# Usb Manager

## Table of Contents

# 1 UsbChannelsManager.

## 1.1 General.

1.1.1 The channels manager is the main component handling all usb transactions.

1.1.2 Each transaction is handled by an attached channel.

1.1.3 Each attached channel is associated with an active '*UsbChannelSingle*' object.

1.1.4 The code supports up to 8 active (attached) channels.

1.1.5 Each channel instance must be a member of a single device.

1.1.6 Each active channel is associated with a distinct port. A port may be associated with multiple channels.

1.1.7 The channels manager holds the following usb modules -

1.1.7.1 'UsbPortManager' – The (platform dependent) physical port interaction.

1.1.7.2 'UsbChannelsManager' – The (chip dependent) channels transactions interfae.

1.1.7.3 'UsbDevicesManager' – The system devices manager.

1.1.8

# 2 UsbChannelSingle.

## 2.1 General.

2.1.1 The channel class is a state based object.

2.1.2 Each channel instance must be a member of a single device.

2.1.3 Each active channel is associated with a distinct port. A port may be associated with multiple channels.

2.1.4 Channels may be of several types:

2.1.4.1 *'UsbChannelSingleControl'* – A special channel that implements both input and output. Each transaction has two phases. First phase is sending an 8 bytes standard control query. Second phase is receiving the device reply.

2.1.4.2 *'UsbChannelSingleInterruptIn'* – An input channel that queries device periodically.

2.1.4.3 *'UsbChannelSingleBulkOut'* – An output channel that sends packets to devices.

2.1.5 A state change may be triggered by an interrupt action, a device request, or port error.

2.1.6 Each transaction is initiated by the owner device. The device sets up the transaction parameters, but does not start the transaction. The channels manager is in charge of starting a transaction.

2.1.7 Each channel tries to send / receive one packet each start of frame, until the transaction terminates.

2.1.8 During transaction setup, the device determines how many frames to wait before transaction initiates. This delay must be at least 1 frame.

2.1.9 The frame delay is a main synchronization component between the main process, and the interrupts handler.

2.1.10 A device may move the channel to error state, or to detached. Error state would cause the associated port to be disabled.

## 2.2 Channel state.

2.2.1 Active channel state is maintained by the channels manager. It consists of physical state, as well as associated port pointer.

2.2.2 The ‘*UsbChannelSingle*’ object holds a reference to table in the channels manager. A non negative reference is an active channel. A value of ‘*-1*’ is a detached channel. Any value under ‘*-1*’ is considered an error state.

2.2.3 Channel states are as follows:

2.2.3.1 ‘*USB_CHANNEL_DETACHED*‘ – Channel is not active. (channel index is ‘-1’).

2.2.3.2 ‘*USB_CHANNEL_IDLE*’ – Channel has no active transaction.

2.2.3.3 ‘*USB_CHANNEL_ACTIVE*’ – Channel is in the middle of an active transaction.

2.2.3.4 ‘*USB_CHANNEL_READY_IN*’ – Channel has finished the receive action, but has not notified parent object yet. Treated as an active state.

2.2.3.5 ‘*USB_CHANNEL_READY_OUT*’ - Channel has finished the send action, but has not notified parent object yet. Treated as an active state.

2.2.3.6 ‘*USB_CHANNEL_NACK*’ - Channel has finished the receive action with a ‘nack’ answer, but has not notified parent object yet. Treated as an active state. The ‘nack’ state is valid only for a receive channel, before any data was returned.

2.2.3.7 ‘*USB_CHANNEL_ERROR*’ – Two cases. One is the single channel registering an error (moving the channel index to a negative number under ‘-1’), and waiting for the channels manager to notice and detach. The other is the channels manager notifying the single channel about an error, causing the single channel to detach (and move immediately to detached state).

## 2.3 ‘*tick*’ action.

2.3.1 Active channel tick is invoked by the *UsbChannelsManager.*

2.3.2 A detached channel would not invoke ‘*tick*’.

2.3.3 The ‘*tick*’ action would return ‘*true*’ if all is good.

2.3.4 The ‘*tick*’ action would return ‘*false*’ on error. The channel would move state to detached (‘*m_channel_index*’ is set to -1).

2.3.5 An error may happen when the channels manager notifies about a physical error, or when the device moved related channel to error state.

# 3 UsbPortManager.

## 3.1 States.

3.1.1 The port manager is the module that manages all the actual physical usb port.

3.1.2 The port manager is a state based object.

3.1.3 All state changes are done through the main process using the ‘tick’ method, invoked by the ‘*ModuleManagerUsb*’ object. The state changes do not use the interrupts mechanism. We do a constant polling instead.

3.1.4 State ‘*USB_PORT_POWERED_DOWN*’:

3.1.4.1 Disable the external usb power supply.

3.1.4.2 Powered down state disables the usb board power regulator. The usb power may remain high if a connected device has internal power source.

3.1.4.3 When entering the powered down state, commit a physical core reset.

3.1.4.4 When leaving the powered down state, setup the fifo queues parameters.

3.1.4.5 Power down state is active for 1 second, after which we automatically move to ‘*USB_PORT_DISCONNECTED*’

3.1.5 State ‘*USB_PORT_DISCONNECTED*’:

3.1.5.1 Waits for a device connection.

3.1.5.2 When a device is connected, move state to ‘*USB_PORT_ENUMERATION_PENDING*’, with timeout of 200 ms (before initiating enumeration process).

3.1.5.3 If detecting over-current on the usb power, move state to ‘*USB_PORT_POWER_DOWN_PENDING*’.

3.1.6 State ‘*USB_PORT_ENUMERATION_PENDING*’:

3.1.6.1 Enumeration pending is active for 100 ms, after which we automatically initiate the enumeration process.

3.1.6.2 If detecting over-current on the usb power, move state to ‘*USB_PORT_POWER_DOWN_PENDING*’.

3.1.7 State ‘*USB_PORT_RESET_ACTIVE*’:

3.1.7.1 Wait for an external trigger to clear the reset signal and move to ‘*USB_PORT_CONNECTED*’ state. Set timeout to 200 ms.

3.1.7.2 When detecting over-current on the usb power, move state to ‘*USB_PORT_POWER_DOWN_PENDING*’.

3.1.8 State ‘*USB_PORT_CONNECTED*’:

3.1.8.1 Wait for a channel enable signal.

3.1.8.2 If channel is enabled, move state to ‘*USB_PORT_ENABLED*’.

3.1.8.3 If reached connection timeout, move state to ‘*USB_PORT_POWER_DOWN_PENDING*’.

3.1.8.4 When detecting over-current on the usb power, move state to ‘*USB_PORT_POWER_DOWN_PENDING*’.

3.1.9 State ‘*USB_PORT_ENABLED*’

3.1.9.1 Important – this is the only state where the usb interrupts are active.

3.1.9.2 When an enabled device enters a logical error state, it would physically disable the port.

3.1.9.3 Immediately after entering the enabled state, ***unmask* usb interrupts**.

3.1.9.4 Prior to leaving the enabled state, ***mask* usb interrupts**.

3.1.9.5 If detecting a physical disable action, move to ‘*USB_PORT_POWER_DOWN_PENDING*’.

3.1.9.6 When detecting over-current on the usb power, move state to ‘*USB_PORT_POWER_DOWN_PENDING*’.

3.1.10 State ‘*USB_PORT_POWER_DOWN_PENDING*’

3.1.10.1 Disable the external usb power regulator.

3.1.10.2 Disable all physical active channels.

3.1.10.3 Commit a core reset action (flushes all fifo channels).

3.1.10.4 Move to ‘*USB_PORT_POWERED_DOWN*’ with timeout of 1 second.

# 4 UsbDevicesManager.

## 4.1 General.

4.1.1 The usb devices manager is the class managing all active and non active devices.

4.1.2 Each usb device must be registered with this manager when initiating system.

4.1.3 The usb devices manager invokes a ‘tick’ action for each active device.

4.1.4 The devices manager manages all the enumeration process for new active ports, through the ‘*UsbDeviceSingleEnumerator*’ classs.

4.1.4.1 The devices manager is the owner of ‘*UsbDeviceSingleEnumerator’.*

4.1.4.2 The devices manager is the owner of ‘*UsbDeviceSingleHub*’ (we currently support only a single hub, up to 4 ports).

4.1.4.3 The usb devices manager allocates a unique address for each new active port.

# 5 UsbDeviceSingle.

## 5.1 General.

5.1.1 This class is the base for every active usb device.

5.1.2 Each device would register with the devices manager as part of the constructor.

5.1.3 Each device class participates in the enumeration process by accepting a queried configuration, participating in configuration, and supporting the periodic tick action.

5.1.4 Each device may hold any number of active channels (but at least one).

## **5.2** States.

5.2.1 '*USB_DEVICE_SINGLE_CLOSED*' – The device handler is closed, and may be associated to any newly enable port.

5.2.2 '*USB_DEVICE_SINGLE_CONFIGURING*' – The device handle is assoicated to an enabled port (after receiving a compatible configuration). The object has a unique ep address, and is currently configuring the attached device through the control channel.

5.2.3 '*USB_DEVICE_SINGLE_ACTIVE*' – The device is active, invoking the virtual '*deviceTick*' method each system tick.

5.2.4 '*USB_DEVICE_SINGLE_ERROR*' – The device has entered the error state for some reason, and would close on the next system tick. A device may enter the error state when the associated port is disabled, when any member channel is detached, or when the attached device returned an unexpected answer.

## **5.3** Sub-classes.

5.3.1 '*UsbDeviceSingleHid*' – the hid boot devices. Either a mouse, or a keyboard.

5.3.2 '*UsbDeviceSinglePrinter*' – A receit printer using the ESC/POS protocol.

5.3.3 '*UsbDeviceSingleHub*' – Supports upto 4 hub ports. If the exist more than 4 ports, supports only ports 1 to 4.

# 6 Supported devices

## 6.1 ‘UsbDeviceSingleHub’.

### 6.1.1 The usb de

## 6.2 ‘UsbDeviceSingleHid’.

### 6.2.1 The usb de

## 6.3 ‘UsbDeviceSinglePrinter’.

### 6.3.1 The usb de