USB Support On Non-Linux STBs

Contents

Chapter 1. Preface

Purpose of this Document

Content

This document provides technical information about the USB feature integration for non-Linux STBs

Audience

This document is intended for

- software engineers who are integrating the USB stack on new platforms.
- product owners of HDI projects that need to support USB mass storage and return path using USB.

Assumptions, Dependencies and Constraints

It is assumed that the reader is familiar with Linux and uCos operating systems, the C programming language, and the architecture and operation of an STB.

Within this document, the following words and phrases have the meanings stated here:

- "**shall**" is used to indicate a mandatory requirement or behaviour, which must be met;
- "**should**" is used to indicate a recommendation, which is desirable, but not mandatory;
- "**may**" is used to indicate a behaviour which is acceptable, but not required;
- "**will**" does not indicate a requirement, but can be used in a statement of fact supporting a requirement, or to indicate a future intention.

References

Table 1 lists documents and other reference sources containing information that may be essential to understanding topics in this document.

Table 1. References

No.	Designation	Title
1	USB20CV x64-bit	USB 2.0 Specification
2	Document 124945r00JB	Jungo USBware Documents
3	HDI-ICD-171	HDI4 specification

Terminology

Table 2 provides a short glossary of any terms crucial to the understanding of this document, and lists the acronyms and abbreviations used in the document

Table 2. Terminology

Term	Definition
USB	Universal Serial Bus
STB	Set-Top Box
HDI	Hardware Device Interface
CDI	Common Device Interface
PnP	Plug and Play
EHCI	Enhanced Host Controller Interface
OHCI	Open Host Controller Interface
UHCI	Universal Host Controller Interface
XTVFS	XTV File System
HAL	Hardware Abstraction Layer

Chapter 2. The USB host stack

This section provides general information on USB host stacks.

To connect and use a USB device on an STB (or any other device), a USB host stack is needed to interact with the device that is connected through a USB port. The USB host stack does the following:

- Detection of USB device insertion/removal
- Communication with the hardware drivers of the box to interact with the USB device that is connected.
- Collection of information about the type of the device that is connected.
- Management of USB standard requests via control transfers
- Management of data flow between host and devices
- Supports EHCI, OHCI and non-standard USB host controllers. It optionally supports hubs to enable management of multiple USB devices

Chapter 3. Linux and non-Linux systems

The USB feature implementation depends on the architecture of the STB. This is because the operating systems (OS) used by STBs can be different. There are two categories of STB architectures:

- STBs based on the CDI (Common Device Interface) architecture use Linux OS. Linux already has an internal USB host stack. Therefore, additional support is not required.
- STBs based on the HDI (Hardware Device Interface) architecture use uCOS OS which does have USB stack support. Therefore, an external USB stack needs to be installed in such systems to interact with the USB devices. This external USB stack is provided by Jungo to be integrated into the STB software stack.

  The below diagram dipicts the architecture of the Jungo USB stack on HDI based systems.

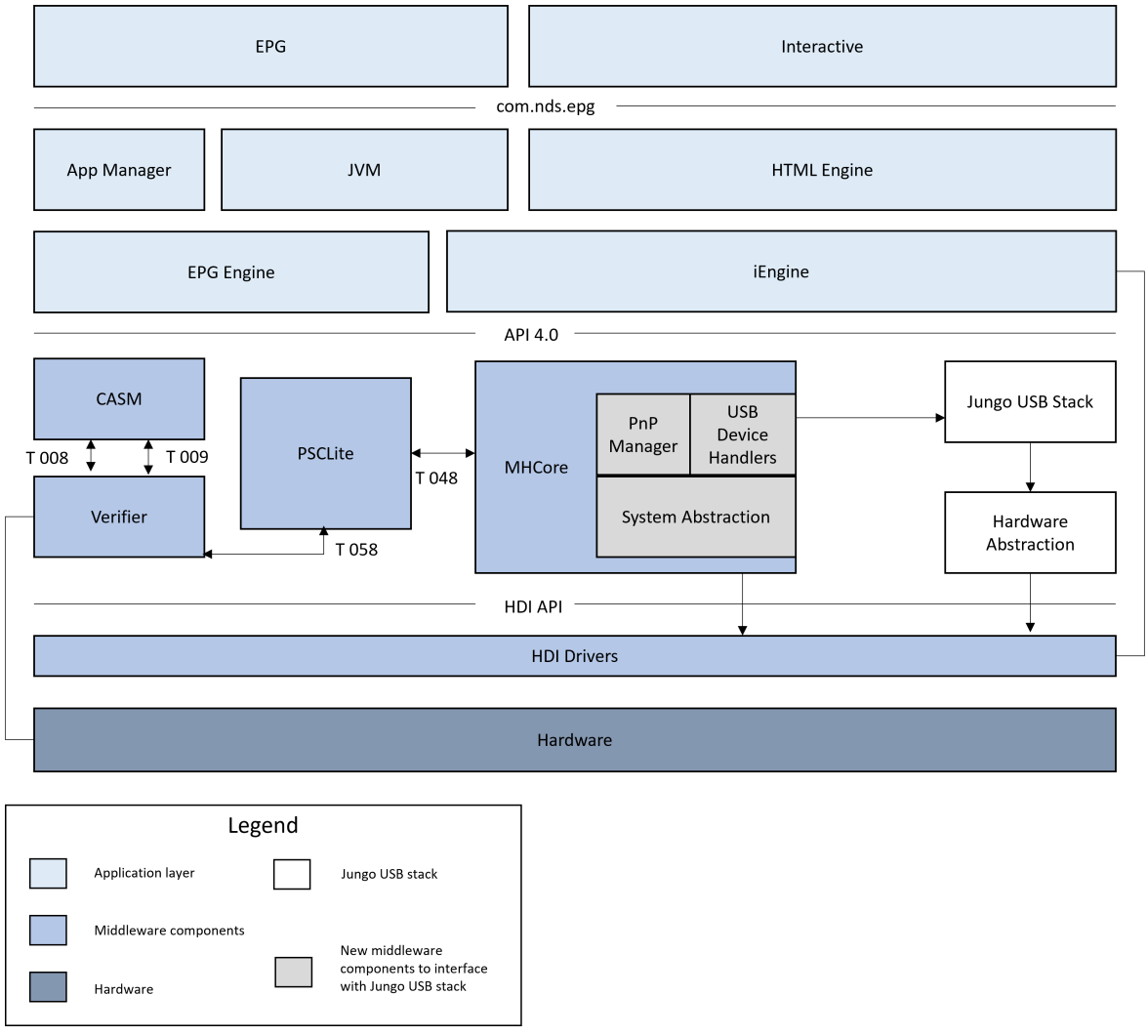

Figure 1. HDI based STB architecture with Jungo USB stack

The **Hardware Abstraction layer** is needed to make the Jungo USB stack independent of the hardware being used. After implementation of the HDI functions, this layer will not be required as the HDI function calls will be made directly and hardware details will be handled by the HDI.

The **System Abstaction layer** will be present in the Core component to achieve abstraction between the USB stack used, the file system used and the IP stack used.

The **PnP Manager** component will implement functionalities to communicate with the USB device that is connected. These functionalities will be called by the Application depending on the device and user preference. The module is discussed in detail here *(on page 11)*

The **USB Device handler** component will reside in the Core module. It will directly interact with the Jungo USB stack to carry out various I/O operations of the USB.

Chapter 4. Jungo USB Stack

Overview

Jungo provides a complete, small footprint USB Host Stack. This includes Application Programming Interfaces (APIs) and services that enable software and device manufacturers to incorporate standard USB Host connectivity in their embedded device. USB Host stack also enables the use of multiple USB classes on the embedded hardware platform.

The stack supports the following USB versions:

- USB 1.1: Full Speed rate of 12 Mbit/s (1.5 MB/s)
- USB 1.0: Low Speed rate of 1.5 Mbit/s (187.5 kb/s)
- USB 2.0: High-Speed rate of 480 Mbit/s (60 MB/s)

It provides support for the following CPUs:

- ARM
- MIPS
- x86
- ATOM
- x86_64

Porting Layer

Driver Level

The HDI driver has to provide the following functionalities to the Jungo USB stack

USB Controllers

The USB stack is generic to all platforms and the implementation is done in accordance with the USB specification (EHCI, OHCI and UHCI). The USB stack should know the controllers available in the platform and register an offset to initialize the stack.

The HDI driver provides the below interface to pass the controller and register information to USB stack. It is done during the HDIInit() call.

```
HDI_STATUS CORESetUSBHostController( DEVICE_NO device_no, HDI_USBHC_TYPE hc_type, ULONG *reg_address_p)
```

Note:

- If the driver does not invoke this call, the USB stack will not be initialized.
- The HDI can invoke this function more than once to support more than one controller.

Memory

The USB stack requires both cache and non-cache memory.

- **Non-cache memory:**

  The USB controller requires non-cache memory for USB read and write operations. The below interfaces must be implemented by the driver to bring up the USB stack. These calls must be mapped in the Jungo memory porting layer.

  ```
  HDI_STATUS HDIGetNonCachedRam(void **non_cached_virt_p, void **non_cached_phys_p,ULONG non_cached_size)
  ```

  ```
  HDI_STATUS HDIFreeNonCachedRam(void *non_cached_virt_p)
  ```

  > **Note:**
  > - The stack requires atleast 256KB of non-cache memory.
  > - The USB stack will not work correctly, if virtual address or physical address is incorrect. The controller will throw spurious interrupts when the physical address is incorrect.

- **Cache Memory:**

  Cache memory is provided by the middleware. The USB stack internally maintains a memory pool and the USB allocation and de-allocation is done from this pool by the USB stack. The USB stack shall not access any external memory.

  The middleware creates a separate partition for USB stack cache memory operation. It provides the partition allocation and de-allocation interfaces. These interfaces are then mapped to the Jungo memory porting layer.

  The stack requires atleast 256KB cache memory. This memory shall increase when project wants to support more USB classes.

Interrupts

The stack handles interrupts for USB operations such as attach, detach and I/O. To make the stack common across all platforms, HDI driver provides the below interface to associate with the interrupt.

```
HDI_STATUS HDIInitUSBHostController(DEVICE_NO device_no, HDI_USB_FP function_p)
```

USB Stack Tasks

USB stack uses five tasks for USB operations. The task details are listed in the below table. The middleware has one interface task to initialize the USB stack and also for future use.

Table 3. USB Task details

No.	Task	uCOS Priority
1	Controller	67
2	DSR	68
3	Core	69
4	Driver	70
5	Other (debug thread)	71

Porting Interface

Threads and IPC interfaces

The USB stack requires operating system interfaces for thread creation and inter-process communication. To make the USB stack generic, the Jungo OS layer defines the interfaces to be implemented by the porting layer and the middleware defines another generic interface (X_NDS_OS_..) which should be used by the porting layer.

> **Note:**
> The middleware interfaces are implemented using uCOS calls. No further porting should be required when integrating the USB stack to a new platform which uses uCOS.

Memory

The USB stack memory porting layer defines two set of interfaces for memory allocation and deallocation operations.

1. Allocation and Deallocation from cache memory:
    a. os_malloc()
    b. os_free()

> **Note:**
> A separate USB partition of size 256K is created to handle this.

2. Allocation and Deallocation from non-cache memory:
    a. os_dma_alloc()
    b. os_dma_free()

> **Note:**
> - HDI calls are re-mapped to get the allocated and deallocated non-cache memory.
> - No further porting should be required on this because the HDI has already abstracted this implementation.

Bus Access

Jungo USB stack porting layer provides a set of interfaces to be implemented by the porting layer for USB bus operations. However, the bus implementation differs from one platform to another.

Platform specific HAL layer is added in the porting layer for USB bus operations and the Jungo Bus porting layer invokes the platform specific functions implemented in HAL layer. The following platforms are support by the HAL layer :

- ST-5202
- ST-5228
- BCM-7454

Chapter 5. Implementation details

The following components need to be implemented in the middleware to support the USB feature on HDI systems.

Porting the USB Host Stack

The Jungo USB stack needs to be added on non-linux systems only. It will be added as a new separate component called USBWare outside of the Core component folder.

The code tree structure is as follows:

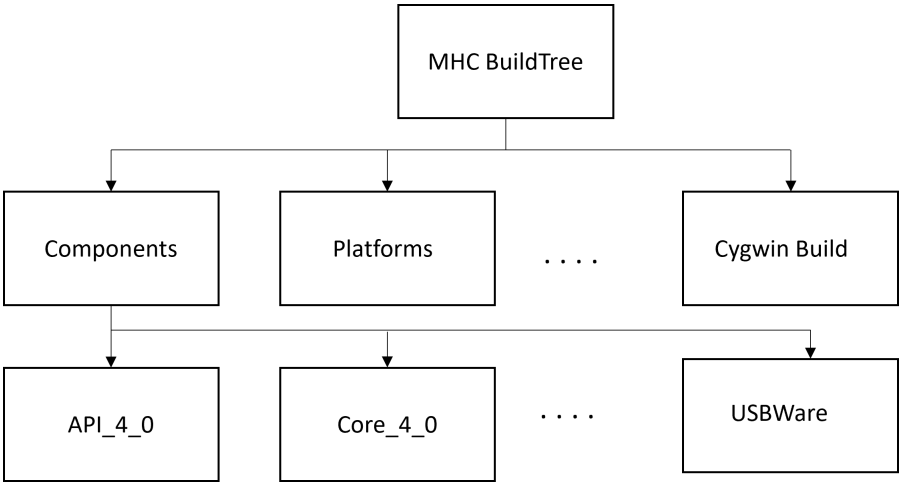

Figure 2. Build Tree structure

The UsbWare component will be compiled externally using the Cygwin tool and the object file will linked thereafter.

These are the steps to build the USBWare component:

1. Run the cygwinbuild batch file available in MHC_BuildTree folder.
2. Change directory to root directory of USBWare.
3. If you have previously built USBWare for a different configuration, the in order to clear the previous build, run the following command:

   ```
   Make distclean
   ```
4. Modify the relevant config/target_cfg.cfg file in order to define the modules you want to build.
5. The following command creates the config.h file for the configurations contained in the target configuration file:

   ```
   Make dist=target_cfgs
   ```
6. Build the USBWare component by running the following command from root directory:

`Make`

7. Copy the created main USBWare library from the USBWare root directory to the target specific driver's folder within the platforms folder.
8. Change directory to root directory in the MHC_BuildTree.
9. Run the following command to build the executable image using all the available libraries:

   `./make_hdi release <TARGET_BOX>`

Hardware abstraction layer for the USB host stack

Currently the Jungo USB host stack needs to call the USB related hardware functions. To make the Jungo stack independent of the platform, the hardware abstraction layer (HAL) is written, which takes care of the hardware details. When the HDI specification for this functionality will be provided, the hardware related tasks will be taken care of by the HDI. This layer can then be removed, as the HDI functions can be directly invoked.

PnP Manager Component

PnPDevMan is the core component of the USB feature. This component takes care of attach and detach notifications to the application, maintains the device structures that tell about the device type, number of devices connected and status of the devices.

The source files for the PnPDevMan are:

1. Apipnp.h

   It contains the data structures that are used by the USB stack component and function declarations for the application interface.
2. Pnpman.c

   It contains implementations for the PnPManTask and device status information. PnPManTask gets events from the USB host stack about the attach or detach state of a USB device.
3. Gfs.c

   It contains the implementation of the generic file system interface for XTVFS/Linux File systems. It has functions such as `APIGFSMount` to mount the USB mass storage device. `APIGFSUnMount, APIGFSOpen, APIGFSRead` etc. are the other similar functions.

The PnP Manager will be added in the Core component structure as a separate folder. It will compiled as a separate Core component. These are the changes to be made to the Core makefiles:

1. In Core_libraries.mk: include the following lines

   LIBRARIES_MAKEDIR_pnpmanager := $(LIBRARIES_MAKEDIR_core_4_0)/pnpmanager/code

   LIBRARIES_OUTDIR_usbware := $(LIBRARIES_MAKEDIR_core_4_0)/pnpmanager/code
2. In the makefile: include the following line

   COMPONENT_SOURCE_LIBRARIES+=pnpmanager

System Abstraction Layer

The System Abstraction Layer is present in the middleware to facilitate the following abstractions:

1. The USB stack used – either Linux or Jungo
2. The File system to be used – either Linux or XTVFS
3. The IP Stack to be used – either Linux or Treck (IP Stack existing in the middleware)

A compilation switch can be used for this purpose. For example, LINUX_USED switch can be introduced. For the CDI architecture boxes, this flag will be defined and Linux calls will be compiled and used in this layer. For HDI architecture boxes this flag will not be defined and Jungo USB host stack calls are made for host stack. Similarly the IP stack which processes the IP packets received from a USB device is different based on the system architecture - Treck is used in HDI boxes and Linux IP stack is used in CDI boxes. These components can also be compiled selectively using the switch.

Chapter 6. Sequence Diagrams

USB Stack Initialization Sequence

The sequence diagram below shows the new controller notification sent by the driver as a trigger to initialize the Jungo USB stack

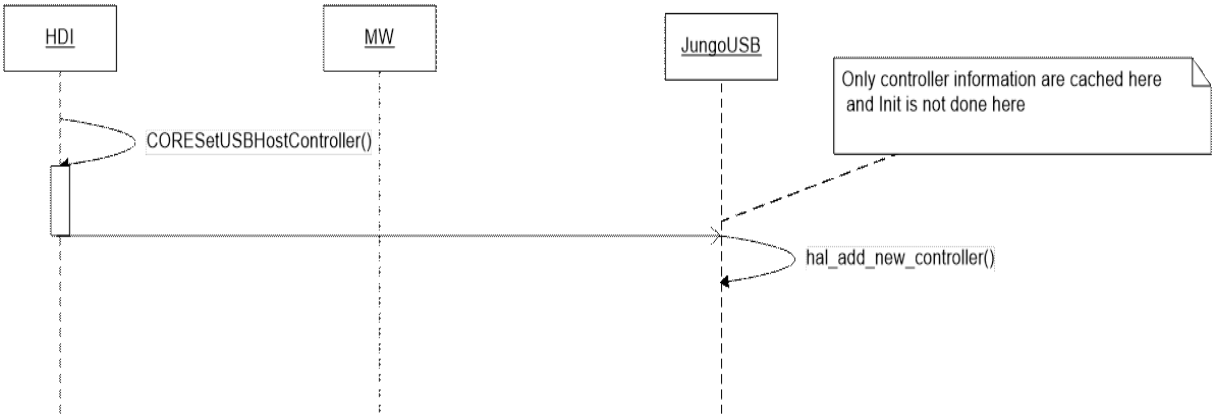

Figure 3. Jungo USB stack initialization by driver

The sequence diagram below shows the complete high-level initalization sequence of Jungo USB stack.

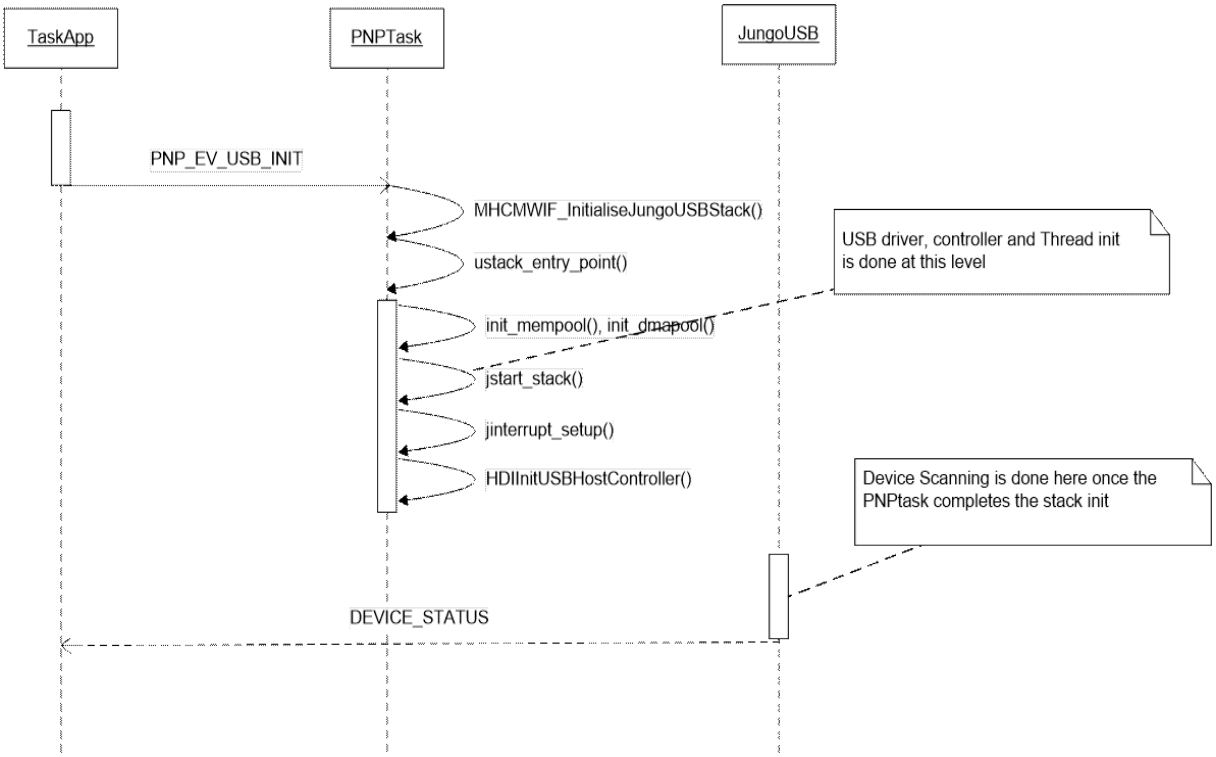

Figure 4. Jungo USB stack complete initialization

USB Interaction Sequences

The sequence diagram below shows the high-level communication between the Jungo USB and the middleware. It is not specific to any USB class.

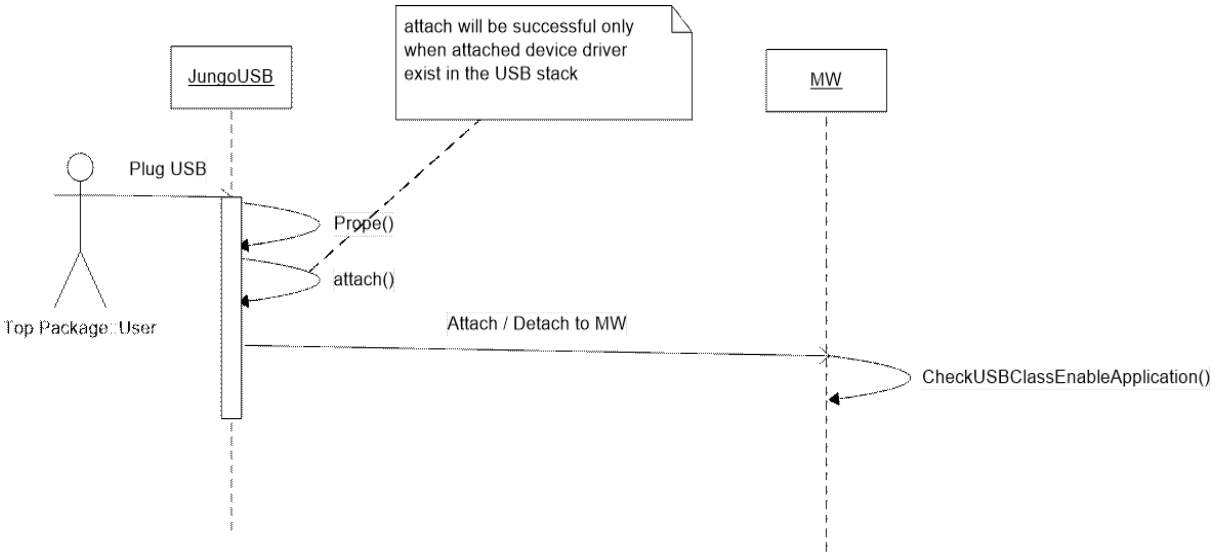

Figure 5. Jungo USB stack interaction with middleware