USB CDC-ACM Driver for the Texas Instruments TMS320C6748 DSP

v1.3.0

August 2012

Contents

Chapter 1

Introduction

This document contains details of Multiple Access Communication Limited's USB Driver for the Texas Instrument's C6748 DSP and other compatible DSPs.

This document is a concise description of how to use the USB driver, including an example program to aid rapid deployment of the driver, and reference descriptions of all the user accessible functions.

In addition to the USB driver, Multiple Access Communications Limited also provides a number of supporting C++ classes that can be used with the USB driver. These support the passing of memory (buffers) to and from the USB driver.

1.1 USB Driver Overview

The USB driver is a pre-compiled library that contains an implementation of a CDC-ACM device. This type of device is commonly referred to as a serial port, a COM port, a UART or an RS232 port. It allows the simple transfer of data to and from the host(PC) and the device(DSP) via the USB.

The USB driver mimics a hardware UART, but all communication is performed according to the requirements of the USB standard. This means that there are some differences when compared to the operation of a real UART or RS232 port. A significant difference is that a hardware UART often implements some form of flow control. This can be some form of dedicated hardware signalling on separate wires or via data handshaking or high layer communication protocols. This flow control is implemented to prevent data loss, something that cannot occur with Multiple Access Communications Ltd's USB driver. The USB driver is implemented such that no data can be lost, and flow control is managed by the host as it is in complete control of all data transfers.

An additional difference is the speed at which data may be transferred. The USB driver is only limited by two factors:

- the number of DSP cycles available for data transfers and
- the proportion of time that the DSP can use the USB as controlled by the host.

The USB 2.0 system has a theoretical data transfer rate of 480 Mbit/s, but once signalling and other overheads are taken into account real transfer rates can be no more than 360 Mbit/s. To fully utilise the USB, both the host system and the device must be able to send and receive data when the opportunity arises. Commonly these maximum speeds are only achieved with mass storage devices such as hard disk drives or flash storage media as there is always data to send. Data rates for other devices can be lower. The USB driver supports data transfer rates of over 240 Mbit/s from the host to the device, and from device to host. Higher data rates are possible from the device to the host, but significant optimisation is required. The combined data transfer rate, if transferring data in both directions simultaneously can also be over 240 Mbit/s. It is also worth noting that setting the baud rate, stop bits, flow control method, and number of data bits has no effect on the performance of the USB driver.

1.2 Using the USB Driver

The USB driver is designed to be used in customer applications with the simplest of setup. To use the USB driver the developer must construct a USB driver object. To support the reading and writing of memory the developer will also need to construct some BufferQueueObjects. These objects point to raw memory buffers, e.g. arrays of bytes, and contain the size of the raw memory and also the number of valid bytes in the buffer.

1.2.1 Detecting connection to the USB

The first step to using the USB driver is to register a semaphore object to detect when the USB is connected to a host, e.g. a PC. It is possible to register one semaphore per port, but as all the semaphores are posted to at the same time, only one is necessary.

1.2.2 Initializing the USB Driver

The second step is to initialize the USB driver. This allows the developer to set the number of ports that the program will be using, in the range one to three, and the serial number the USB driver will report to the host. Initialization also requires the developer to set which interrupt number the USB Driver should use and which ports, if any, should be double buffered. Double buffering a port should provide an improvement to the data transfer rate for that port, but is dependent on a number of other factors, see the initialize function for a more detailed discussion. The final two elements required at the initialization stage are a function that can be used to control the USB peripheral in the DSP's power sleep controller, and whether the USB's data output pins should be inverted or not. It is expected that the data output will be inverted because the USB is operating in peripheral mode.

1.2.3 Reading data from the USB

To read data from the USB, the USB driver must be initialized and connected to a host. The developer must also register a callback function with the port that is to receive data. To do this the registerRxCallback function is called, with the port number and the callback function as arguments. The next step is to submit a BufferQueueObject to the USB driver using the submitRead function. The BufferQueueObject should contain a valid pointer to a raw memory buffer, the buffer's size, and the number bytes that are already in use at the beginning of the buffer. Initially the number of bytes used in the buffer should be set to zero. Setting any other value will mean that any received data will be written to the raw memory buffer start offset by the number of bytes used. After the buffer has been submitted, the USB driver will wait to receive data on the specified port and copy the received data into the BufferQueueObject. Once the USB driver has filled the buffer or the USB transfer has finished, the USB driver wll call the callback function for that port. The callback function will then be called with the BufferQueueObject with the bytesUsed field incremented by the number of bytes the USB driver read. It is then possible to process the data, although it should be noted the callback is called from the USB's interrupt handler.

1.2.4 Writing data to the USB

Writing data to the USB requires that the USB driver is initialized and connected to a host. To write data to the USB the developer must use a BufferQueueObject. The - BufferQueueObject's addr field must point to the address of the data to be sent. The size field should be set to the size of the memory in bytes at the addr field's location and the bytesUsed field should be set to indicate how many bytes at the addr location are valid and to be sent. In addition to setting the bytesUsed field the returnQueue field should also be set. The returnQueue field is used by the USB driver to release or return the BufferQueueObject once the content has been sent. The developer will therefore need to construct a BufferQueue or a BlockBufferQueue and use the address of this queue to set the returnQueue field. The BufferQueueObject can then be sent using the submitWrite function, passing in the port number and the BufferQueueObject. The BufferQueueObject will then be returned to its return queue once the data has been sent, or immediately if the USB is not connected.

The Example Program chapter shows how the API is used and gives example callback functions.

Note that the best data-rates are achieved if the data to be sent is aligned on a 32 bit boundary.

1.3 Evaluation Driver

There are two versions of the driver, the evaluation version and the full version. The evaluation version, which is made available solely for the purposes of evaluation of the driver, has been designed to fail after about an hour of use. The full version does not have this restriction.

Chapter 2

Example Program

2.1 Overview of the Example Program

An example program that uses the USB driver is provided. This example has six distinct functions.

The first function is the standard main function. This sets up the memory buffers and queue for use by the USB driver and other functions in the example program.

The second function is the application's USB task, this task performs the steps of initializing the USB driver, registering callback functions for each of the ports the USB driver provides, waiting for the USB driver to signal that the DSP is now connected, and submitting buffers to be used for reading from the USB ports.

The third, fourth and fifth functions are callback functions. The usb0ExampleCallback function is called by the USB driver when receiving data on port 0. This function immediately sends the received data back on port 0. Functions usb1ExampleCallback and usb2ExampleCallback behave slightly differently. Data received on port 1 is modified and looped back on port 2, whilst data received on port 2 is modified and looped back on port 1. (Although we annotate the port number 0, 1 and 2, it is known that Microsoft Windows (TM) may or may not allocate the port numbers in the same order. This can be controlled using configuration files when first connecting the device, or could be differentiated using customer code.)

The sixth function is the usbPowerSleepController function. This function is used by the USB driver to enable and disable the USB peripheral in the power sleep controller(PSC). This function is given as a useful utility function to enable the USB driver to be quickly evaluated, but similar functionality could be provided by the developer.

2.1.1 Running the Example Program

To run the example program, compile and download the program to a DSP, then connect the DSP to a host device, e.g. a PC.

If testing using Microsoft Windows (TM) use the provided .inf files to install the new USB

virtual COM ports. This may warn that the driver is not signed, never the less it is safe to continue installation because the device uses Microsoft's own USB COM port device driver.

From the PC open a terminal program, e.g. hyperterminal, connecting to each of the serial ports separately. The first serial port's terminal will echo the character typed into it immediately, whilst the second will convert all the characters to uppercase and then echo the typed characters to the third terminal's screen. The third terminal will echo the characters typed onto the second terminal's screen, but with the characters converted to lower-case. This is the expected behaviour of the example program. This program is listed in Appendix A Example Program Listing, and an explanation of that program line by line is presented here.

2.2 The #includes

Initially a number of #include directives are defined.

```
#include <stdint.h>
```

This file includes the standard types with known sizes such as uint8_t and int16_t for unsigned 8-bit integer and signed 16-bit integer. These are useful for cross platform development and are specified in the C99 standard and are expected to be adopted by C++0x.

```
#include <std.h>
```

This #include includes the DSP/BIOS standard types and is commonly required before any other DSP/BIOS header files.

```
#include <cctype>
```

This header file is included so that the conversion routines toupper() and tolower() can be used in the callback functions.

```
#include "MacDspBiosSupport/BufferQueue.h"
```

This #include is used to define the memory buffers and queues used with the USB driver.

```
#include "UsbDriver/CdcAcmUsbController.h"
```

This is the USB driver's header file and declares all the user functions for controlling the USB Driver.

2.3 Constants

After including all the requred headers a number of useful constants are defined so that the program may be altered easily and to assist in the readability of the code.

```
const uint8_t NUMBER_OF_PORTS(3);
```

This controls the number of ports that this example program will create A maximum of 3 ports may be specified for the USB driver.

```
const uint32_t BUFFER_SIZE( 1024 * 8 );
```

Memory buffers of 8k bytes are created for sending and receiving data

```
const std::size_t NUM_MESSAGE_BUFFERS_PER_PORT( 9 );
```

Each port is allocated 9 buffers for sending and receiving on the USB. The example program allocates the same number of buffers for each port, but a real program may provide more buffers for some ports.

2.4 Forward Declaration of Function Prototypes

The next section of code declares 5 function prototypes.

```
extern "C" void usbTsk();

void usb0ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
   rxBuffer );
void usb1ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
   rxBuffer );
void usb2ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
   rxBuffer );

void usbPowerSleepController( bool enable );
```

These functions are defined later in the main.cpp file, except for the usbPowerSleep-Controller function. This function is defined separately.

2.5 Global Variables

The example program defines a number of globally accessible objects. These objects are the memory buffer queues, the buffer queue objects that point to and manage the memory buffers, and a large memory buffer. The large memory buffer is constructed with a #pragma directive so that the Texas Instrument's compiler aligns the memory on a cache boundary. This enables the USB driver to operate more efficiently because all accesses to the memory can be accessed using 32-bit or 64-bit memory operations, except for the last few bytes of data read or written in each data transfer.

```
MacDspBiosSupport::BufferQueue rxBufferQueues[ NUMBER_OF_PORTS ];

MacDspBiosSupport::BufferQueueObject bufferQueueObjects[
   NUMBER_OF_MESSAGE_BUFFERS_PER_PORT * NUMBER_OF_PORTS ];

#pragma DATA_ALIGN( 128 )
char bufferMemory[ BUFFER_SIZE * NUMBER_OF_MESSAGE_BUFFERS_PER_PORT *
    NUMBER_OF_PORTS ];
```

The final global object is the usbController object itself.

```
UsbDriver::CdcAcmUsbController usbController;
```

The usbController is an instance of our USB driver and allows access to all the facilities the USB driver offers.

2.6 The main() Function

The main function is a required function for all DSP/BIOS programs. In our example program main() assigns a block of memory from the bufferMemory variable to each of the bufferQueueObjects. The bufferQueueObjects are then pushed into one of the rxBufferQueues. Before pushing each of the bufferQueueObjects into a BufferQueue the returnQueue is set. This allows the object to be returned when it is not needed to a particular queue of free bufferQueueObjects. In the example program the bufferQueueObjects are shared equally amongst the rxBufferQueues, and are all of equal size. In real applications there may be differing numbers and sizes of buffers, in which case a number of queues of free objects will need to be created.

```
int main()
{
   int bufferNumber(0);
   for( int port = 0; port < NUMBER_OF_PORTS; ++port )
   {
      for( int buffersAdded = 0 ; buffersAdded <
   NUMBER_OF_MESSAGE_BUFFERS_PER_PORT; ++buffersAdded, ++bufferNumber )
      {
         bufferQueueObjects[ bufferNumber ].addr        = (uint8_t*)&
   bufferMemory[ bufferNumber * BUFFER_SIZE ];
         bufferQueueObjects[ bufferNumber ].size        = BUFFER_SIZE;
         bufferQueueObjects[ bufferNumber ].bytesUsed   = 0;
         bufferQueueObjects[ bufferNumber ].returnQueue = &rxBufferQueues[
   port];

         rxBufferQueues[port].push( &bufferQueueObjects[ bufferNumber ] );
      }
   }

   return 0;
}
```

2.7 The usbTsk() Function

The usbTsk function shows the main interaction between the example program and the USB driver. Taking each line at a time and describing its effects we see the first line defines a simple serial number used later by the USB driver.

```
void usbTsk()
{
   // Set the serial number in the USB descriptor.
   uint64_t serialNumber( 0x123456789ABCDEF0ULL );
```

The next step is to register a semaphore with the USB driver. A semaphore can be registered with each of the ports, but all semaphores will be posted to when the USB is connected and disconnected. In this example we just check when the USB connects to a host. It is important therefore to register the semaphore prior to initializing the USB driver, so that we know the USB is not connected to a host.

```
SEM_Obj connected;
SEM_new( &connected, 0 );
usbController.registerConfiguredStatusChangedSemaphore( 0, &connected );
```

The next section of code registers three callback functions to the three ports. If the USB driver is initialized to only one port the callbacks for the other ports will not be called. The first callback passes a pointer to the usbController as the optional user argument. This could be a pointer to any object or structure, and enables the same callback to be registered with each port, but for a different structure or class instance to be passed as the optional argument.

```
if( 1 <= NUMBER_OF_PORTS )
{
   usbController.registerRxCallback( 0, usb0ExampleCallback, (void*)&
   usbController );
}
if( 2 <= NUMBER_OF_PORTS )
{
   usbController.registerRxCallback( 1, usb1ExampleCallback );
}
if( 3 <= NUMBER_OF_PORTS )
{
   usbController.registerRxCallback( 2, usb2ExampleCallback );
}
```

The next line initializes the USB driver with all the information needed to enable the driver to respond to a USB host when the DSP is connected. After this function call the DSP will respond to any USB host that it is connected to.

```
usbController.initialize( NUMBER_OF_PORTS,
                          usbPowerSleepController,
                          0,
                          true,
                          19,
                          7,
                          0,
```

```
    UsbDriver::CdcAcmUsbController::NO_DOUBLE_BUFFERING,

    UsbDriver::CdcAcmUsbController::NO_DOUBLE_BUFFERING,
                          true,
                          true,
                          serialNumber );
```

The initialize function's parameters are set to so that the USB driver will create three ports and use the provided power sleep control function, with a null argument. It is also set to use the ECM peripheral for interrupts. For the C6748, the next three numbers set the ECM peripheral to use the USB hardware interrupt number (19), and the ECM interrupt handler attached to interrupt seven, with a interrupt selection number of zero. This will enable interrupts to be correctly enabled for the USB. The following two lines disable double buffering on the transmit and receive ports. This can be altered to enable double buffering on one port in one direction if three ports are enabled. The second to last option, 'true', specifies that the USB data output lines are to be inverted, due to its use as a peripheral. (This is the option that is most likely to need changing if the example program does not work for your hardware.) The following 'true' value specifies that the USB hardware should request to operate in high-speed mode. If this value is false the USB hardware will only run in full-speed mode. The final value is the serial number, it is recommended that this is always set to a non-zero value because this aids registration under some operating systems and is the only way to distinguish two USB devices of the same type on the USB bus.

The example program then waits for the USB driver to post to the connected semaphore signifying that the device has been connected.

```
SEM_pendBinary( &connected, SYS_FOREVER );
```

The following code then continuously submits read buffers to the three ports from their three separate buffer queues. Initially the buffer queues will be full, and so this loop will successfully submit all the read buffers to the USB driver. The buffer queues will then all be empty until one of the ports receives some data. The data received will then be copied into a receive buffer and one of the callback functions will be called. The callback functions in the example program then re-submit the buffer to the USB driver for sending to the host. Once sent the buffers are returned or released to their returnQueue, using the release() function. At this point the read buffer queues will have a new buffer to submit to the USB driver for reading more data.

```
while( 1 )
{
    for( int port = 0; port < NUMBER_OF_PORTS; ++port )
    {
        MacDspBiosSupport::BufferQueueObject *bqo =
            rxBufferQueues[ port ].pop();

        if( 0 != bqo )
        {
            bqo->bytesUsed = 0;
            usbController.submitRead( port, bqo );
        }
    }
}
```

2.8 The Example Callback Functions

The usb0ExampleCallback function is first registered with the USB driver in the usbTsk function. The function is only called when the USB driver receives a complete message or the read buffer is full. The first argument, ptr, is the optional user argument that was passed when the callback function was registered. In this case, it was a pointer to the usbController, and so we re-cast the pointer back to its original type. The second argument is the received rxBuffer object. This object contains a pointer to the buffer that the USB driver has filled, and the number of bytes that are valid in that buffer up to the size of the buffer. This allows the receiving callback function to interpret any message sent or piece together the parts of a message in a separate buffer or using several received buffers. This example callback however, immediately submits the received buffer to be written out via the port that it received the data from, port 0.

```
void usb0ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
   rxBuffer )
{
   UsbDriver::CdcAcmUsbController* usbController = (
   UsbDriver::CdcAcmUsbController*)ptr;

   usbController->submitWrite( 0, rxBuffer );
}
```

The final two functions are also callbacks. The usb1ExampleCallback function echoes the data received on port 1 out to port 2 after first converting each character to upper-case. The usb2ExampleCallback function operates similarly, but echoes the data received on port 2 to the output of port 1, and also converts the received characters, but this time to lower-case. The number of bytes received by each callback is contained in the BufferQueueObject's bytesUsed field. To change the number of bytes echoed, the callback functions could reduce the value in the bytesUsed field. Alternatively, if it was desirable to append bytes to the end of a received buffer, the bytesUsed field could be increased and the data appended in the buffer, being careful not to overrun the underlying memory buffer as indicated by the size field of the BufferQueueObject. Both these functions did not pass an optional user argument when registering them, and therefore neither function uses the ptr argument.

```
void usb1ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
   rxBuffer )
{
   for( int currentCharacter = 0;
        currentCharacter < rxBuffer->bytesUsed;
        ++currentCharacter )
   {
      int upperChar = std::toupper( rxBuffer->addr[ currentCharacter ] );
      if( 0 <= upperChar )
      {
         rxBuffer->addr[ currentCharacter ] = (char)upperChar;
      }
   }

   usbController.submitWrite( 2, rxBuffer );
}

void usb2ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
```

```
    rxBuffer )
{
    // convert to lowercase
    for( uint8_t* currentAddr = rxBuffer->addr;
         currentAddr < ( rxBuffer->addr + rxBuffer->bytesUsed );
         ++currentAddr )
    {
       // convert the character at the current address to lowercase
       int lowerChar = std::tolower( *currentAddr );
       if( 0 <= lowerChar )
       {
          *currentAddr = (char)lowerChar;
       }
    }

    usbController.submitWrite( 1, rxBuffer );
}
```

Chapter 3

USB Driver Class Documentation

3.1 UsbDriver::CdcAcmUsbController Class Reference

This class is able to configure and manage the USB2.0 peripheral on the Texas - Intruments' C674x DSP family as a USB CDC-ACM device. It is designed and produced by Multiple Access Communications Ltd.

Public Types

- enum DoubleBufferControlWord { **NO_DOUBLE_BUFFERING** = 0, **PORT1** = 1, **PORT2** = 2, **PORT1_AND_2** = 3, **PORT3** = 4, **PORT1_AND_3** = 5, **PORT2_AND_3** = 6, **PORT1_AND_2_AND_3** = 7 }

  This enum is used to control which ports are set to be double buffered. The values are separate for the transmit and receive ports.

- typedef void(∗ UsbPowerSleepControlFunctionPtr)(void ∗, bool)

  This is the USB power sleep control function function type. The C674x contains a Power Sleep Control block that can be used to change the power state of the peripherals on the DSP. This function pointer should point to a function that can be used by the USB driver whilst initializing the USB to enable and disable the power to the USB2.0 peripheral. The function is also used when the device is connected and disconnected to a USB Host. The first argument is a copy of the pointer supplied to the initialize function, usbPowerSleepControlFunctionArg. The bool argument is set to true to enable the USB peripheral and false to disable it.

- typedef void(∗ RxTransferCompleteFunctionPtr)(void ∗, MacDspBiosSupport::-BufferQueueObject ∗)

  This is the receive transfer complete callback function type. A function of this type should be registered for each port to receive buffers that have been filled with data from the USB. The function is called when the USB driver has filled the current buffer, or the USB transfer has completed. Large USB data transfers are split into packets of 64 bytes (full-speed) or 512 bytes (high-speed). The data transfer is marked as complete when a packet is received that is not the maximum size of 64 or 512. If data is transferred that is an exact multiple of the maximum size then an empty packet is sent to mark the end of a transfer. The completed data transfer causes the callback

function to then be called. This means that the callback is called in the context of the interrupt handler, it is therefore important to not call functions that will block, e.g. SEM_pend(), and also to limit the size of the function so that DSP/BIOS can respond to other interrupts in a timely manner. The number of bytes received by the USB driver is added to the BufferQueueObject's bytesUsed field, therefore if read buffers are submitted with zero bytesUsed then the bytesUsed field will be the number of bytes received in this transfer. The value of the bytesUsed field will not exceed the value of the size field in the BufferQueueObject.

Public Member Functions

- void initialize (uint8_t numberOfPorts, UsbPowerSleepControlFunctionPtr usbPowerSleepControlFunctionPtr, void *usbPowerSleepControlFunctionArg, bool useEcm, uint8_t usbHardwareInterruptSourceNumber, uint8_t usbDesiredInterruptNumber, uint8_t ecmInterruptSelectionNumber, DoubleBufferControlWord whichTxPortsToDoubleBuffer, DoubleBufferControlWord whichRxPortsToDoubleBuffer, bool isUsbDataOutputInverted, bool enableHighSpeedMode, uint64_t serialNumber)

  A function to initialize the usb controller.
- bool submitRead (uint8_t portNumber, MacDspBiosSupport::BufferQueueObject *bufferQueueObject)

  This function adds a new read buffer to the read queue for the specified port.
- bool submitWrite (uint8_t portNumber, MacDspBiosSupport::BufferQueueObject *bufferQueueObject)

  This function is used to submit a buffer to be written via the specified port.
- void registerRxCallback (uint8_t portNumber, RxTransferCompleteFunctionPtr callbackFunction, void *callbackArg=0)

  This function registers a callback function to a specified port number.
- void registerConfiguredStatusChangedSemaphore (uint8_t portNumber, SEM_Handle configuredStatusChangedSemaphore)

  This function registers a semaphore to be set when the status of a given port changes.

3.1.1 Detailed Description

Definition at line 63 of file CdcAcmUsbController.h.

3.1.2 Member Enumeration Documentation

3.1.2.1 enum UsbDriver::CdcAcmUsbController::DoubleBufferControlWord

See also

initialize

Definition at line 70 of file CdcAcmUsbController.h.

3.1.3 Member Function Documentation

3.1.3.1 void UsbDriver::CdcAcmUsbController::initialize (uint8_t *numberOfPorts*, UsbPowerSleepControlFunctionPtr *usbPowerSleepControlFunctionPtr*, void * *usbPowerSleepControlFunctionArg*, bool *useEcm*, uint8_t *usbHardwareInterruptSourceNumber*, uint8_t *usbDesiredInterruptNumber*, uint8_t *ecmInterruptSelectionNumber*, DoubleBufferControlWord *whichTxPortsToDoubleBuffer*, DoubleBufferControlWord *whichRxPortsToDoubleBuffer*, bool *isUsbDataOutputInverted*, bool *enableHighSpeedMode*, uint64_t *serialNumber*)

The initialization of the USB controller occurs at a number of points. After calling this function the UsbDriver will have all the information needed to configure the USB peripheral and register the device as a periperal when the USB2.0 is connected to a USB Host. It will also set up all the necessary interrupts for the UsbDriver to operate.

Parameters

in	*numberOfPorts*	The number of serial ports to be registered with the Host for this device.
in	*usbPowerSleepControlFunctionPtr*	A function pointer that when called enables or disables the USB peripheral in the DSP's power sleep controller. An example function is provided that can be used.
in	*usbPowerSleepControlFunctionArg*	A void pointer that is passed as the first argument when the usbPowerSleepControlFunctionPtr function is called.
in	*useEcm*	This boolean controls whether the USB's interrupt handler is registered with the ECM module of DSP/BIOS or if false then the USB's interrupt handler will be registered directly with the HWI module of DSP/BIOS.
in	*usbHardwareInterruptSourceNumber*	This is the interrupt source number for the USB peripheral. On the C674x it is 19.
in	*usbDesiredInterruptNumber*	This is the interrupt number for which the ECM is using for handling the USB interrupt or it is the hardware interrupt number to connect the USB to directly if not using the ECM module. For the initial sample programs from TI the ECM is assigned interrupt numbers 7, 8, 9 and 10. For the C674x the USB is in group 0 (zero) which is normally assigned interrupt number 7.
in	*ecmInterruptSelectionNumber*	This is the interrupt selection number as defined by the ECM module for the USB interrupt. For the C6748 this is 0 (zero).

in	*whichTxPortsToDoubleBuffer*	This control word defines which of the transmit ports should be set to be double buffered. There are seven buffers available for the ports, and each port requires one buffer for each direction. For a single serial port configuration it is recommended to use double buffering on both receive and transmit. For two serial ports, it will depend on whether a greater emphasis is to be placed on one of the ports or on one of the directions of data flow. If the user needs greater data transfer rates from the device to the host, then the transmit ports should be double buffered, and the more commonly used receive port may also be double buffered. For three port configurations only one port and one direction of flow for that port may be double buffered. This is because six of the buffer are already allocated, two buffers per port. Note ALL ports are double buffered if the device is connected to using "Full-speed".
in	*whichRxPortsToDoubleBuffer*	This control word defines which of the receive ports should be set to be double buffered. See the description of whichTxPortsToDoubleBuffer for further details.
in	*isUsbDataOutputInverted*	This controls whether the physical output signals are inverted or not. This will depend on the DSP's board configuration. It is expected that this will be set to true because the DSP is acting as a peripheral.
in	*enableHighSpeedMode*	This controls whether the driver will request to operate in High-speed mode, if possible, when connected to a host PC.
in	*serialNumber*	Your product's serial number as a uint64_t. If this is 0 (zero) then no serial number will be used. A device with a fixed serial number in Microsoft Windows (TM) will maintain the same COM port number when the DSP is unplugged and replugged into the host PC.

3.1.3.2 bool UsbDriver::CdcAcmUsbController::submitRead (uint8_t *portNumber,* MacDspBiosSupport::BufferQueueObject * *bufferQueueObject*)

Parameters

in	*portNumber*	the port number the rxEndPointTransfer is to be queued on.
in	*bufferQueueObject*	a pointer to a read buffer structure.

Returns

true if the bufferQueueObject was successfully queued to be used to read data from the given port.

Note

Unlike the submitWrite function read buffers are held until data is read.

Warning

If the device is not connected the buffers are passed immediately to the read callback, but without any bytes having been read. It is important therefore to ensure that read buffers are not submitted whilst the USB is not connected.

3.1.3.3 bool UsbDriver::CdcAcmUsbController::submitWrite (uint8_t *portNumber,* MacDspBiosSupport::BufferQueueObject * *bufferQueueObject*)

Parameters

in	*portNumber*	The port to write the buffer to.
in	*bufferQueueObject*	A pointer to the packet to be written.

Returns

true if the txEndpointTransfer was successfully queued for transmission.

Note

If the device is not configured the buffers are released immediately, and the write is unsuccessful.

3.1.3.4 void UsbDriver::CdcAcmUsbController::registerRxCallback (uint8_t *portNumber,* RxTransferCompleteFunctionPtr *callbackFunction,* void * *callbackArg* = 0)

Parameters

in	*portNumber*	the port number to associate the function with
in	*callbackFunction*	the function to call with the read data
in	*callbackArg*	a pointer that is passed to the callback when the callback is called.

3.1.3.5 void UsbDriver::CdcAcmUsbController::registerConfiguredStatusChangedSemaphore (uint8_t *portNumber,* SEM_Handle *configuredStatusChangedSemaphore*)

Parameters

in	*portNumber*	the port number to associate the function with
in	*configuredStatusChangedSemaphore*	a pointer to the semaphore that will be set whenever the status of the port changes

Chapter 4

Supporting Class Documentation

4.1 MacDspBiosSupport::QueueObject Class Reference

This is the base object for any object to be put inside a Queue or Queue derivative.

Inherited by MacDspBiosSupport::BufferQueueObject.

Public Member Functions

- QueueObject * next () const

  From the queue this object is currently a member of, get the next object after this.
- QueueObject * prev () const

  From the queue this object is currently a member of, get the previous object before this.

4.1.1 Detailed Description

Definition at line 255 of file Queue.h.

4.1.2 Member Function Documentation

4.1.2.1 QueueObject* MacDspBiosSupport::QueueObject::next ( ) const

Returns

the next object in the Queue or the Head node for the Queue.

Note

If this is the last object in a queue then the queue's head is retrieved.
If the object is not part of a queue the behaviour is undefined.

Definition at line 270 of file Queue.h.

4.1.2.2 QueueObject* MacDspBiosSupport::QueueObject::prev ( ) const

Returns

the previous object in the Queue or the Head node for the Queue.

Note

If this is the first object in a queue then the queue's head is retrieved.
If the object is not part of a queue the behaviour is undefined.

Definition at line 276 of file Queue.h.

4.2 MacDspBiosSupport::BufferQueueObject Class Reference

A QueueObject that includes a buffer, a count of the valid bytes in the buffer and a pointer to the queue to return the buffer to after use.

Inherits MacDspBiosSupport::QueueObject.

Public Member Functions

- BufferQueueObject ()

  Constructs a BufferQueueObject with no associated memory block.
- BufferQueueObject (uint8_t *addrIn, uint32_t sizeIn)

  Constructs a BufferQueueObject from a block of memory pointed to by addrIn of a size(sizeIn).
- void release ()

  Used to return the buffer to its originating queue object.
- QueueObject * next () const

  From the queue this object is currently a member of, get the next object after this.
- QueueObject * prev () const

  From the queue this object is currently a member of, get the previous object before this.

Public Attributes

- uint8_t * addr

  The buffer start address.
- uint32_t size

  The size in bytes of the buffer pointed to by addr.
- uint32_t bytesUsed

  The bytes that are valid in the buffer.
- BufferQueue * returnQueue

  This is used to return the object to a known queue when the object is finished with.

4.2.1 Detailed Description

Note

This class is used by many of the communications functions to pass blocks of memory around avoiding dynamic allocation

Definition at line 53 of file BufferQueue.h.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 MacDspBiosSupport::BufferQueueObject::BufferQueueObject (uint8_t * *addrIn*, uint32_t *sizeIn*)

Parameters

in	*addrIn*	A pointer to the address of a block of memory.
in	*sizeIn*	The size of the memory in bytes pointed to by addrIn.

Definition at line 63 of file BufferQueue.h.

4.2.3 Member Function Documentation

4.2.3.1 QueueObject* MacDspBiosSupport::QueueObject::next ( ) const [inherited]

Returns

the next object in the Queue or the Head node for the Queue.

Note

If this is the last object in a queue then the queue's head is retrieved.
If the object is not part of a queue the behaviour is undefined.

Definition at line 270 of file Queue.h.

4.2.3.2 QueueObject* MacDspBiosSupport::QueueObject::prev ( ) const [inherited]

Returns

the previous object in the Queue or the Head node for the Queue.

Note

If this is the first object in a queue then the queue's head is retrieved.
If the object is not part of a queue the behaviour is undefined.

Definition at line 276 of file Queue.h.

4.3 MacDspBiosSupport::Queue< OBJECT > Class Template Reference

Implements a type-safe queue that can be used by multiple tasks/interrupts.

Inherited by MacDspBiosSupport::BlockingQueue< OBJECT >.

Public Member Functions

- Queue ()
  Class constructor.
- virtual ~Queue ()
  Class destructor.
- virtual void push (OBJECT *obj)
  Push the object into the queue - releasing ownership of the object.
- virtual void unsafe_push (OBJECT *obj)
  Push the object into the queue - releasing ownership of the object.
- const OBJECT & front () const
  Returns a const refernce to the first object in the queue.
- OBJECT & front ()
  Returns a refernce to the first object in the queue.
- OBJECT * pop ()
  Get an object from the queue.
- OBJECT * unsafe_pop ()
  Get an object, without turning off interrupts.
- void remove (OBJECT *obj)
  Remove an object from its current queue.
- bool empty () const
  Returns true if the queue is empty.

4.3.1 Detailed Description

template<class OBJECT>class MacDspBiosSupport::Queue< OBJECT >

Note

this class is based upon the DSP/BIOS QUE_Obj type and associated functions

Definition at line 67 of file Queue.h.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 template<class OBJECT > MacDspBiosSupport::Queue< OBJECT >::Queue ( )

Note

Initializes the queue.

Definition at line 77 of file Queue.h.

4.3.2.2 template<class OBJECT > virtual MacDspBiosSupport::Queue< OBJECT >::~Queue ( ) [virtual]

Note

Virtual destructors are recommended for all classes.

Definition at line 89 of file Queue.h.

4.3.3 Member Function Documentation

4.3.3.1 template<class OBJECT > virtual void MacDspBiosSupport::Queue< OBJECT >::push (OBJECT * *obj*) [virtual]

The object is added to the queue.

Parameters

in	*obj*	the object

Note

This must be virtual to allow a Queue or its derived types to be pointed to by the same type of pointer, AND for a push to be resolved correctly for the queue type being pointed too. For example a derived type may alter a semaphore after adding the object to the queue.

Reimplemented in MacDspBiosSupport::BlockingQueue< OBJECT >.

Definition at line 103 of file Queue.h.

4.3.3.2 template<class OBJECT > virtual void MacDspBiosSupport::Queue< OBJECT >::unsafe_push (OBJECT * *obj*) [virtual]

The object is added to the queue WITHOUT turning off interrupts! You must be certain that no other task will interrupt the call to this function that may also alter the Queue. This function is faster than push().

Parameters

in	*obj*	the object.

Note

This function must be virtual to allow a Queue or its derived types to be pointed to by the same type of pointer, AND for a push to be resolved correctly for the queue type being pointed too.

Definition at line 122 of file Queue.h.

4.3.3.3 template<class OBJECT > const OBJECT& MacDspBiosSupport::Queue< OBJECT >::front ( ) const

The object remains in the queue, and is not protected from popping off of the queue.

Returns

A reference to the first element in the queue.

Warning

The object is not protected from being removed from the queue.
Undefined behaviour for an empty queue.

Definition at line 138 of file Queue.h.

4.3.3.4 template<class OBJECT > OBJECT& MacDspBiosSupport::Queue< OBJECT >::front ( )

The object remains in the queue, and is not protected from popping off of the queue.

Returns

A reference to the first element in the queue.

Warning

The object is not protected from being removed from the queue.
Undefined behaviour for an empty queue.

Definition at line 155 of file Queue.h.

4.3.3.5 template<class OBJECT > OBJECT* MacDspBiosSupport::Queue< OBJECT >::pop ( )

The object is returned from the Queue.

Returns

Pointer to the object or 0 if the queue is empty.

Definition at line 170 of file Queue.h.

4.3.3.6 template<class OBJECT > OBJECT* MacDspBiosSupport::Queue< OBJECT >::unsafe_pop ( )

The object is removed from the queue WITHOUT turning off interrupts! This means you must know that no other process will interrupt the call to this function that may also alter the Queue. This function is faster than pop().

See also

unsafe_push

The obj is returned from the objQueue.

Returns

Pointer to the obj or 0 if one is not present.

Definition at line 196 of file Queue.h.

4.3.3.7 template<class OBJECT > void MacDspBiosSupport::Queue< OBJECT >::remove (OBJECT * *obj*)

Parameters

in	*obj*	This is the object to remove from the queue.

Definition at line 216 of file Queue.h.

4.3.3.8 template<class OBJECT > bool MacDspBiosSupport::Queue< OBJECT >::empty ( ) const

This is safe within an interrupt.

Note

The result of this function may be out-of-date or stale, before the function returns, if another task is able to push an object to the Queue after the internal calculation and before the call returns. To ensure the result is not stale you must disable interrupts or any task or interrupt that may push an object to the queue.

Returns

Returns true if the queue is empty.

Definition at line 235 of file Queue.h.

4.4 MacDspBiosSupport::BlockingQueue< OBJECT > Class Template Reference

Implements a blocking queue, where the blocking occurs on being able to pop an object from the Queue.

Inherits MacDspBiosSupport::Queue< OBJECT >.

Public Member Functions

- BlockingQueue ()

  Class constructor.
- virtual ~BlockingQueue ()

  Virtual destructors are recommended for all classes.
- virtual void push (OBJECT *obj)

  post the object into the queue - releasing ownership of the object
- OBJECT * pop (unsigned timeOut)

  Get an object.
- std::size_t size () const

  Get the size of the list.
- virtual void unsafe_push (OBJECT *obj)

  Push the object into the queue - releasing ownership of the object.
- const OBJECT & front () const

  Returns a const refernce to the first object in the queue.
- OBJECT & front ()

  Returns a refernce to the first object in the queue.
- OBJECT * pop ()

  Get an object from the queue.
- OBJECT * unsafe_pop ()

  Get an object, without turning off interrupts.
- void remove (OBJECT *obj)

  Remove an object from its current queue.
- bool empty () const

  Returns true if the queue is empty.

4.4.1 Detailed Description

template<class OBJECT>class MacDspBiosSupport::BlockingQueue< OBJECT >

This allows a piece of code to wait for an object to be present in a queue before attempting to pop an object from it. This class is based upon the Queue class and therefore a pointer to the Queue class can also be used to point to a BlockingQueue class. This is useful for BufferQueueObjects because the release() function calls the queue's virtual push() function. The virtual push function allows an object to be returned to a blocking queue or queue, and in the blocking queue's case to cause any waiting function to wake.

The BlockingQueue wraps up a QUE_Obj and SEM_Obj, so that they can be used in a type-safe manner.

Definition at line 76 of file BlockingQueue.h.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 template<class OBJECT > MacDspBiosSupport::BlockingQueue< OBJECT >::BlockingQueue ( )

Note

Initialises the queue and semaphores

Definition at line 86 of file BlockingQueue.h.

4.4.3 Member Function Documentation

4.4.3.1 template<class OBJECT > virtual void MacDspBiosSupport::BlockingQueue< OBJECT >::push (OBJECT * *obj*) [virtual]

The obj is added to the message queue and the message count incremented.

Parameters

in	*obj*	the object being pushed onto the queue

Returns

None

Note

This must be virtual to allow a BlockingQueue and a Queue to be pointed to by the same type of pointer, AND for a push to be resolved correctly for the queue type being pointed too.

Reimplemented from MacDspBiosSupport::Queue< OBJECT >.

Definition at line 108 of file BlockingQueue.h.

4.4.3.2 template<class OBJECT > OBJECT* MacDspBiosSupport::BlockingQueue< OBJECT >::pop (unsigned *timeOut*)

The object is returned from the objQueue, when one is present according to a counting semaphore.

Parameters

in	*timeOut*	This is a time to wait before returning without an object. If an object is present the function will return immediately.

Returns

Pointer to the obj or 0 if the function times out without obtaining an object.

Note

IF timeOut == SYS_FOREVER the function will block indefinitely
This function will block if there are no free objs available.

Warning

Inside interrupts semaphores are not updated, and back-to-back interrupts can allow no time for semaphore to be updated. This can cause logic errors if the writer assumes a previous interrupt's result is placed in a BlockingQueue. In those instances use a Queue<>.

Definition at line 134 of file BlockingQueue.h.

4.4.3.3 template<class OBJECT > std::size_t MacDspBiosSupport::BlockingQueue< OBJECT >::size ( ) const

Returns

the size of the list, bearing in mind this may have changed before the result is returned, OR may be out of sync, if DSP/BIOS has not had a chance to update the semaphore object.

Note

Use empty() in preference, to size() in production code. We recommend using size() only for debug messages particularly because semaphores may not be up-to-date.

Definition at line 158 of file BlockingQueue.h.

4.4.3.4 template<class OBJECT > virtual void MacDspBiosSupport::Queue< OBJECT >::unsafe_push (OBJECT * *obj*) [virtual, inherited]

The object is added to the queue WITHOUT turning off interrupts! You must be certain that no other task will interrupt the call to this function that may also alter the Queue. This function is faster than push().

Parameters

in	*obj*	the object.

Note

This function must be virtual to allow a Queue or its derived types to be pointed to by the same type of pointer, AND for a push to be resolved correctly for the queue type being pointed too.

Definition at line 122 of file Queue.h.

4.4.3.5 template<class OBJECT > const OBJECT& MacDspBiosSupport::Queue< OBJECT >::front ( ) const [inherited]

The object remains in the queue, and is not protected from popping off of the queue.

Returns

A reference to the first element in the queue.

Warning

The object is not protected from being removed from the queue.
Undefined behaviour for an empty queue.

Definition at line 138 of file Queue.h.

4.4.3.6 template<class OBJECT > OBJECT& MacDspBiosSupport::Queue< OBJECT >::front ( ) [inherited]

The object remains in the queue, and is not protected from popping off of the queue.

Returns

A reference to the first element in the queue.

Warning

The object is not protected from being removed from the queue.
Undefined behaviour for an empty queue.

Definition at line 155 of file Queue.h.

4.4.3.7 template<class OBJECT > OBJECT* MacDspBiosSupport::Queue< OBJECT >::pop ( ) [inherited]

The object is returned from the Queue.

Returns

Pointer to the object or 0 if the queue is empty.

Definition at line 170 of file Queue.h.

4.4.3.8 template<class OBJECT > OBJECT* MacDspBiosSupport::Queue< OBJECT >::unsafe_pop ( ) [inherited]

The object is removed from the queue WITHOUT turning off interrupts! This means you must know that no other process will interrupt the call to this function that may also alter the Queue. This function is faster than pop().

See also

unsafe_push

The obj is returned from the objQueue.

Returns

Pointer to the obj or 0 if one is not present.

Definition at line 196 of file Queue.h.

4.4.3.9 template<class OBJECT > void MacDspBiosSupport::Queue< OBJECT >::remove (OBJECT * *obj*) [inherited]

Parameters

in	*obj*	This is the object to remove from the queue.

Definition at line 216 of file Queue.h.

4.4.3.10 template<class OBJECT > bool MacDspBiosSupport::Queue< OBJECT >::empty ( ) const [inherited]

This is safe within an interrupt.

Note

The result of this function may be out-of-date or stale, before the function returns, if another task is able to push an object to the Queue after the internal calculation and before the call returns. To ensure the result is not stale you must disable interrupts or any task or interrupt that may push an object to the queue.

Returns

Returns true if the queue is empty.

Definition at line 235 of file Queue.h.

Appendix A

Example Program Listing

The following code is an example program that uses the USB driver to create three ports. The first port loops back all data sent to it, whilst the second and third ports loop the data they receive back via the other port converting all the data to upper and lower case respectively.

```
/*-----------------------------------------------------------------------------
  include files
-----------------------------------------------------------------------------*/

#include <stdint.h>
#include <std.h>

#include <cctype>

#include "MacDspBiosSupport/BufferQueue.h"

#include "UsbDriver/CdcAcmUsbController.h"


/*-----------------------------------------------------------------------------
  manifest constants
-----------------------------------------------------------------------------*/
const uint8_t NUMBER_OF_PORTS(3);

const uint32_t BUFFER_SIZE( 1024 * 8 );

const std::size_t NUMBER_OF_MESSAGE_BUFFERS_PER_PORT( 9 );

/*-----------------------------------------------------------------------------
  prototypes
-----------------------------------------------------------------------------*/
extern "C" void usbTsk();

void usb0ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
      rxBuffer );
void usb1ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
      rxBuffer );
void usb2ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
      rxBuffer );
```

```
void usbPowerSleepController( void* ignored, bool enable );


/*-----------------------------------------------------------------------------
  global variables
-----------------------------------------------------------------------------*/
MacDspBiosSupport::BufferQueue rxBufferQueues[ NUMBER_OF_PORTS ];

MacDspBiosSupport::BufferQueueObject bufferQueueObjects[
      NUMBER_OF_MESSAGE_BUFFERS_PER_PORT * NUMBER_OF_PORTS ];

#pragma DATA_ALIGN( 128 )
char bufferMemory[ BUFFER_SIZE * NUMBER_OF_MESSAGE_BUFFERS_PER_PORT *
      NUMBER_OF_PORTS ];


UsbDriver::CdcAcmUsbController usbController;


/*-----------------------------------------------------------------------------
  functions
-----------------------------------------------------------------------------*/


/*=============================================================================
 main
-----------------------------------------------------------------------------*/
int main()
{
   int bufferNumber(0);
   for( int port = 0; port < NUMBER_OF_PORTS; ++port )
   {
      for( int buffersAdded = 0 ; buffersAdded <
      NUMBER_OF_MESSAGE_BUFFERS_PER_PORT; ++buffersAdded, ++bufferNumber )
      {
         bufferQueueObjects[ bufferNumber ].addr        = (uint8_t*)&
      bufferMemory[ bufferNumber * BUFFER_SIZE ];
         bufferQueueObjects[ bufferNumber ].size        = BUFFER_SIZE;
         bufferQueueObjects[ bufferNumber ].bytesUsed   = 0;
         bufferQueueObjects[ bufferNumber ].returnQueue = &rxBufferQueues[port]
      ;

         rxBufferQueues[port].push( &bufferQueueObjects[ bufferNumber ] );
      }
   }

   return 0;
}


/*=============================================================================
 usbTsk
-----------------------------------------------------------------------------*/
void usbTsk()
{
   // Set the serial number in the USB descriptor.
   uint64_t serialNumber( 0x123456789ABCDEF0ULL );

   SEM_Obj connected;
   SEM_new( &connected, 0 );
   usbController.registerConfiguredStatusChangedSemaphore( 0, &connected );
```

```
   if( 1 <= NUMBER_OF_PORTS )
   {
      usbController.registerRxCallback( 0, usb0ExampleCallback, (void*)&
      usbController );
   }
   if( 2 <= NUMBER_OF_PORTS )
   {
      usbController.registerRxCallback( 1, usb1ExampleCallback );
   }
   if( 3 <= NUMBER_OF_PORTS )
   {
      usbController.registerRxCallback( 2, usb2ExampleCallback );
   }

   usbController.initialize( NUMBER_OF_PORTS,
                             usbPowerSleepController,
                             0,
                             true,
                             19,
                             7,
                             0,

      UsbDriver::CdcAcmUsbController::NO_DOUBLE_BUFFERING,

      UsbDriver::CdcAcmUsbController::NO_DOUBLE_BUFFERING,
                             true,
                             true,
                             serialNumber );


   SEM_pendBinary( &connected, SYS_FOREVER );


   while( 1 )
   {
      for( int port = 0; port < NUMBER_OF_PORTS; ++port )
      {
         MacDspBiosSupport::BufferQueueObject *bqo =
            rxBufferQueues[ port ].pop();

         if( 0 != bqo )
         {
            bqo->bytesUsed = 0;
            usbController.submitRead( port, bqo );
         }
      }
   }
}

/*============================================================================
 usb0ExampleCallback
----------------------------------------------------------------------------*/
void usb0ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
      rxBuffer )
{
   UsbDriver::CdcAcmUsbController* usbController = (
      UsbDriver::CdcAcmUsbController*)ptr;

   usbController->submitWrite( 0, rxBuffer );
}
```

```
/*=============================================================================
 usb1ExampleCallback
-----------------------------------------------------------------------------*/
void usb1ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
      rxBuffer )
{
   // convert to uppercase
   for( int currentCharacter = 0;
        currentCharacter < rxBuffer->bytesUsed;
        ++currentCharacter )
   {
      // convert the current character to uppercase
      int upperChar = std::toupper( rxBuffer->addr[ currentCharacter ] );
      if( 0 <= upperChar )
      {
         rxBuffer->addr[ currentCharacter ] = (char)upperChar;
      }
   }

   usbController.submitWrite( 2, rxBuffer );
}


/*=============================================================================
 usb1ExampleCallback
-----------------------------------------------------------------------------*/
void usb2ExampleCallback( void *ptr, MacDspBiosSupport::BufferQueueObject*
      rxBuffer )
{
   // convert to lowercase
   for( uint8_t* currentAddr = rxBuffer->addr;
        currentAddr < ( rxBuffer->addr + rxBuffer->bytesUsed );
        ++currentAddr )
   {
      // convert the character at the current address to lowercase
      int lowerChar = std::tolower( *currentAddr );
      if( 0 <= lowerChar )
      {
         *currentAddr = (char)lowerChar;
      }
   }

   usbController.submitWrite( 1, rxBuffer );
}
```