notifications to the Host. This endpoint can be required by a USBTMC subclass specification.
A list of the messages that are sent across these endpoints is reported in Table 1.

Message	Endpoint	Standard	Description
CHECK_ABORT_BULK_IN_STATUS	Control	USBTMC	Returns the status of the previously sent INITIATE_ABORT_BULK_IN transfer.
CHECK_ABORT_BULK_OUT_STATUS	Control	USBTMC	Returns the status of the previously sent INITIATE_ABORT_BULK_OUT transfer.
CHECK_CLEAR_STATUS	Control	USBTMC	Returns the status of the previously sent INITIATE_CLEAR transfer.
GET_CAPABILITIES	Control	USBTMC	Returns attributes of the USBTMC interface.
INDICATOR_PULSE	Control	USBTMC	(Optional) Pulses the activity indicator light for identification purposes.
INITIATE_ABORT_BULK_IN	Control	USBTMC	Aborts a Bulk-IN transfer.
INITIATE_ABORT_BULK_OUT	Control	USBTMC	Aborts a Bulk-OUT transfer.
INITIATE_CLEAR	Control	USBTMC	Clears the device.
GO_TO_LOCAL	Control	USBTMC-USB488	Equivalent of GTL command byte.
LOCAL_LOCKOUT	Control	USBTMC-USB488	Equivalent of LLO command byte.
READ_STATUS_BYTE	Control	USBTMC-USB488	Reads the IEEE Status byte.
REN_CONTROL	Control	USBTMC-USB488	Simulates REN line.
DEV_DEP_MSG_OUT	Bulk-Out	USBTMC	Send data bytes to the device.
REQUEST_DEV_DEP_MSG_IN	Bulk-Out	USBTMC	Request response from device.
REQUEST_VENDOR_SPECIFIC_IN	Bulk-Out	USBTMC	Request vendor data from device.
VENDOR_SPECIFIC_OUT	Bulk-Out	USBTMC	Send vendor data to device.
TRIGGER	Bulk-Out	USBTMC-USB488	Triggers the device.
DEV_DEP_MSG_IN	Bulk-In	USBTMC	Send data to host.
VENDOR_SPECIFIC_IN	Bulk-In	USBTMC	Send vendor data to host.

Table 1: USBTMC Messages

It is possible to identify the ways used by the USBTMC protocol to emulate the GPIB characteristics. The synchronization is necessary to properly emulate GPIB, which uses a three-wire handshake that has the controller initiate an action by placing a command byte on the bus, with the device indicating the action has been completed by finishing the handshake. As USB doesn't let the device delay completing the handshake until the action is finished, the USBTMC standard converts these out-of-band messages into split transactions to guarantee proper synchronization. The first part of a split transaction corresponds to the GPIB controller placing the data on the bus; it's known as the initiate action. The second part allows the controller to poll the device to determine whether the action has been completed.
One common message used in the Bulk-Out endpoint is DEV_DEP_MSG_OUT (Device-Dependent Message Output). It's the equivalent of a GPIB write from the controller to the device. The message-specific header for this message ID contains a transfer count to indicate the total number of bytes in this transfer,

and an end-of-message bit, which is used to emulate the GPIB EOI (End-or-Identify) signal to indicate whether the last byte of this transfer is the last byte in the total message.
For the bulk-in endpoint the USBTMC protocol uses headers similar to the ones used by the bulk-out endpoint. All bulk-in endpoint transfers start with a bulk-out transfer that contains a message DEV_DEP_MSG_IN (Device-Dependent Message Input). The message-specific header contains a transfer count field to indicate the maximum number of bytes that the device can transfer to host. This guarantees that the device will never transfer more data than asked for by the instrument control application. By doing this, the host doesn't need to buffer any data, and if the device needs to flush its output queue, there will be no data integrity problems.
The purpose of the Interrupt endpoint is to emulate the GPIB service request mechanism. It's optional for devices that don't wish to use service requests. In GPIB, an SRQ (service request) line is used by a device to inform the controller that it requires service. As all traffic in USB is initiated by the host, an interrupt endpoint is used to emulate a system interrupt. However, it isn't a real interrupt; it's actually a high-priority bulk-in endpoint that's polled at a periodic rate. To work around the long first-byte latency of USB, USBTMC devices automatically serial-poll themselves when they need to request service, and return both the SRQ line and the status byte in a single transmission. The advantage is that the host can receive the status byte without generating another USB transaction.

2.2 Atmel's UC3 Software Framework

This framework provides software drivers and libraries to build any application for AVR32 UC3 devices. It has been carefully developed to help develop and glue together the different components of a software in order to be easily integrable into an operating system as well as to operate in a stand-alone way.
The framework is divided into several modules. Each module is provided with full source code, example of usage and a rich HTML documentation. The modules are:

- **UC3 Drivers (directory/DRIVERS)**
  This directory contains software drivers such as ADC, GPIO or Timer peripherals. Each driver is composed of a driver.c and driver.h file that provides low level functions to access the peripheral.
- **Software Services (directory/SERVICES)**
  This directory provides application-oriented piece of software such as a USB mass storage class, a FAT file system and an optimized DSP library.
- **Hardware Components (directory/COMPONENTS)**
  This directory provides software drivers to access hardware components such as external memory or LCD.
- **C/C++ Utilities (directory/UTILS)**
  This directory provides several linker script files and C/C++ files with general usage defines, macros and functions.

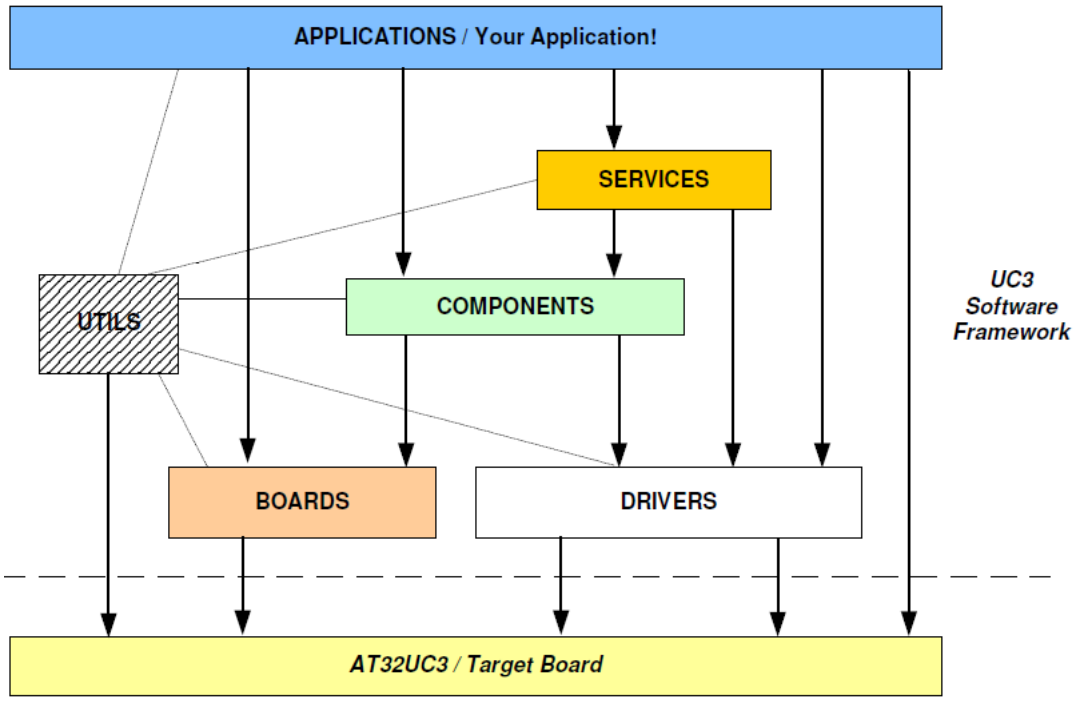

Figure 2: Block Diagram Overview

- **Demo Applications (directory/APPLICATIONS)**
  This directory provides application examples that are based on services, components and drivers modules.

2.2.1 USB Software Library

The USB Software Library is designed to hide the complexity of the USB development. The architecture of the USB firmware is designed to avoid any hardware interfacing (drivers layer should not be modified by the user). The USB software library can manage both device or host enumeration process. The global USB firmware architecture is illustrated in Figure 3 with an example of a dual role sample application.

The source files are organized as follows:

- **lib_mcu/usb/usb_drv.c**
  USB Interface Low Level drivers

- **modules/usb/device_chap9/usb_device_task.c**
  USB device chapter 9 management (connection, disconnection, suspend, resume and call for enumeration process)

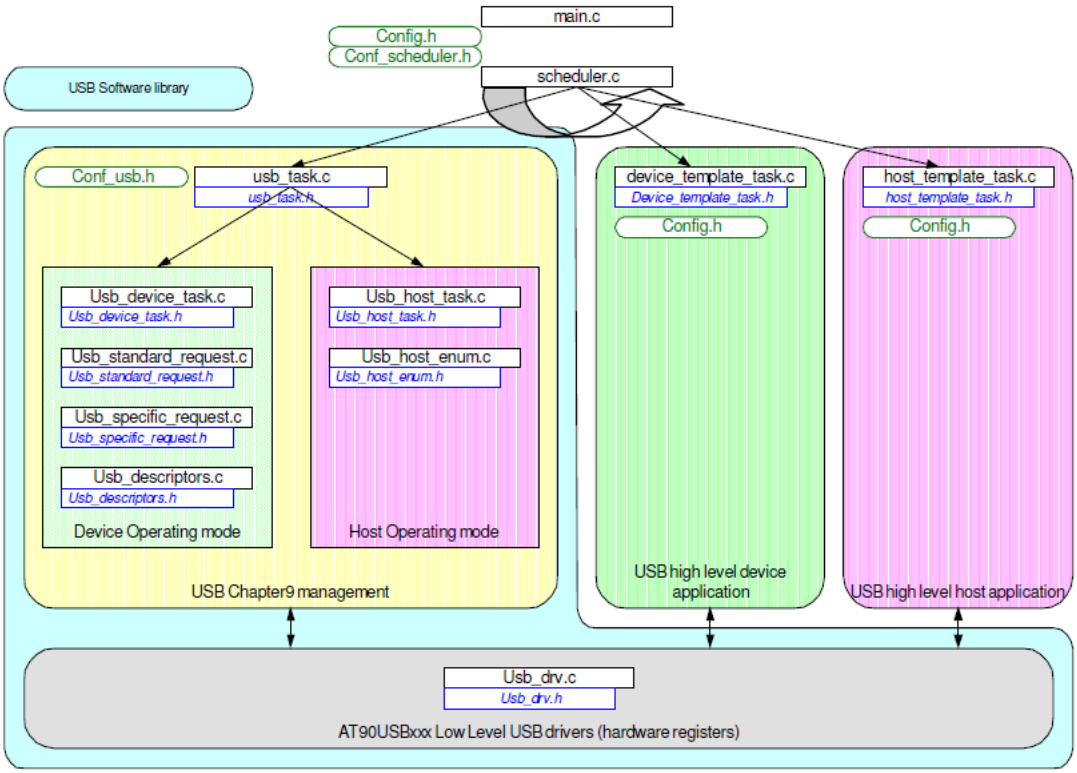

Figure 3: USB Firmware Architecture for a dual role application

- **modules/usb/device_chap9/usb_standard_request.c**
  Device enumeration process
- **modules/usb/host_chap9/usb_host_task.c**
  USB host chapter 9 management (device connection, disconnection, suspend, resume and high level enumeration process)
- **modules/usb/host_chap9/usb_host_enum.c**
  Low level enumeration functions (check VID/PID, configure the host pipe according device descriptors)
- **modules/usb/usb_task.c**
  Entry point for USB task management

An example of a template for a demo application can be organized as follows:

- **demo/template/conf/config.h**
  Global Configuration file for the application
- **demo/template/conf/conf_usb.h**
  Configuration file for the USB software library
- **demo/template/conf/conf_scheduler.h**
  Scheduler configuration (tasks declaration)
- **demo/template/main.c**
  Main entry point (scheduler initialisation)
- **demo/template/usb_specific_request.c**
  User or class specific device enumeration requests (none standard enumeration requests)
- **demo/template/usb_descriptors.c**
  Device descriptors structures (used for device enumeration process)
- **demo/template/device_template_task.c**
  High level user application for USB device mode (sample device application)
- **demo/template/host_template_task.c**
  High level user application for USB host mode (sample host application)

3 Implementation of the USBTMC protocol driver

3.1 Enumeration process

Every USB device communicates its requirements to the host through a process called enumeration. During the enumeration phase, the host asks the device several descriptor values to identify it and load the correct drivers. The device descriptors are transferred to the host which assigns a unique address to the device. Each USB device should have at least this descriptors, in order to be recognized by the host:

- **Device descriptor**
  The USB device can have only one device descriptor. This descriptor describes the entire device. It gives information about the USB version, the maximum packet size of the endpoint 0, the vendor ID, the product ID, the product version, the number of the possible configurations the device can have, etc.
- **Configuration descriptor(s)**
  The USB device can have more than one configuration descriptor, however the majority of devices use a single configuration. This descriptor specifies the power-supply mode (self-powered or bus-powered), the maximum power that can be consumed by the device, the interfaces belonging to the device, the total size of all the data descriptors, etc.
- **Interface descriptor(s)**
  A single device can have more than one interface. The main information given by this descriptor is the number of endpoints used by this interface and the USB class and subclass.
- **Endpoint descriptor(s)**
  This descriptor is used to describe the endpoint parameters such as: the direction (IN or OUT), the transfer type supported (Interrupt, Bulk, Isochronuous), the size of the endpoint, the interval of data transfer in case of interrupt transfer mode, etc.
- **Class Specific descriptor(s)**
- **String descriptor(s)**

3.1.1 Configuring the USB library

To enable the USB device mode of the library, the USB_DEVICE_FEATURE should be defined as ENABLED. The device specific configuration section of "CONFIG/conf_usb.h" file contains the definition of the endpoints used by the device application and a set of user specific actions that can be executed upon special events during the USB communication. For example it is possible to map a function executed upon each USB start of frame event or USB bus reset.
The device descriptors used for the device enumeration process are stored in "SERVICES/USB/CLASS/USBTMC/ENUM/usb_descriptors.c" and "SERVICES/USB/CLASS/USBTMC/ENUM/usb_descriptors.h"

files. The descriptors structures are declared in usb_descriptors.h file, so all the enumeration parameters for the configuration of the device should be declared in this file. All these parameters are used to fill up the descriptor fields declared in usb_descriptors.c file. In this phase it's not necessary to implement any method or to know the USB enumeration process more precisely, since all the functions that handle this process are provided by the USB library of the Atmel's UC3 Software Framework.

When the host controller performs the enumeration process, its requests are decoded thanks to the standard_request.c enumeration functions and the user defined descriptors are sent to the host controller. The device user application task knows that the device is properly enumerated thanks to the Is_device_enumerated() function that returns TRUE once the SET_CONFIGURATION request has been received from the host.
An example of what can be configured in the usb_descriptors.c is shown in Listing 1. In this case it is possible to see the declaration of the vendor and product IDs and the definition of the Bulk-OUT endpoint.

```
#define VENDOR_ID               ATMEL_VID
#define PRODUCT_ID              USBTMC_PID
#define NB_CONFIGURATION        1
#define NB_INTERFACE            1
#define NB_ENDPOINT             3
...
            // USB Endpoint 1 descriptor
            // Bulk OUT
#define ENDPOINT_NB_1           BULKOUT_EP
#define EP_ATTRIBUTES_1         TYPE_BULK
#define EP_IN_LENGTH_1_FS       0x40
#define EP_IN_LENGTH_1_HS       512
#define EP_SIZE_1_FS            EP_IN_LENGTH_1_FS
#define EP_SIZE_1_HS            EP_IN_LENGTH_1_HS
#define EP_INTERVAL_1           0x00
```

Listing 1: Example of code from usb_descriptors.h

3.2 Specific Requests

The USBTMC protocol defines a set of class specific requests for commands sent through the Control endpoint, as it is possible to see in Table 1. When the USB library receives an unsupported USB request, the method usb_user_read_request in the file "usb_specific_request.c" is invoked. Then it is possible to identify if the request is class specific with a switch-case statement. The usb_user_read_request method returns TRUE when the request is processed, FALSE if it's not supported and in this case a STALL

```
Bool usb_user_read_request(U8 type, U8 request)
{
  switch (request)
  {
    case USBTMC_GET_CAPABILITIES:
        usbtmc_get_capabilities();
        return TRUE;
    ...
    default:
        return FALSE;
  }
}
```

Listing 2: Example of code in method usb_user_read_request

handshake will be automatically sent by the standard USB read request function.

3.2.1 GET_CAPABILITIES

This is an example of a specific request that has been implemented. This request is used to provide additional attributes and capabilities of a USBTMC interface. The method is implemented in "SERVICES/USB/CLASS/USBTMC/ENUM/usb_specific_request.c" as usbtmc_get_capabilities(). After having acknowledged the Setup packet and reset the queue of the Control endpoint, the attributes and capabilities are sent according to the format specified in the USBTMC specification. In Listing 3 it is shown the implementation of the function. It is possible to notice that the device will inform the host of its capability of accepting the INDICATOR_PULSE request and supporting the TermChar character to end a Bulk-IN transfer.

3.3 Interface Endpoints

In order to let the USBTMC device communicate with the host, a set of helper commands has been implemented with the purpose of a better ease of use of the device driver. These methods are defined in "SERVICES/USB/CLASS/USBTMC/device_usbtmc_helper.c"

3.3.1 Helper Functions

The first function that has been implemented is named usb_read. It is necessary to provide a pointer to a struct S_usbtmc_bulk_out_header that will contain the header received from the host and a pointer to an array of bytes that will contain the data part of the message. The size of the data message can be read in the transferSize value of the header. First, the header bytes are read and, based on the message

```
void usbtmc_get_capabilities(void)
{
        Usb_ack_setup_received_free();
        Usb_reset_endpoint_fifo_access(EP_CONTROL);
        Usb_write_endpoint_data(EP_CONTROL, 8, USBTMC_STATUS_SUCCESS);
        Usb_write_endpoint_data(EP_CONTROL, 8, 0);
        Usb_write_endpoint_data(EP_CONTROL, 16, bcdUSBTMC);
        Usb_write_endpoint_data(EP_CONTROL, 8, USBTMC_ACCEPT_INDICATOR_PULSE);
        Usb_write_endpoint_data(EP_CONTROL, 8, USBTMC_ACCEPT_TERMCHAR);
        Usb_write_endpoint_data(EP_CONTROL, 64, 0);
        Usb_write_endpoint_data(EP_CONTROL, 64, 0);
        Usb_write_endpoint_data(EP_CONTROL, 16, 0);
        ...
}
```

Listing 3: Example of code to implement the GET_CAPABILITIES class specific request

```
#define NB_MS_BEFORE_FLUSH                          10
void usb_read(S_usbtmc_bulk_out_header* header, U32* data_rx);
void usb_write(S_usbtmc_bulk_out_header header_bulkout, U32* data_to_send,
               int size);
void usb_timeout_flush (void);
Bool usb_test_hit(void);
```

Listing 4: Exposed interfaces for helper functions

ID, the S_usbtmc_bulk_out_header is filled. Then if the message is a DEV_DEP_MSG_OUT or a VENDOR_SPECIFIC_OUT, the data part is read with the help of the usb_read_data function. In this simple version I don't take into account the possibility to send messages over different transactions. This is an acceptable assumption considering that the typical Test&Measurement messages are short. With a more complex implementation of the message exchange protocol, it may be necessary to reconsider this assumption. The second exported helper function is named usb_write, and is used to write responses to request messages. This is the case of REQUEST_DEV_DEP_MSG_IN and REQUEST_VENDOR_SPECIFIC_IN. Based on the type of request receive, the appropriate header is sent to the host followed by a data message that must be passed to this method as a parameter, specifying even its size in bytes. The data message is terminated by the TermChar character received in the request header.
Another function that can be used is the usb_flush_timeout. It is useful to flush the Bulk IN endpoint buffer every NB_MS_BEFORE_FLUSH milliseconds, a parameter that can be specified in the header file "device_usbtmc_helper.h". To conclude, it is possible to use the function usb_test_hit to test whether some data was received in the Bulk OUT endpoint.

3.4 Sample Application

To simplify the development of an extension to this driver, in order to generate a message exchange protocol that permits the management of the USBTMC device, it has been implemented a stub application in the file device_usbtmc.c. This application executes a function in an infinite loop. In Listing 5 it is possible to see the core of the function. First, the function waits until it receives a message. Then it decodes the

```
usb_read(&header_bulkout, buffer);
switch (header_bulkout.msgID) {
        case USBTMC_DEV_DEP_MSG_OUT:
        case USBTMC_VENDOR_SPECIFIC_OUT:
        {
                process_input_message(header_bulkout, buffer);
                break;
        }
        case USBTMC_REQUEST_DEV_DEP_MSG_IN:
        case USBTMC_REQUEST_VENDOR_SPECIFIC_IN:
        {
                U32* response = elaborate_response(header_bulkout, buffer);
                usb_write(header_bulkout, response, sizeof(response));
                break;
        }
}
```

Listing 5: A sample of code from the stub application

message ID of the header. If it is received an output message, device dependent or vendor specific, the data content, saved in the variable buffer, is processed. When a request message is received, the device elaborates a response and write it as the answer. In the case of this stub application, the input message is just memorized and the elaborate_response method simply returns the same data content saved in the last output message. To extend and improve the possibility to exchange messages, it is necessary to modify the implementation of the methods process_input_message and elaborate_response.

A Appendix A - USBTMC specification

The USBTMC specification specifies the shared attributes, common services, and data formats for devices with a USBTMC compliant test and measurement interface. This specification addresses the common specification needs that apply to minimal devices (i.e. ADCs, sensors, and transducers), devices that communicate with IEEE 488 messages or devices with sub-addressable components (i.e. mainframes with instrument cards).

A.1 Descriptors

This is the list of Descriptor types as specified in table 9-5 of the USB 2.0 specification.

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER	8

Table 2: Descriptor Types

A.1.1 Device Descriptor

A device descriptor describes general information about a USB device. It includes information that applies globally to the device and all of the device's configurations. A USB device has only one device descriptor.

A.1.2 Device_Qualifier Descriptor

The device_qualifier descriptor describes information about a high-speed capable device that would change if the device were operating at the other speed. For example, if the device is currently operating at full-speed, the device_qualifier returns information about how it would operate at high-speed and vice-versa.

A.1.3 Configuration Descriptor

The configuration descriptor describes information about a specific device configuration. The descriptor describes the number of interfaces provided by the configuration. Each interface may operate independently. A USB device has one or more configuration descriptors. Each configuration has one or more interfaces and each interface has zero or more endpoints. An endpoint is not shared among interfaces within a single

Offset	Field	Size	Value	Description
0	bLength	1	0x12	Size of this descriptor in bytes.
1	bDescriptorType	1	0x01	DEVICE Descriptor Type as specified in Table 2.
2	bcdUSB	2	BCD (0x0200 or greater)	Binary coded decimal field indicating the USB specification level used in the design of this device. As specified in USB 2.0 specification, section 9.6.1.
4	bDeviceClass	1	0x00	Class found in interface descriptor.
5	bDeviceSubClass	1	0x00	Subclass found in interface descriptor.
6	bDeviceProtocol	1	0x00	Protocol found in interface descriptor.
7	bMaxPacketSize0	1	Number	Maximum packet size for endpoint zero (only 8, 16, 32, or 64 are valid). If the device is operating at high-speed, this field must be 64 indicating a 64 byte maximum packet.
8	idVendor	2	ID	Required. Vendor ID assigned by USB-IF
10	idProduct	2	ID	Required. Product ID assigned by the manufacturer
12	bcdDevice	2	BCD	Device release number in binary coded decimal.
14	iManufacturer	1	Index	Index of string descriptor describing manufacturer. Required to be non-zero. Specified in USB 2.0 specification, section 9.6.1. The bLength for the iManufacturer string descriptor must be ¿= 4 and ¡= 128 (1 ¡= number of Unicode characters ¡= 63).
15	iProduct	1	Index	Index of string descriptor describing product. Required to be non-zero. Specified in USB 2.0 specification, section 9.6.1 and section 5.7 of this USBTMC document. The bLength for the iProduct string descriptor must be ¿= 4 and ¡= 128 (1 ¡= number of Unicode characters ¡= 63).
16	iSerialNumber	1	Index	Index of string descriptor describing the devices serial number. Required to be non-zero. Specified in USB 2.0 specification, section 9.6.1. The combination of idVendor, idProduct, and iSerialNumber must be unique for every instance of a device. The bLength for the iSerialNumber string descriptor must be ¿= 4 and ¡= 128 (1 ¡= number of Unicode characters ¡= 63).
17	bNumConfigurations	1	Number	Number of possible configurations at the current operating speed.

Table 3: Device Descriptor

configuration unless the endpoint is used by alternate settings of the same interface. Endpoints may be shared among interfaces that are part of different configurations without this restriction. Once configured, devices may support limited adjustments to the configuration.

Offset	Field	Size	Value	Description
0	bLength	1	0x0A	Size of this descriptor in bytes.
1	bDescriptorType	1	0x06	DEVICE QUALIFIER Descriptor Type as specified in Table 2.
2	bcdUSB	2	BCD	USB specification version number (e.g., 0200H for V2.00).
4	bDeviceClass	1	0x00	Class found in interface descriptor.
5	bDeviceSubClass	1	0x00	Subclass found in interface descriptor.
6	bDeviceProtocol	1	0x00	Protocol found in interface descriptor.
7	bMaxPacketSize0	1	Number	Maximum packet size for other speed.
8	bNumConfigurations	1	Number	Number of Other-speed Configurations.
9	Reserved	1	0x00	Reserved for future use, must be zero

Table 4: Device_Qualifier Descriptor

Offset	Field	Size	Value	Description
0	bLength	1	0x09	Size of this descriptor in bytes.
1	bDescriptorType	1	0x02	CONFIGURATION Descriptor Type as specified in Table 2.
2	wTotalLength	2	Number	Total length of data returned for this configuration. Includes the combined length of all descriptors (configuration, interface, endpoint, and class- or vendor-specific returned for this configuration.
4	bNumInterfaces	1	Number	Number of interfaces supported by this configuration. A device may have multiple USBTMC interfaces.
5	bConfigurationValue	1	Number	Value to use as an argument to the SetConfiguration() request to select this configuration.
6	iConfiguration	1	Index	Index of string describing this configuration.
7	bmAttributes	1	Bitmap	Configuration characteristics D7: Reserved (set to one) D6: Self-powered D5: Remote Wakeup D4..0: Reserved (reset to zero)
8	bMaxPower	1	mA	Maximum power consumption of the USB device from the bus in this specific configuration when the device is fully operational. It is expressed in 2 mA units (i.e., 50 = 100 mA).

Table 5: Configuration Descriptor

A.1.4 Other_Speed_Configuration Descriptor

The other_speed_configuration descriptor describes a configuration of a high-speed capable device if it were operating at its other possible speed. The structure of the other_speed_configuration is identical to a configuration descriptor, except for the bDescriptorType field that has a value of 0x07 as specified in Table 2.

A.1.5 Interface Descriptor

The interface descriptor describes a specific interface withing a configuration. A configuration provides one or more interfaces, each with zero or more endpoint descriptors describing a unique set of endpoints within the configuration. An interface descriptor is always returned as part of a configuration descriptor. An interface may include alternate settings that allow the endpoints and/or their characteristics to be varied after the device has been configured. Alternate settings allow a portion of the device configuration to be varied while other interfaces remain in operation. If a configuration has alternate settings for one or more of its interfaces, a separate interface descriptor and its associated endpoints are included for each setting. A

Offset	Field	Size	Value	Description
0	bLength	1	0x09	Size of this descriptor in bytes.
1	bDescriptorType	1	0x04	INTERFACE Descriptor Type as specified in Table 2.
2	bInterfaceNumber	1	Number	Number of this interface. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration.
3	bAlternateSetting	1	0x00	Value used to select this alternate setting for the interface identified in the prior field.
4	bNumEndpoints	1	Number	Number of endpoints used by this interface (excluding endpoint zero). If this value is zero, this interface only uses the Default Control Pipe.
5	bInterfaceClass	1	Class = 0xFE	Application-Class class code, assigned by USB-IF. The Host must not load a USBTMC driver based on just the bInterfaceClass field.
6	bInterfaceSubClass	1	0x03	Subclass code, assigned by USB-IF.
7	bInterfaceProtocol	1	Protocol	Protocol code 0: USBTMC interface. No subclass specification applies. 1: USBTMC USB488 interface. 2-127: Reserved.
8	iInterface	1	Index	Index of string descriptor describing this interface.

Table 6: Interface Descriptor

USBTMC interface with a bInterfaceProtocol = 0x00 must have exactly one Bulk-OUT endpoint, exactly one Bulk-IN endpoint, and may have at most one Interrupt-IN endpoint. Additional endpoints must be placed in another interface.

A.1.6 Endpoint Descriptors

For USBTMC interfaces it's necessary to specify at least the Bulk-IN Endpoint and the Bulk-OUT endpoint. Following the USBTMC USB488 subclass specification it is also necessary to specify an Interrupt-IN Endpoint.
Each endpoint used for an interface has its own descriptor. This descriptor contains the information re-

quired by the host to determine the bandwidth requirements of each endpoint. There is never an endpoint descriptor for endpoint zero.

A.1.7 String Descriptors

String descriptors use UNICODE encodings as defined by The Unicode Standard, Worldwide Character Encoding, Versione 3.0, The Unicode Consortium, Addison-Wesley Publishing Company, Reading, Massachusetts. The strings in a USB device may support multiple languages. When requesting a string descriptor, the requester specifies the desired language using a sixteen-bit language ID (LANGID) defined by the USB-IF. All devices with a USBTMC interface must implement at least string descriptors with LANGID = 0x0409 (English, United States). The format of string descriptors are as specified in the USB 2.0 specification, section 9.6.7.

A.2 Interface Endpoints and Characteristics

A.2.1 Default control endpoint

The default control endpoint must support control transfers as required in the USB 2.0 specification. The default control endpoint is used to send standard, class and vendor-specific requests to the device, interface, or endpoint. The default control endpoint number must be 0.

A.2.2 Bulk-OUT endpoint

The Host uses the Bulk-OUT endpoint to send USBTMC command messages to the device. For all Bulk-OUT USBTMC command messages the Host must begin the first USB transaction in each Bulk-OUT transfer of command message content with a Bulk-OUT Header.
The following rules apply to all Bulk-OUT USBTMC command messages:

1. The Host must send the USBTMC message data bytes (if applicable) immediately after the USBTMC Bulk-OUT Header in the same USB transaction in the same DATA payload, subject to maximum packet size constraints.
2. The total number of bytes in each Bulk-OUT transaction must be a multiple of 4. The Host must add 0 to a maximum of 3 extra alignment bytes to the last transaction payload to achieve 4-byte (32-bit) alignment. The alignment bytes should be 0x00-valued, but this is not required.
3. The Host must not send a new USBTMC Bulk-OUT Header if a previous Bulk-OUT transfer has not yet completed.
4. The Host must consider a Bulk-OUT data transfer complete when it has transferred exactly the amount of data expected (all of the message data bytes and alignment bytes). If the last data payload is wMaxPacketSize, the Host should not send a zero-length packet. The device must consider the

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	0x05	ENDPOINT Descriptor Type as specified in Table 2.
2	bEndpointAddress	1	Endpoint	The address of the endpoint on the USB device described by this descriptor, encoded as follows: Bit 3..0: The endpoint number Bit 6..4: Reserved, reset to zero Bit 7: Direction, 0 = OUT endpoint 1 = IN endpoint
3	bmAttributes	1	Bitmap	This field describes the endpoint's attributes when it is configured using the bConfigurationValue. Bits 1..0: Transfer Type, 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt Since no isochronous endpoints are used in USBTMC, all other bits must be set to zero.
4	wMaxPacketSize	2	Number	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. Bits 10..0 specify the maximum packet size (in bytes). For the Bulk-OUT endpoint the maximum packet size (in bytes) must be a multiple of 4. For high speed interrupt endpoints: Bits 12..11: specify the number of additional transaction opportunities per microframe, 00 = None 01 = 1 additional 10 = 2 additional 11 = Reserved All other bits must be set to zero.
6	bInterval	1	Number	Interval for polling endpoint for data transfers. Expressed in frames or microframes depending on the device operating speed. For full-speed or low-speed interrupt endpoints, the value of this field may be from 1 to 255. For high-speed interrupt endpoints this value must be in the range from 1 to 16, and it is used as the exponent for a $2^{bInterval-1}$ value. For high-speed bulk OUT endpoints, the bInterval must specify the maximum NAK rate of the endpoint. A value of 0 indicates the endpoint never NAKs. Other values indicate at most 1 NAK each bInterval number of microframes. This value must be in the range from 0 to 255.

Table 7: Endpoint Descriptor

transfer complete when it has received and processed exactly the amount of data expected or the device received and processed a packet with payload size less than wMaxPacketSize. See the USB 2.0 specification, section 5.8.3.

5. The Host must send a complete USBTMC command message with a single transfer. This is illustrated below in Figure 4. If the Host fails to do so, the device must Halt the Bulk-OUT endpoint. The only exception is if, in the specification of a particular USBTMC command message, explicit permission is given to send the command message with multiple transfers.

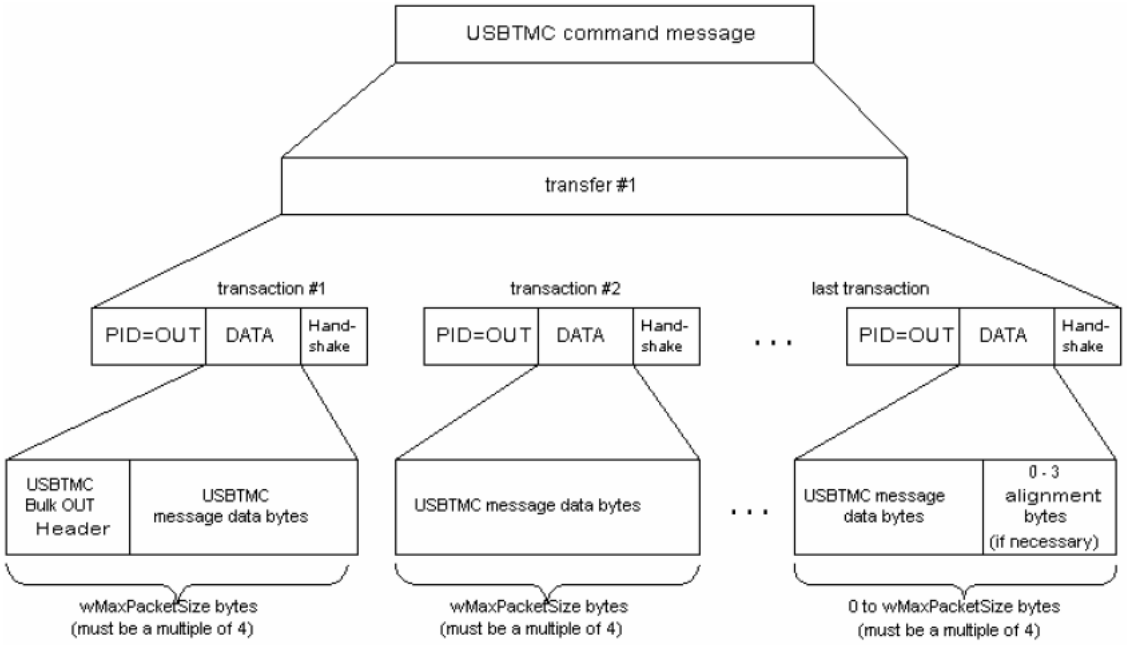

Figure 4: Bulk-OUT USBTMC message sent with a single transfer

Bulk-OUT USBTMC command DEV_DEP_MSG_OUT The Host uses DEV_DEP_MSG_OUT to identify a transfer that sends a USBTMC device dependent command message from the Host to a device. The Bulk-OUT Header command specific content for this command is shown below in Table 8.

The Host may send this USBTMC command message with multiple transfers, as the data becomes available. This is illustrated below in Figure 5. This ability is needed because Host applications may not send a complete message all at once. Another benefit of this ability is that some devices may make use of USBTMC message content as it is delivered.

Offset	Field	Size	Value	Description
0	MsgID	1	0x01	Specifies this specific USBTMC message.
1	bTag	1	Value	A transfer identifier. The Host must set bTag different than the bTag used in the previous Bulk-OUT Header. The Host should increment the bTag by 1 each time it sends a new Bulk-OUT Header. The Host must set bTag such that $1 <= bTag <= 255$.
2	bTagInverse	1	Value	The inverse (ones complement) of the bTag. For example, the bTagInverse of 0x5B is 0xA4.
3	Reserved	1	0x00	Reserved. Must be 0x00.
4-7	TransferSize	4	Number	Total number of USBTMC message data bytes to be sent in this USB transfer. This does not include the number of bytes in this Bulk-OUT Header or alignment bytes. Sent least significant byte first, most significant byte last. TransferSize must be ¿ 0x00000000.
8	bmTransfer Attributes	1	Bitmap	D7..D1 Reserved. All bits must be 0. D0 EOM. 1 - The last USBTMC message data byte in the transfer is the last byte of the USBTMC message. 0 The last USBTMC message data byte in the transfer is not the last byte of the USBTMC message.
9-11	Reserved	3	0x000000	Reserved. Must be 0x000000.

Table 8: DEV_DEP_MSG_OUT Bulk-OUT Header

Bulk-OUT USBTMC command REQUEST_DEV_DEP_MSG_IN The Host uses REQUEST_DEV_DEP_M to identify the transfer as a USBTMC command message to the device, allowing the device to send a USBTMC response message containing device dependent message data bytes.
The REQUEST_DEV_DEP_MSG_IN Bulk-OUT Header and command specific content is shown below in Table 9.

Bulk-OUT USBTMC command VENDOR_SPECIFIC_OUT The Host uses VENDOR_SPECIFIC_OUT to identify a transfer that sends a USBTMC vendor specific command message from the Host to a device. The Bulk-OUT Header command specific content for this command is shown below in Table 10. The Host may send this USBTMC command message with multiple transfers, as the data becomes available. This is illustrated in Figure 3. This ability is needed because Host applications may not send a complete message all at once. Another benefit of this ability is that some devices may make use of USBTMC message content as it is delivered.

Bulk-OUT USBTMC command REQUEST_VENDOR_SPECIFIC_IN The Host uses REQUEST_VENDO to identify the transfer as a USBTMC command message to the device, allowing the device to send a USBTMC response message containing vendor specific message data bytes.

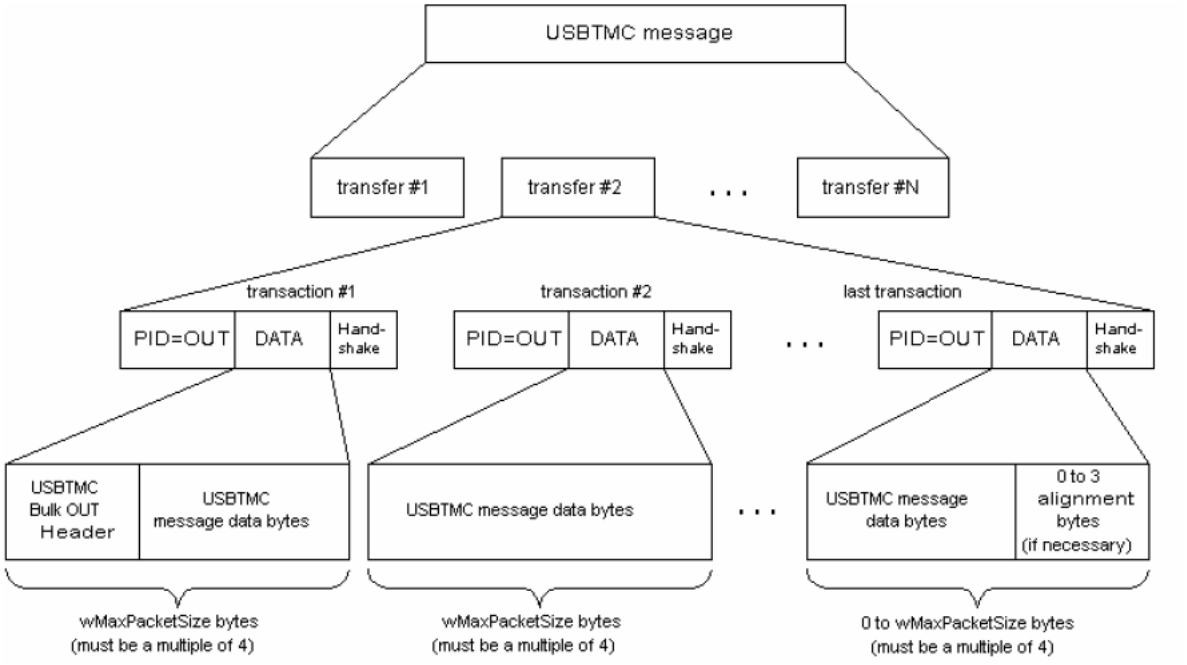

Figure 5: Bulk-OUT USBTMC message sent with multiple transfers

The REQUEST_VENDOR_SPECIFIC_IN Bulk-OUT Header and command specific content is shown below in Table 11.

A.2.3 Bulk-IN endpoint

The Host uses the Bulk-IN endpoint to read USBTMC response messages from the device. For all Bulk-IN USBTMC response messages, whether defined in this specification, a USBTMC subclass specification, or some other specification, the device must begin the first USB transaction in each Bulk-IN transfer of USBTMC response message content with a Bulk-IN Header.
The following rules apply to Bulk-IN USBTMC response messages:

1. USBTMC client software must queue a request to the USB Host Controller to send a USBTMC command message that expects a response before queuing a request to the USB Host Controller that will result in Bulk-IN requests being sent to the USBTMC interface.

2. If a USBTMC interface receives a Bulk-IN request prior to receiving a USBTMC command message that expects a response, the device must NAK the request.

Offset	Field	Size	Value	Description
0	MsgID	1	0x02	Specifies this specific USBTMC message.
1	bTag	1	Value	A transfer identifier. The Host must set bTag different than the bTag used in the previous Bulk-OUT Header. The Host should increment the bTag by 1 each time it sends a new Bulk-OUT Header. The Host must set bTag such that $1 <= bTag <= 255$.
2	bTagInverse	1	Value	The inverse (ones complement) of the bTag. For example, the bTagInverse of 0x5B is 0xA4.
3	Reserved	1	0x00	Reserved. Must be 0x00.
4-7	TransferSize	4	Number	Maximum number of USBTMC message data bytes to be sent in response to the command. This does not include the number of bytes in this Bulk-IN Header or alignment bytes. Sent least significant byte first, most significant byte last. TransferSize must be ¿ 0x00000000.
8	bmTransfer Attributes	1	Bitmap	D7..D2 Reserved. All bits must be 0.
				D1 TermCharEnabled. 1 The Bulk-IN transfer must terminate on the specified TermChar. The Host may only set this bit if the USBTMC interface indicates it supports TermChar in the GET_CAPABILITIES response packet. 0 The device must ignore TermChar.
				D0 Must be 0.
9	TermChar	1	Value	If bmTransferAttributes.D1 = 1, TermChar is an 8-bit value representing a termination character. If supported, the device must terminate the Bulk-IN transfer after this character is sent. If bmTransferAttributes.D1 = 0, the device must ignore this field.
10-11	Reserved	2	0x0000	Reserved. Must be 0x0000.

Table 9: REQUEST_DEV_DEP_MSG_IN Bulk-OUT Header

3. The device must not queue any Bulk-IN DATA until it receives a valid USBTMC command message that expects a response.

4. The Host must consider the Bulk-IN transfer to be in progress once the transaction containing the Bulk-OUT Header for the USBTMC command message has been ACKd.

5. The device must consider the Bulk-IN transfer to be in progress when the device parses the MsgID of a valid USBTMC command message that expects a response.

6. The device is not required to respond immediately after receiving a USBTMC command message that expects a response. A device must not send a DATA payload until a termination condition is detected (EOM, TermChar, or the maximum number of USBTMC response message data bytes the Host has specified to send are available) or until the device can not buffer any more data.

Offset	Field	Size	Value	Description
0	MsgID	1	0x7E	Specifies this specific USBTMC message.
1	bTag	1	Value	A transfer identifier. The Host must set bTag different than the bTag used in the previous Bulk-OUT Header. The Host should increment the bTag by 1 each time it sends a new Bulk-OUT Header. The Host must set bTag such that 1 <= $bTag$ <= 255.
2	bTagInverse	1	Value	The inverse (ones complement) of the bTag. For example, the bTagInverse of 0x5B is 0xA4.
3	Reserved	1	0x00	Reserved. Must be 0x00.
4-7	TransferSize	4	Number	Total number of USBTMC message data bytes to be sent in this USB transfer. This does not include the number of bytes in this Bulk-OUT Header or alignment bytes. Sent least significant byte first, most significant byte last. TransferSize must be ¿ 0x00000000.
8-11	Reserved	4	0x00000000	Reserved. Must be 0x0000000.

Table 10: VENDOR_SPECIFIC_OUT Bulk-OUT Header

Offset	Field	Size	Value	Description
0	MsgID	1	0x7F	Specifies this specific USBTMC message.
1	bTag	1	Value	A transfer identifier. The Host must set bTag different than the bTag used in the previous Bulk-OUT Header. The Host should increment the bTag by 1 each time it sends a new Bulk-OUT Header. The Host must set bTag such that 1 <= $bTag$ <= 255.
2	bTagInverse	1	Value	The inverse (ones complement) of the bTag. For example, the bTagInverse of 0x5B is 0xA4.
3	Reserved	1	0x00	Reserved. Must be 0x00.
4-7	TransferSize	4	Number	Maximum number of USBTMC message data bytes to be sent in response to the command. This does not include the number of bytes in this Bulk-IN Header or alignment bytes. Sent least significant byte first, most significant byte last. TransferSize must be ¿ 0x00000000.
8-11	Reserved	4	0x00000000	Reserved. Must be 0x0000000.

Table 11: REQUEST_VENDOR_SPECIFIC_IN Bulk-OUT Header

7. The first USB transaction in a Bulk-IN transfer must begin with a complete Bulk-IN Header.

8. The USBTMC message data bytes must immediately follow the USBTMC Bulk-IN Header in the same USB transaction in the same DATA payload, subject to maximum packet size constraints.

9. A device may return less than the maximum number of USBTMC response message data bytes the Host specified to send. When the Bulk-IN transfer is completed, if more message data bytes are expected, the Host may send a new USBTMC command message to read the remainder of the

message.

10. The device must always terminate a Bulk-IN transfer by sending a short packet. The short packet may be zero-length or non zero-length. The device may send extra alignment bytes (up to wMaxPacketSize 1) to avoid sending a zero-length packet. The alignment bytes should be 0x00- valued, but this is not required. A device is not required to send any alignment bytes.

11. Once a transfer is terminated, the device must not queue any more Bulk-IN DATA until it receives another USBTMC command message that expects a response.

12. A device may defer the parsing and processing of Bulk-OUT data while a Bulk-IN transfer is in progress.

13. The device may send a Bulk-IN message using multiple transfers, as the data becomes available. This is illustrated below in Figure 6. This ability is needed because some devices may not have enough memory to buffer a complete USBTMC message. Another benefit of this ability is that some Hosts may make use of USBTMC message content as it is delivered.

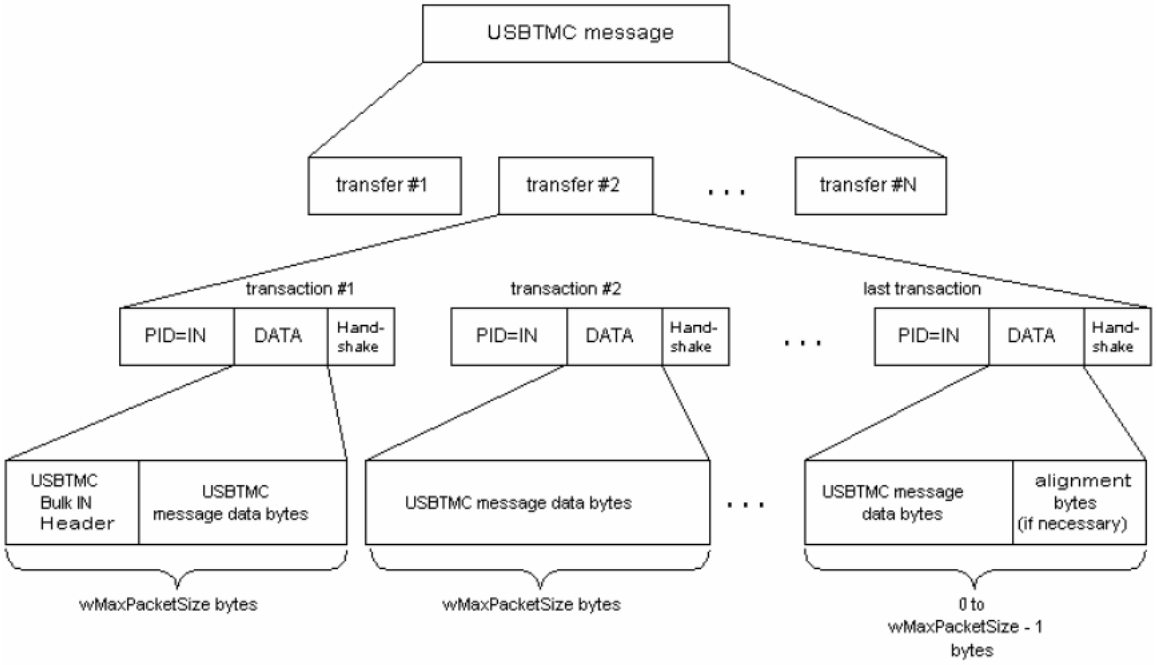

Figure 6: Bulk-IN USBTMC message sent with multiple transfers

Bulk-IN USBTMC command DEV_DEP_MSG_IN The device uses DEV_DEP_MSG_IN to identify the transfer as a USBTMC response message to the Host sending a REQUEST_DEV_DEP_MSG_IN USBTMC command message. The response specific content is shown in Table 12. A device may set Trans-

Offset	Field	Size	Value	Description
0	MsgID	1	0x02	Specifies this specific USBTMC message. Must match MsgID in the USBTMC command message transfer causing this response.
1	bTag	1	Value	Must match bTag in the USBTMC command message transfer causing this response.
2	bTagInverse	1	Value	Must match bTagInverse in the USBTMC command message transfer causing this response.
3	Reserved	1	0x00	Reserved. Must be 0x00.
4-7	TransferSize	4	Number	Total number of message data bytes to be sent in this USB transfer. This does not include the number of bytes in this header or alignment bytes. Sent least significant byte first, most significant byte last. TransferSize must be ¿ 0x00000000.
8	bmTransfer Attributes	1	Bitmap	D7..D2 Reserved. All bits must be 0.
				D1 1 - All of the following are true: The USBTMC interface supports TermChar The bmTransferAttributes.TermCharEnabled bit was set in the REQUEST_DEV_DEP_MSG_IN. The last USBTMC message data byte in this transfer matches the TermChar in the REQUEST_DEV_DEP_MSG_IN. 0 - One or more of the above conditions is not met.
				D0 EOM. 1 - The last USBTMC message data byte in the transfer is the last byte of the USBTMC message. 0 The last USBTMC message data byte in the transfer is not the last byte of the USBTMC message.
9-11	Reserved	3	0x000000	Reserved. Must be 0x000000.

Table 12: DEV_DEP_MSG_IN Bulk-IN Header

ferSize larger than the number of message data bytes it can buffer, provided the device knows the exact number of USBTMC message data bytes it will eventually send in the transfer. The Host must ignore EOM if the device does not send TransferSize message data bytes.

Bulk-IN USBTMC command VENDOR_SPECIFIC_IN The device uses VENDOR_SPECIFIC_IN to identify the transfer as a USBTMC response message to the Host sending a REQUEST_VENDOR_SPECIFIC_IN USBTMC command message. The response specific content is shown in Table 13. A device may set Trans-

Offset	Field	Size	Value	Description
0	MsgID	1	0x7F	Specifies this specific USBTMC message. Must match MsgID in the USBTMC command message transfer causing this response.
1	bTag	1	Value	Must match bTag in the USBTMC command message transfer causing this response.
2	bTagInverse	1	Value	Must match bTagInverse in the USBTMC command message transfer causing this response.
3	Reserved	1	0x00	Reserved. Must be 0x00.
4-7	TransferSize	4	Number	Total number of message data bytes to be sent in this USB transfer. This does not include the number of bytes in this header or alignment bytes. Sent least significant byte first, most significant byte last. TransferSize must be ¿ 0x00000000.
8-11	Reserved	4	0x00000000	Reserved. Must be 0x0000000.

Table 13: VENDOR_SPECIFIC_IN Bulk-IN Header

ferSize larger than the number of message data bytes it can buffer, provided the device knows the exact number of USBTMC message data bytes it will eventually send in the transfer.

A.3 Control endpoint requests

A.3.1 Standard requests

USBTMC devices must support the standard requests required by the USB 2.0 specification, section 9.4. In addition, they must follow the behaviors below:

- The Host, after sending a CLEAR_FEATURE request to clear a Halt condition on a USBTMC interface Bulk-OUT endpoint, must begin the next Bulk-OUT transaction with a Bulk-OUT Header.
- The device, after receiving the CLEAR_FEATURE request, must interpret the first part of the next Bulk-OUT transaction as a new USBTMC Bulk-OUT Header.
- The Host, after sending a CLEAR_FEATURE request to clear a Halt condition on a USBTMC interface Bulk-IN endpoint, must interpret the next Bulk-IN transaction as a new transfer beginning with a new USBTMC Bulk-IN Header.
- The device, after receiving the CLEAR_FEATURE request, must not queue any Bulk-IN DATA until it receives a USBTMC command message that expects a response.

A.3.2 USBTMC class specific requests

All USBTMC class specific requests must be sent with a Setup packet as shown below in Table 14. All USBTMC class-specific requests return data to the Host (bmRequestType direction = Device-to-host) and have a data payload that begins with a 1 byte USBTMC_status field. The USBTMC_status values are defined below in Table 16.

Offset	Field	Size	Value	Description
0	bmRequestType	1	Bitmap	D7: Data transfer direction 0 - Host-to-device 1 - Device-to-host D6..D5: Type 0 - Standard 1 - Class 2 -Vendor 3 - Reserved Type = Class for all control endpoint requests specified in this USBTMC specification D4..D0: Recipient 0 - Device 1 - Interface 2 - Endpoint 3 - Other 4..31 - Reserved
1	bRequest	1	Value	Specify request. See Table 15
2	wValue	2	Value	Word sized field that varies according to request. See the USB 2.0 specification, section 9.3.3.
4	wIndex	2	Index or Offset	Word sized field that varies according to request, typically used to pass an index or offset. See the USB 2.0 specification, section 9.3.4.
6	wLength	2	Count	Number of bytes to transfer if there is a Data stage. See the USB 2.0 specification, section 9.3.5.

Table 14: USBTMC class specific request format

A response with USBTMC_status indicating a failure (>= 0x80) must contain all of the required response bytes. Devices should send the most appropriate and most specific USBTMC_status.
Devices must return a STALL PID in response to the next Data stage transaction or in the Status stage of the message, when the device receives a request that is not defined for the device, is inappropriate for the current setting of the device, or has values that are not compatible with the request.
If a Setup transaction is received by an endpoint before a previously initiated control transfer is completed, the device must abort the current transfer/operation and handle the new control Setup transaction.

USBTMC split transactions USBTMC split transactions are specified for operations that on some test and measurement devices may take a long time. USBTMC split transactions are done with an INITI-

bRequest	Name	Required/Optional	Comment
0	Reserved	Reserved	Reserved.
1	INITIATE_ABORT_BULK_OUT	Required	Aborts a Bulk-OUT transfer.
2	CHECK_ABORT_BULK_OUT_STATUS	Required	Returns the status of the previously sent INITIATE_ABORT_BULK_OUT request.
3	INITIATE_ABORT_BULK_IN	Required	Aborts a Bulk-IN transfer.
4	CHECK_ABORT_BULK_IN_STATUS	Required	Returns the status of the previously sent INITIATE_ABORT_BULK_IN request.
5	INITIATE_CLEAR	Required	Clears all previously sent pending and unprocessed Bulk-OUT USBTMC message content and clears all pending Bulk-IN transfers from the USBTMC interface.
6	CHECK_CLEAR_STATUS	Required	Returns the status of the previously sent INITIATE_CLEAR request.
7	GET_CAPABILITIES	Required	Returns attributes and capabilities of the USBTMC interface.
8-63	Reserved	Reserved	Reserved for use by the USBTMC specification.
64	INDICATOR_PULSE	Optional	A mechanism to turn on an activity indicator for identification purposes. The device indicates whether or not it supports this request in the GET_CAPABILITIES response packet.
65-127	Reserved	Reserved	Reserved for use by the USBTMC specification.
128-191	Reserved	Reserved	Reserved for use by USBTMC subclass specifications.
192-255	Reserved	Reserved	Reserved for use by the VISA specification.

Table 15: USBTMC bRequest values

ATE request followed by a CHECK_STATUS request. After receiving the INITIATE request, the device must queue the appropriate control endpoint response packet with the most appropriate USBTMC_status, that the Host should not interpret as a warning. If a device receives an INITIATE request, sends a control endpoint response packet with USBTMC_status = STATUS_SUCCESS, and then receives a new control endpoint request other than the expected CHECK_STATUS, the device behaviors depend on the request type.
If it is a Class endpoint requests and the device has a prepared CHECK_STATUS response packet, the device must discard it. All other actions started by the INITIATE should complete. If all other actions have already completed, the device must handle the new request. If the actions have not completed, the device must send the appropriate response packet with USBTMC_status = STATUS_SPLIT_IN_PROGRESS.
For Standard control endpoint requests, whenever possible, all actions started by the INITIATE should complete. If this is not possible, due to a resource conflict between the device resources affected by the standard request and the device resources being used or affected by the INITIATE, the device must abort the INITIATE.
In the case of Vendor control endpoint requests, the device must assume the USBTMC client software will never send a CHECK_STATUS request. If the device has a prepared CHECK_STATUS response packet,

USBTMC_status	MACRO	Recommended interpretation by Host software	Description
0x00	Reserved	Reserved	Reserved
0x01	STATUS_SUCCESS	Success	Success
0x02	STATUS_PENDING	Warning	This status is valid if a device has received a USBTMC split transaction CHECK_STATUS request and the request is still being processed.
0x03-0x1F	Reserved	Warning	Reserved for USBTMC use.
0x20-0x3F	Reserved	Warning	Reserved for subclass use.
0x40-0x7F	Reserved	Warning	Reserved for VISA use.
0x80	STATUS_FAILED	Failure	Failure, unspecified reason, and a more specific USBTMC_status is not defined.
0x81	STATUS_TRANSFER_NOT_IN_PROGRESS		This status is only valid if a device has received an INITIATE_ABORT_BULK_OUT or INITIATE_ABORT_BULK_IN request and the specified transfer to abort is not in progress.
0x82	STATUS_SPLIT_NOT_IN_PROGRESS	Failure	This status is valid if the device received a CHECK_STATUS request and the device is not processing an INITIATE request.
0x83	STATUS_SPLIT_IN_PROGRESS	Failure	This status is valid if the device received a new class-specific request and the device is still processing an INITIATE.
0x84-0x9F	Reserved	Failure	Reserved for USBTMC use.
0xA0-0xBF	Reserved	Failure	Reserved for subclass use.
0xC0-0xFF	Reserved	Failure	Reserved for VISA use.

Table 16: USBTMC_status values

the device must discard it. All other actions started by the INITIATE should complete. If all other actions have completed, the device must handle the new request. If the actions have not completed, the device must respond with a Request Error.

INITIATE_ABORT_BULK_OUT A Host may use the INITIATE_ABORT_BULK_OUT request to abort a Bulk-OUT transfer and restore Bulk-OUT synchronization. A Host should only send an INITIATE_ABORT_BULK_OUT request when re-synchronization is necessary. After receiving an INITIATE_ABORT_BULK_OUT request, the device must return a control endpoint response packet as shown in Table 17.

CHECK_ABORT_BULK_OUT_STATUS The Host uses CHECK_ABORT_BULK_OUT_STATUS to determine if the device has completed all processing associated with a previously received INITIATE_ABORT_BULK_ request. After receiving a CHECK_ABORT_BULK_OUT_STATUS request, the device must return a control endpoint response packet as shown in Table 18.

Offset	Field	Size	Value	Description
0	USBTMC_status	1	Value	Status indication for this request. STATUS_SUCCESS: the specified transfer is in progress. STATUS_TRANSFER_NOT_IN_PROGRESS: there is a transfer in progress, but the specified bTag does not match or there is no transfer in progress, but the Bulk-OUT FIFO is not empty. STATUS_FAILED: there is no transfer in progress and the Bulk-OUT FIFO is empty.
1	bTag	1	Value	The bTag for the the current Bulk-OUT transfer. If there is no current Bulk-OUT transfer, bTag must be set to the bTag for the most recent bulk-OUT transfer. If no Bulk-OUT transfer has ever been started, bTag must be 0x00.

Table 17: INITIATE_ABORT_BULK_OUT response packet

Offset	Field	Size	Value	Description
0	USBTMC_status	1	Value	Status indication for this request. STATUS_PENDING: The device has not yet aborted the specified transfer and is unable to calculate NBYTES_RXD STATUS_SUCCESS: The device has aborted the specified transfer. The device must set NBYTES_RXD to the appropriate value.
1-3	Reserved	3	0x000000	Reserved. Must be 0x000000.
4	NBYTES_RXD	4	Number	The total number of USBTMC message data bytes (not including Bulk-OUT Header or alignment bytes) in the transfer received, and not discarded, by the device. The device must always send NBYTES_RXD bytes to the Function Layer. Sent least significant byte first, most significant byte last.

Table 18: CHECK_ABORT_BULK_OUT_STATUS response packet

INITIATE_ABORT_BULK_IN A Host may use the INITIATE_ABORT_BULK_IN request to abort a Bulk-IN transfer and restore Bulk-IN synchronization. A Host should only send an INITIATE_ABORT_BULK_IN request when resynchronization is necessary. After receiving the request, the device must return a control endpoint response packet as shown in Table 19.

CHECK_ABORT_BULK_IN_STATUS The Host uses CHECK_ABORT_BULK_IN_STATUS to determine if the device has completed all processing associated with a previously received INITIATE_ABORT_BULK_IN request. After receiving the request, the device must return a control endpoint response packet as shown in Table 20.

INITIATE_CLEAR The Host uses INITIATE_CLEAR to clear all input buffers and output buffers associated with the specified USBTMC interface. After receiving the request, the device must Halt the Bulk-OUT endpoint, queue the control endpoint response shown in Table 21, and clear all input buffers and output buffers.

Offset	Field	Size	Value	Description
0	USBTMC_status	1	Value	Status indication for this request. STATUS_SUCCESS: the specified transfer is in progress. STATUS_TRANSFER_NOT_IN_PROGRESS: there is a transfer in progress, but the specified bTag does not match or there is no transfer in progress, but the Bulk-OUT FIFO is not empty. STATUS_FAILED: there is no transfer in progress and the Bulk-OUT FIFO is empty.
1	bTag	1	Value	The bTag for the current Bulk-IN transfer. If there is no current Bulk-IN transfer, bTag must be set to the bTag for the most recent bulk-IN transfer. If no Bulk-IN transfer has ever been started, bTag must be 0x00.

Table 19: INITIATE_ABORT_BULK_IN response packet

Offset	Field	Size	Value	Description
0	USBTMC_status	1	Value	Status indication for this request. STATUS_PENDING: if a short packet has not been sent or the device is not ready to receive a USBTMC command message that expects a response.The device must set NBYTES_TXD = 0x00000000. STATUS_SUCCESS: if a short packet has been sent, the Bulk-IN FIFO is empty, and the device is ready to receive a USBTMC command message that expects a response. The device must set NBYTES_TXD to the appropriate value.
1	bmAbortBulkIn	1	Bitmap	D7..D1: Reserved. All bits must be 0. D0 BulkInFifoBytes 1 - The device either has some queued DATA bytes in the Bulk-IN FIFO or has a short packet that needs to be sent to the Host. The USBTMC_status must not be STATUS_SUCCESS. 0 - The Bulk-IN FIFO is empty.
2-3	Reserved	2	0x0000	Reserved. Must be 0x0000.
4	NBYTES_TXD	4	Number	The total number of USBTMC message data bytes (not including Bulk-IN Header or alignment bytes) sent in the transfer. Sent least significant byte first, most significant byte last.

Table 20: CHECK_ABORT_BULK_IN_STATUS response packet

Offset	Field	Size	Value	Description
0	USBTMC_status	1	Value	Status indication for this request. STATUS_SUCCESS: if it has set up a Halt condition on the Bulk-OUT endpoint. The device, after the response packet is queued, must clear input and output buffers.

Table 21: INITIATE_CLEAR response packet

CHECK_CLEAR_STATUS The Host uses CHECK_CLEAR_STATUS to determine if the device has completed all processing associated with a previously received INITIATE_CLEAR request. Upon receiv-

ing the CHECK_CLEAR_STATUS request, the device must determine if it is still processing an INITIATE_CLEAR and then queue the control endpoint response packet shown below in Table 22.

Offset	Field	Size	Value	Description
0	USBTMC_status	1	Value	Status indication for this request. STATUS_PENDING: Either the device has not yet finished clearing input buffers and output buffers, or the Bulk-IN FIFO is not empty, or the Function Layer is not ready for Bulk transfers. STATUS_SUCCESS: The device has finished clearing the input and output buffers, the Bulk-IN FIFO is empty, and the Function Layer is ready for Bulk transfers.
1	bmClear	1	Bitmap	D7..D1: Reserved. All bits must be 0. D0 BulkInFifoBytes 1 - The device either has some queued DATA bytes in the Bulk-IN FIFO that it could not remove, or has a short packet that needs to be sent to the Host. The USBTMC_status must not be STATUS_SUCCESS. 0 - The device has completely removed queued DATA in the Bulk-IN FIFO and the Bulk-IN FIFO is empty.

Table 22: CHECK_CLEAR_STATUS response packet

GET_CAPABILITIES The Host uses GET_CAPABILITIES to read additional attributes and capabilities of a USBTMC interface. A device must be ready to receive a GET_CAPABILITIES request at any time. When a device receives this request, the device must queue the control endpoint response shown below in Table 23.

INDICATOR_PULSE This request provides the Host with a mechanism to turn on an activity indicator for identification purposes. A device indicates whether it supports this request in the GET_CAPABILITIES response packet. A device must be ready to receive an INDICATOR_PULSE request at any time. If not implemented, the device must respond with a Request Error. When a device receives this request, the device must queue the control endpoint response shown below in Table 24. If the device supports the request, the device then turns on an implementation-dependent activity indicator for a human detectable length of time (recommend time is >= 500 milliseconds and <= 1 second). The activity indicator then automatically turns off.

Offset	Field	Size	Value	Description
0	USBTMC_status	1	Value	Status indication for this request. STATUS_PENDING: if a short packet has not been sent or the device is not ready to receive a USBTMC command message that expects a response.The device must set NBYTES_TXD = 0x00000000. STATUS_SUCCESS: if a short packet has been sent, the Bulk-IN FIFO is empty, and the device is ready to receive a USBTMC command message that expects a response. The device must set NBYTES_TXD to the appropriate value.
1	Reserved	1	0x00	Reserved. Must be 0x00.
2	bcdUSBTMC	2	BCD (0x0100 or greater)	BCD version number of the relevant USBTMC specification for this USBTMC interface. Format is as specified for bcdUSB in the USB 2.0 specification, section 9.6.1.
4	USBTMC Interface Capabilities	1	Bitmap	D7..D3: Reserved. All bits must be 0. D2 1 - The USBTMC interface accepts the INDICATOR_PULSE request. 0 - The USBTMC interface does not accept the INDICATOR_PULSE request. The device, when an INDICATOR_PULSE request is received, must treat this command as a non-defined command and return a STALL handshake packet. D1 1 - The USBTMC interface is talk-only. 0 - The USBTMC interface is not talk-only. D0 1 - The USBTMC interface is listen-only. 0 - The USBTMC interface is not listen-only.
5	USBTMC Device Capabilities	1	Bitmap	D7..D1: Reserved. All bits must be 0. D0 1 - The device supports ending a Bulk-IN transfer from this USBTMC interface when a byte matches a specified TermChar. 0 - The device does not support ending a Bulk-IN transfer from this USBTMC interface when a byte matches a specified TermChar.
6	Reserved	6	All bytes must be 0x00.	Reserved for USBTMC use. All bytes must be 0x00.
12	Reserved	12	Reserved	Reserved for USBTMC subclass use. If no subclass specification applies, all bytes must be 0x00.

Table 23: GET_CAPABILITIES response packet

Offset	Field	Size	Value	Description
0	USBTMC_status	1	Value	Status indication for this request.

Table 24: INDICATOR_PULSE response packet