AT91 USB Composite Driver Implementation

1. Introduction

The USB **Composite Device** is a general way to integrate two or more functions into one single device. It is defined in the *USB Specification Revision 2.0*, as "A device that has multiple interfaces controlled independently of each other".

This document introduces basics for USB **composite device** and gives examples to implement the composite device that has two functions included, based on the Atmel® AT91 SAM Softpack for its ARM® Thumb® based microcontrollers.

To add more than one functions to the host via one USB port, the USB **composite device** works. The available functions are listed as following:

- **MSD: To extend hard disk storage capacity.**
- **HID: USB mouse, USB keyboard, Game controller ...**
- **Audio: USB Speaker or Recorder for host.**
- **CDC: To work as virtual COM port, modems or networking devices such as ADSL or Cable modem.**

For more detailed information, please refer to the Application Notes of the class related device implementations.

Normally when you use USB functions, you will need one USB cable for each function, which means when you use several functions, many cables are needed. The composite device helps to reduce the number of cables used at the same time.

This Application Note details the following composite solutions:

- **HID + MSD:** Adds keys, navigation pads or even handwrite pad with a storage disk.
- **HID + Audio:** Integrates game controller with Audio codec.
- **CDC + MSD:** Add more serial ports or modems when extending the storage of your laptop.
- **CDC + Audio:** Modem and an audio speaker at the same time. Or extend your serial port when you are playing music.
- **CDC + HID:** Extends the interface mostly for laptops, which allows extra serial port and mouse/keyboard.
- **CDC + CDC:** To extends two or more serial ports through one single USB cable, mostly for laptop.

Note that the composite device has several functions that work concurrently. The host can see all of these functions available simultaneously. Some of the cellphones that can manually select their connection types as “Modem” or “USB Disk” is just product with two different USB devices pre-selected and not “composite” one.

AT91 ARM Thumb Microcontrollers

Application Note

6436A–ATARM–01-Jul-09

2. Related Documents

[1] *Universal Serial Bus Specification*, Revision 2.0, April 27, 2000

[2] *Interface Association Descriptor, USB Engineering Change Notice* (ECN)

[2] *Device Class Definitions for Human Interface Devices* (HID), Version 1.11, June 27, 2001

[3] *HID Usage Tables*, Version 1.12, January 21, 2005.

[4] *Universal Serial Bus Mass Storage Class Specification Overview*, Revision 1.2, June 23, 2003

[5] *Universal Serial Bus Mass Storage Class Bulk-Only Transport*, Revision 1.0, September 31, 1999

[6] *Universal Serial Bus Device Class Definition for Audio Devices*, Release 1.0, March 18, 1998

[7] *Universal Serial Bus Class Definitions for Communication Devices*, Version 1.1, January 19, 1999

[8] Atmel Corp., *AT91 USB Device Framework*, 2006 (6263A)

[9] Atmel Corp., *AT91 USB Mass Storage Driver Implementation* (6283A)

[10] Atmel Corp, *AT91 USB HID Driver Implementation* (6273A)

[11] Atmel Corp, *AT91 USB Audio Class Driver Implementation* (...)

[12] Atmel Corp., *AT91 USB CDC Driver Implementation* (6269A)

3. USB Composite Device Basics

3.1 Purpose

The Universal Serial Bus (USB) offers an easy way to connect PC with portable devices and to expand external peripherals. Usually a USB device provides a single function to the host, such as a storage disk, serial RS-232 port, digital microphone, or speakers, so one function occupies one USB port. Nevertheless, to allow several functions through a single physical port, the USB specification mentions two kinds of devices:

- Composite device: a device has multiple logical interfaces controlled independently of each other. A single physical logical address is used for the device.
- Compound device: a separate hub with multiple functions attached.

Composite devices are an efficient solution to reduce the number of hub and USB cables. To give an example: a device being at the same time modem and audio speaker requires only a single USB cable.

3.2 Architecture

3.2.1 Communication flow

Figure 3-3 highlights logical communication flows between several clients (host side) and functions (device side), over a single physical USB connection. The highlighted layer is the function layer. In host side, there is some client software that can operate with the different functions in the same physical device simultaneously. E.g. when a composite device with card reader and printing function is connected to a PC, a new printer device and storage disks may appear for you to explore your pictures on your media card, and to print them with the printer.

3.2.2 Interfaces

Interface descriptors for composite devices correspond to the concatenation of interfaces descriptors defined for each functions available in the system. E.g., a composite with HID+CDC needs three interface descriptors (Section 4.7.3 on page 15), while HID+MSD (Section 4.7.1 on page 13) only needs two.

There are some general rules to assemble the interfaces:

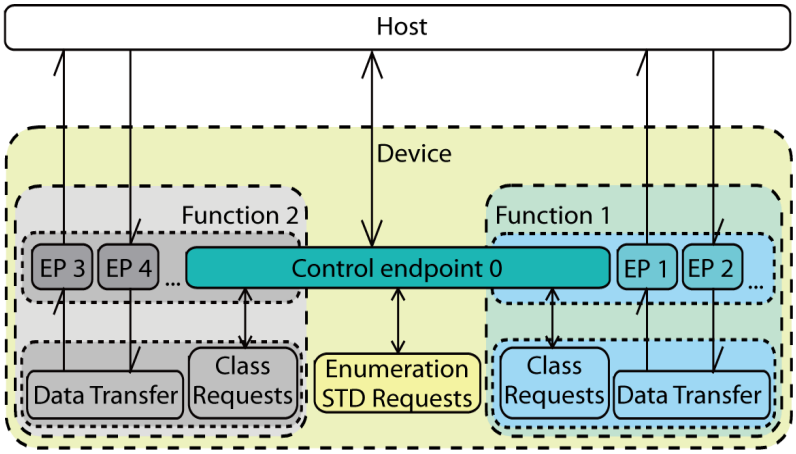
- *bInterfaceNumber* should start from zero and then increase by one;
- the interfaces that use the same function should be placed together, that is, the values of *bInterfaceNumber* should be contiguous;
- if two or more interfaces are associated to one device function (such as CDC), the Interface Association Descriptor (IAD) is used, please refer to Section 3.2.4.1 on page 6 for more details.

3.2.2.1 Composite with SINGLE-interface functions

This kind of composite device includes the functions that have only one interface, such as HID+MSD.

The default endpoint (endpoint 0) is shared by all functions. Each function has one interface only, to handle class-specific requests, control commands and data. The interface and endpoint settings are function specific.

Figure 3-1. USB Composite Device with SINGLE-interface-functions

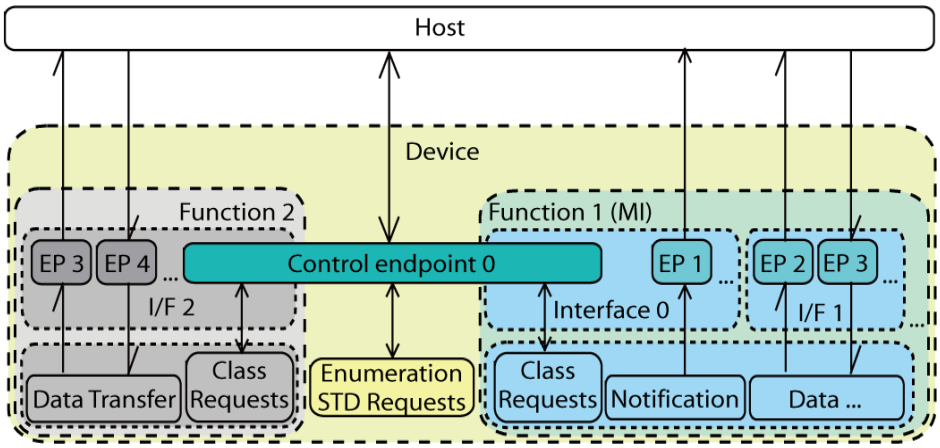


3.2.2.2 Composite with MULTI-interface functions

This kind of composite device includes the functions that are composed of more than one interface, such as CDC class function, Audio Class function, etc..

The default endpoint (endpoint 0) is usually shared by single-interface-function and the control interface of the multi-interface-function. There may be other endpoints such as an interrupt IN endpoint for CDC in the control interface for the function to accept control commands or update status. The interface and endpoint settings are function specific. The multi-interface function also needs a Interface Association Descriptor (IAD, See Section 3.2.4.1 on page 6) to associate its interfaces.

Figure 3-2. USB Composite Device with MULTI-interface-functions



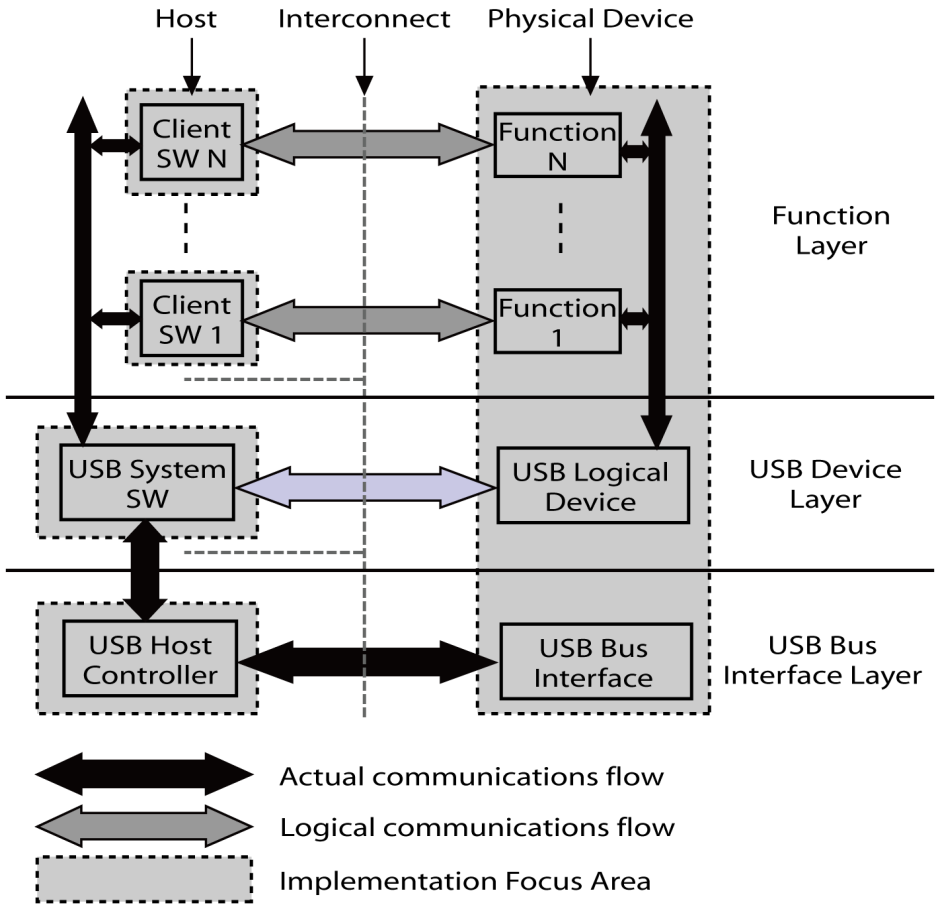
3.2.3 Endpoints

Excluding the default endpoint, composite devices shall declare a number of endpoints equal to the sum of the number of endpoint required for individual function. E.g., a composite with HID+CDC needs six endpoint descriptors (Section 4.7.3 on page 15), while HID+MSD (Section 4.7.1 on page 13) only needs five.

Endpoint 0 (default endpoint) is used for standard, class-specific and vendor-specific requests, depending on the implementation of the functions inside. It can be used to operate on any interface for any device function.

The other endpoints are function-specific, their usage is defined by the class of the function or defined by the vendor.Figure 3-1 and Figure 3-2 are examples for the composite device architecture with two device functions.

Figure 3-3. USB composite composition



3.2.3.1 *Interface & Endpoints compositions for AT91 chip*

Since the number of endpoints and the endpoint fifo size is based on hardware, the assignment of the endpoints should be arranged carefully. Here are some examples on the endpoint allocation for AT91SAM7SE chip, since it has 7 endpoints to use.

Please refer to the AT91SAM product datasheet to get the number and the size of endpoints.

Generally, ISO endpoints should be considered firstly, because they always require large FIFO size and double buffer; the BULK endpoints come the second, which only require a double buffer with transfer speed considered; the last ones are the Interrupt endpoints, which only require a small FIFO and the lowest speed. Please refer to the **Ping-pong mechanism in the USB device** section of the AT91SAM product datasheet.

- **HID**: For HID device function, only two interrupt endpoints are needed, and since the size is small and speed is low, its endpoints are considered as the lowest priority.
- **MSD**: For MSD device function, the high speed transaction, two endpoints, bulk transfer, double/triple bank for performance.

- **Audio**: For Audio device function, the endpoint size should be larger than the audio frame size (here the size is 192 bytes), more than one endpoint, isochronous transfer and at least double bank.
- **CDC**: CDC needs three endpoints excluding the default endpoint to work. Two Bulk endpoints and one Interrupt endpoint. So dual-CDC device will occupy 7 endpoints. The interrupt endpoints can be assigned to single-buffered endpoints.

3.2.3.2 Composite Composition Summary

The following is a summary of the interfaces and endpoints assigned. You can also find reference descriptions and figures in the following sections.

Table 3-1. Interfaces and Endpoints assignments Summary

	HID + MSD	HID + AUDIO	CDC + HID	CDC + MSD	CDC + AUDIO	dual-CDC
Interface 0	HID		CDC0			
Interface 1	MSD	AUDIO Control	CDC0 Data			
Interface 2	-	AUDIO Stream	HID	MSD	AUDIO Control	CDC1
Interface 3	-	-	-	-	AUDIO Stream	CDC1 Data
EP0	Default endpoint for all interfaces and functions.					
EP1	HID Interrupt IN		CDC0 Bulk OUT			
EP2	HID Interrupt OUT		CDC0 Bulk IN			
EP3	-	-	CDC0 Interrupt IN			
EP4	MSD Bulk OUT	-	HID Interrupt IN	MSD Bulk OUT	-	CDC1 Bulk OUT
EP5	MSD Bulk IN	AUDIO ISO OUT	HID Interrupt OUT	MSD Bulk IN	AUDIO ISO OUT	CDC1 Bulk IN
EP6	-	-	-	-	-	CDC1 Interrupt IN
Reference	**Figure 4-3 Section 4.7.1.1**	**Figure 4-4 Section 4.7.2.1**	**Figure 4-5 Section 4.7.3.1**	**Figure 4-6 Section 4.7.4.1**	**Figure 4-7 Section 4.7.5.1**	**Figure 4-8 Section 4.7.6.1**

3.2.4 Specific Descriptors

Since the composite is not for specific class, most of the descriptors for functions are defined in the included class related specifications. Only one descriptor is necessary to define a multi-interface function: the Interface Association Descriptor (IAD).

3.2.4.1 Interface Association Descriptor

The Interface Association Descriptor (IAD) is a new standard descriptor defined in USB Engineering Change Notice, to allow a device to describe which interfaces are associated with the same device function. This allows the Operation System to bind all of the appropriate interfaces to the same driver instance for the function. Please see *USB ENGINEERING CHANGE NOTICE (Title: Interface Association Descriptors)* for detailed information.

The IAD is used to describe that two or more interfaces are associated to the same function. The ‘association’ includes two or more interfaces and all of their alternative setting interfaces. A device must use an IAD for each multi-interfaced device function. An IAD is always returned as part of the configuration information returned by a GetConfigurationDescriptor request. It must be located before the interface descriptors (including all alternate settings) for the interfaces it associates with. All of the interface numbers in a particular set of associated interfaces must be contiguous. Table 3-2 shows the standard *interface association descriptor* includes function

class, subclass and protocol fields. The values in these fields can be the same as the values from interface class, subclass and protocol from any one of the associated interfaces.

Table 3-2.

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes.
1	*bDescriptorType*	1	Constant	INTERFACE ASSOCIATION Descriptor.
2	*bFirstInterface*	1	Number	Interface Number of the first interface that is associated with this function.
3	*bInterfaceCount*	1	Number	Number of contiguous interfaces that are associated with this function.
4	*bFunctionClass*	1	Class	Class code (assigned by USB-IF). A value of zero is not allowed in this descriptor. If this field is FFH, the function class is vendor-specific. All Other values are reserved for assignment by the USB-IF.
5	*bFunctionSubClass*	1	SubClass	Subclass code (assigned by USB-IF). If the *bFunctionClass* field is not set to FFH all values are reserved for assignment by the USB-IF.
6	*bFunctionProtocol*	1	Protocol	Protocol code (assigned by USB-IF). These codes are qualified by the values of the *bFunctionClass* and *bFunctionSubClass* fields.
7	*iFunction*	1	Index	Index of string descriptor which describes this function.

Note: Since this particular feature was not included in earlier versions of the USB specification, there is an issue about how existing USB OS implementations will support devices that use this descriptor. It is strongly recommended that the device implementation utilizing the *interface association descriptor* use the Multi-interface Function Class code in the device descriptor. The Multi-Interface Function Class code is documented on the *http://www.usb.org/developers/docs* web site.

3.2.5 Specific Requests

No special request is used for a composite device, all requests are defined by the device function that is integrated.

3.3 Host Drivers

Usually the Operating System supports the composite device via two part of drivers - composite driver and the function drivers depending on the functions that are integrated. The OS will recognize the USB device as a composite device first then distinguish the included functions and install their driver modules one by one. Then the functions will appear as normal separated devices for OS to access.

Most OSs now include generic drivers for a wide variety of USB classes. This makes developing a function device simpler, since the host complexity is now handled by the OS. Manufacturers can thus concentrate on the device itself, not on developing specific host drivers.

As described before in Section 3.2.4.1 note, the IAD of multi-interface USB devices is a new feature, there may be issues about how existing USB OS implementations will support devices with

IAD. For Linux®, it is supported now. For Windows you can refer to *Support for USB Interface Association Descriptor in Windows*, but patches may needed, only Windows XP with some hot fix or service pack 3 or later updates fully support this feature now.

Here is a brief list of the various function implementations supported by several OSs (for CDC maybe additional .inf file is required to install the device but the driver files themselves are from windows source disk):

- Windows (see *Windows Supported USB Classes* for more)
  - MSD: USB Storage disks
  - HID: USB Keyboard, Mouse, etc.
  - Audio: USB Desktop speaker, recorder.
  - CDC: Abstract Control Model, Remote NDIS ...
- Linux (see *Linux Device Driver Support* for more)
  - MSD: USB Storage disks
  - HID: USB Keyboard, Mouse, etc.
  - CDC: Abstract Control Model
  - CDC: Ethernet Model

Please refer to the sections about the functions or the class implement application notes for details about the OS compatibility on the device driver.

4. Use cases of Composite Device for AT91 chips

This section describes how to implement the general composite device with the class related specification, the device related application notes and *AT91 USB Device Framework* application note, details about whcih can be found in the related application notes and the specifications from USB.org.

4.1 Purpose

The USB composite device here will include two device functions inside. The functions included is pre-defined by some definition in the make file. The supported device functions are listed here:

- HID Keyboard + MSD Disk
- HID Keyboard + Audio Speaker
- CDC Serial Port + HID Keyboard
- CDC Serial Port + MSD Disk
- CDC Serial Port + Audio Speaker
- Dual CDC Serial Port

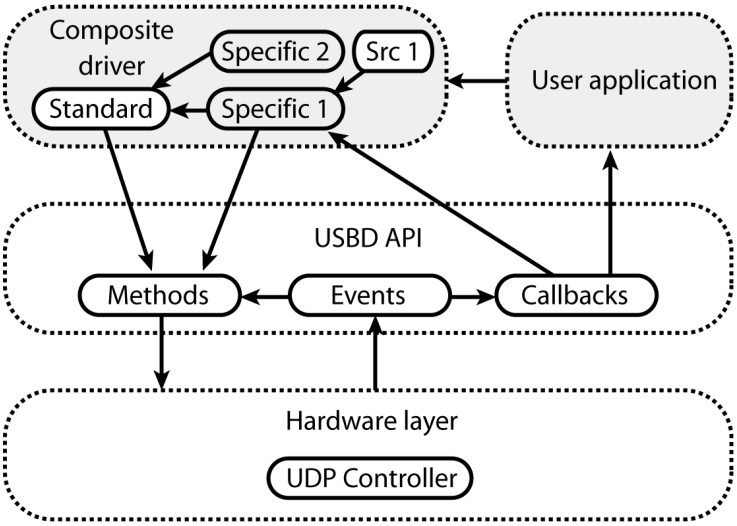
For more detailed information about the device functions inside, please refer to the application notes for the devices:

- Section 5.1, *USB HID Driver Implementation* (6273A)
- Section 3.1, *USB Mass Storage Driver Implementation* (6263A)
- Section 4.1, *USB CDC Driver Implementation* (6269A)
- *USB Audio Driver Implementation*

4.2 Architecture

The AT91 USB Device Framework offers an API which makes it easy to build USB class drivers. The example software provided with this application note is based on this framework, and reuses some resources of the related class device driver. Figure 4-1 shows the application architecture with the framework.

Figure 4-1. Software Architecture Using the AT91 USB Framework



4.3 Descriptors

The descriptors of the device is returned to host via **GetDescriptor** request, to report all information about the device, its interface and endpoint architecture, and its function classes. There are many standard descriptors such as device descriptor, interface descriptor and endpoint descriptor, etc.. See setion 9.5, *Universal Serial Bus Specification, Revision 2.0* for more details.

4.3.1 Device Descriptor

The **Device descriptor** of a composite device is very basic according to the definition in section 9.6.1, *Universal Serial Bus Specification, Revision 2.0*, but some of the fields may have different values based on if there is multi-interface function in the device: *bDeviceClass*, *bDeviceSubClass*, *bDeviceProtocol* set to zero for a device without multi-interface function inside; *bDeviceClass*, *bDeviceSubClass*, *bDeviceProtocol* set to EFH, 02H, 01H for a device with multi-interface function inside, such as CDC or Audio class function, as shown below:

```
// Composite Device Descriptor
const USBDeviceDescriptor deviceDescriptor = {

  sizeof(USBDeviceDescriptor),
  USBGenericDescriptor_DEVICE,
  USBDeviceDescriptor_USB2_00,
#if defined(usb_HIDMSD)
  0x00,
  0x00,
  0x00,
#else
```

```
  0xEF,// MI
  0x02,//
  0x01,//
#endif
  BOARD_USB_ENDPOINTS_MAXPACKETSIZE(0),
  COMPOSITEDDriverDescriptors_VENDORID,
  COMPOSITEDDriverDescriptors_PRODUCTID,
  COMPOSITEDDriverDescriptors_RELEASE,
  0, // No string descriptor for manufacturer
  1, // Index of product string descriptor is #1
  0, // No string descriptor for serial number
  1 // Device has 1 possible configuration
};
```

Note that the Vendor ID is a special value attributed by the USB-IF organization. The product ID can be chosen freely by the vendor.

The HID and MSD class functions are all single-interface functions, so the corresponding fields are set to zeros.

4.3.2 Configuration Descriptor

It's just standard one as defined in section 9.6.3, *Universal Serial Bus Specification, Revision 2.0*, with *wTotalLength* set to the total length of all standard and function specific descriptors followed, *bNumInterface* set to the summary number of interfaces of all functions used.

```
// Composite Configuration Descriptor
{
  sizeof(USBConfigurationDescriptor),
  USBGenericDescriptor_CONFIGURATION,
  sizeof(CompositeDriverConfigurationDescriptors),
  COMPOSITEDDriverDescriptors_NUMINTERFACE,
  1, // This is configuration #1
  0, // No string descriptor for this configuration
  BOARD_USB_BMATTRIBUTES,
  USBConfigurationDescriptor_POWER(100)
},
```

When the Configuration descriptor is requested by the host (by using the GET_DESCRIPTOR command), the device must also send all the related descriptors, i.e. IAD, Interface, Endpoint and Class-specific descriptors. It is convenient to create a single structure to hold all this data, for sending everything in one trunk. In the example software, a CompositeDriverConfiguration-Descriptors structure has been declared for that, and different function related descriptors will be automatically selected by a pre-defined value when making (usb_HIDMSD, usbHIDAUDIO, usb_CDCHID, usb_CDCAUDIO, usb_CDCMSD or usb_CDCCDC), see Section 4.9 on page 22 for more details.

4.3.3 General Interface Association Descriptor

Since Audio or CDC functions which has two interfaces may be included in the device, the IAD (Section 3.2.4.1 on page 6) is necessary. Its a standard new descriptor defined by USB.org. The detailed information for Audio IAD and CDC IAD can be found in the example driver source code.

4.3.4 Interface, Class-Specific & Endpoint Descriptors

All these descriptors are almost the same as their definitions in a single USB device, except the interface index related or endpoint addresses related settings. Please refer to the project source code and following application notes:

- **HID**: section 5.3, *AT91 USB HID Implementation*
- **MSD**: section 4.2, AT91 *USB Mass Storage Driver Implementation*
- **Audio**: AT91 *USB Audio Driver Implementation*
- **CDC**: section 4.4, AT91 *USB CDC Driver Implementation*

4.3.5 String Descriptor

Several descriptors can be commented with a String Descriptor. The latter is completely optional and does not have any effect on the detection of the device by the operating system. Whether or not to include them is entirely up to the programmer.

There is one exception to this rule when using the MSD class. According to the specification, there must be a **Serial Number** string. It must contains at least 12 characters, and these character must only be either letters (a-z, A-Z) or numbers (0-9). This causes no problem for the driver in practice, but this is a strict requirement for certification. Also please remember that string descriptors use the **Unicode** format.

4.4 Requests

4.4.1 General Request Handler

The composite itself does not define any special requests, but the device functions do.

Each function in the device has its own requests handler. The device invokes one of the function requests handler first, when the return value of the handler indicates the request is handled by the device function, then the device request handler return directly, but if it is not handled, the device request handler calls next function request handler to check it. It will be forwarded to the standard request handler if there is no valid handler for this request.

In the device function request handlers, *wIndex* of the request is checked first to confirm that the request is accepted by the function (interface). Then check if it is a standard request and handle it in two different ways.

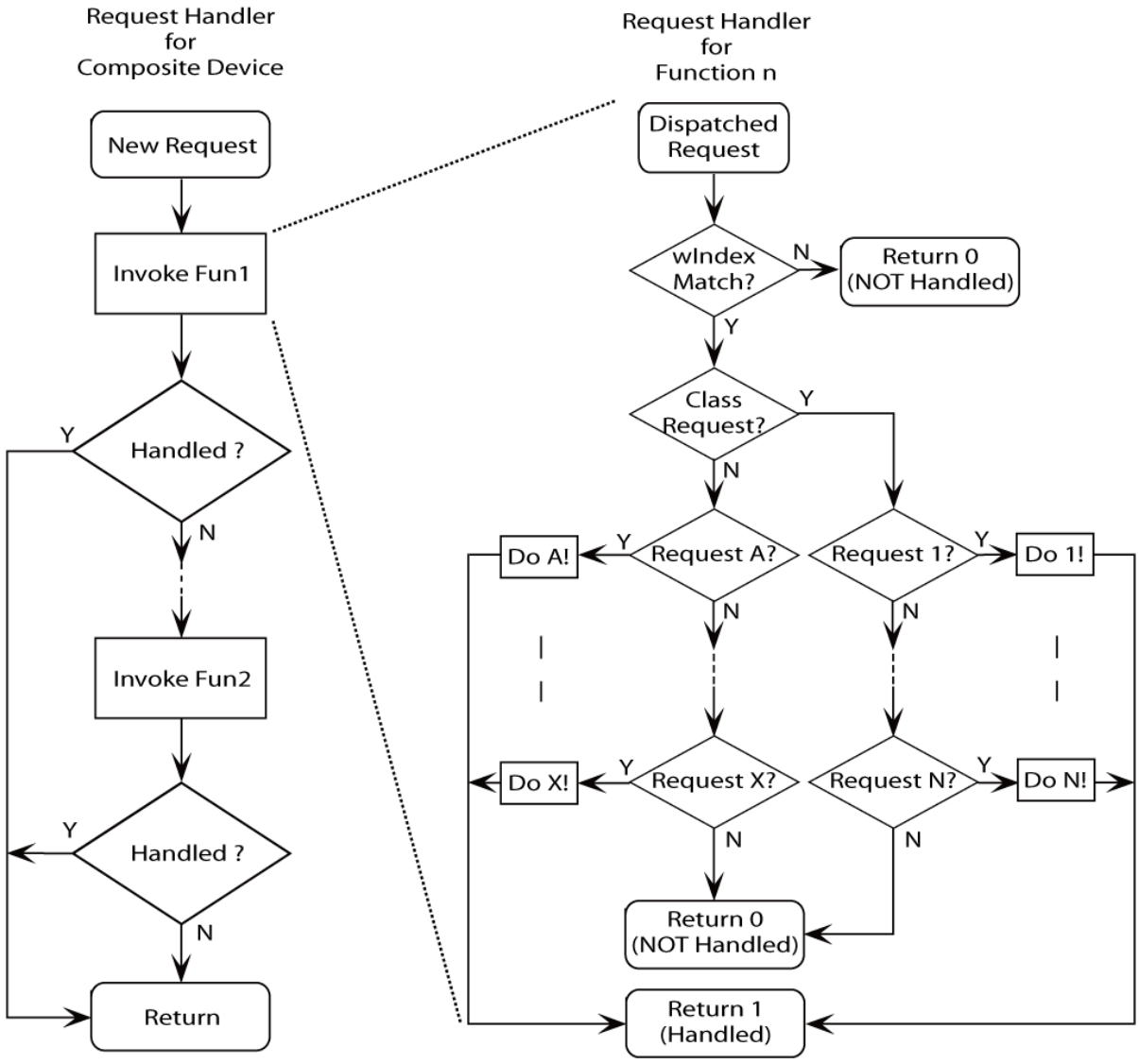
See Figure 4-2 for the general request handler work flow.

4.4.2 Class-Specific Requests

The handlers for functions to handle the standard and class-specific requests are the same as their way in a single device driver, except that the target interface index may be checked to avoid errors. Please refer to their related application notes for details:

- **HID**: section 4.4, *AT91 USB HID Driver Implementation*
- **MSD**: section 4.3, AT91 *USB Mass Storage Driver Implementation*
- **Audio**: AT91 *USB Audio Class Driver Implementation*
- **CDC**: section 4.5, AT91 *USB CDC Class Driver Implementation*

Figure 4-2. General Request Handler



4.5 Notifications

Notifications are supported only if CDC serial port device function is included, please refer to section 4.6 of AT91 *USB CDC Driver Implementation*.

4.6 Callbacks

The function in the device may need some callbacks to handle the USB events, such as configured or changed interface.

4.6.1 HID Keyboard Function Callbacks

For a HID Keyboard, the device function should start reporting to host through interrupt endpoint as soon as the device is configured, so USBDDriverCallbacks_ConfigurationChanged should be re-implement to monitor the configured status of the device, and then forward the event to the device function to handle it. For more details, see AT91 *USB HID Driver Implementation*.

4.6.2 Audio Speaker Function Callbacks

The USB Audio Speaker defines alternative interfaces to control the USB bandwidth, so there is a callback defined to handle the event of AT91 *USB Device Framework*. The function is USBDDriverCallbacks_InterfaceSettingChanged, which should be re-implemented to replace the original one. AUDDFunctionCallbacks_InterfaceSettingChanged is invoked and then let the Audio Device Function to handle the event. For more details, see AT91 *USB Audio Class Driver Implementation*.

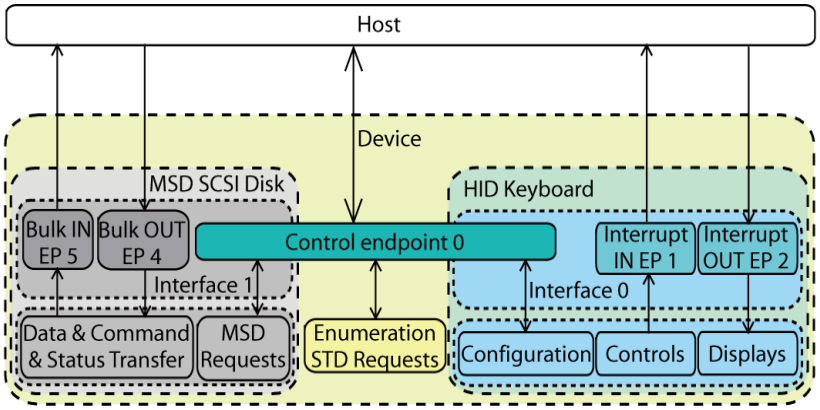
4.7 Composite Device Examples

Here lists the example entities for composite device with two functions. The actual device architecture and its descriptor structure are shown below, together with the implemented requests.

4.7.1 HID + MSD

4.7.1.1 Architecture

Figure 4-3. HID MSD Composite Driver Architecture



4.7.1.2 Device Descriptors

The actual descriptor structure used is defined as blow, for more detailed information, please refer to Section 4.3.4 on page 11 and all related application notes.

```
typedef struct {

  // Standard configuration descriptor.
  USBConfigurationDescriptor configuration;

  // --- HID
  USBInterfaceDescriptor hidInterface;
  HIDDescriptor hid;
  USBEndpointDescriptor hidInterruptIn;
  USBEndpointDescriptor hidInterruptOut;

  // --- MSD
  // Mass storage interface descriptor.
  USBInterfaceDescriptor msdInterface;
  // Bulk-out endpoint descriptor.
```

```
    USBEndpointDescriptor msdBulkOut;
    // Bulk-in endpoint descriptor.
    USBEndpointDescriptor msdBulkIn;

  } __attribute__ ((packed)) CompositeDriverConfigurationDescriptors;
```

4.7.1.3 *Device Requests*

These requests are dispatched by device request handler and handled in function request handler. See Section 4.4.1 on page 11 for general request handler procedure. See Figure 4-2 and Section 4-2 on page 12 and all the related application notes for the detailed implementation.

4.7.1.4 *MSD State Machine, LUN & SCSI Commands*

See section 4.4 to section 4.6, AT91 *Mass Storage Driver Implementation*. The general code procedure for MSD class is re-used.

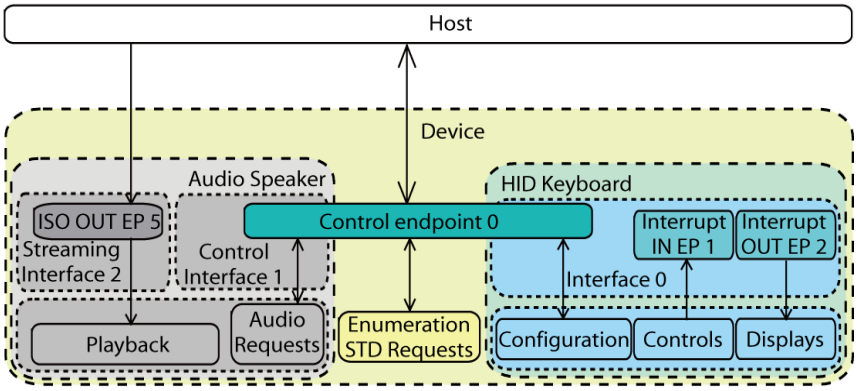
4.7.1.5 *MSD Driver class diagram*

See figure 4-5, AT91 *Mass Storage Driver Implementation*

4.7.2 HID + Audio

4.7.2.1 *Architecture*

Figure 4-4. HID Audio Composite Device Architecture



4.7.2.2 *Descriptors*

The actual descriptor structure used is defined as following, for more detailed information, please refer to Section 4.3.4 on page 11 and all related application notes.

```
typedef struct {

  // Standard configuration descriptor.
  USBConfigurationDescriptor configuration;

  // --- HID
  USBInterfaceDescriptor hidInterface;
  HIDDescriptor hid;
  USBEndpointDescriptor hidInterruptIn;
```

```
    USBEndpointDescriptor hidInterruptOut;

    // --- AUDIO
    // IAD 1
    USBInterfaceAssociationDescriptor audIAD;
    // Audio control interface.
    USBInterfaceDescriptor audInterface;
    // Descriptors for the audio control interface.
    AUDDSpeakerDriverAudioControlDescriptors audControl;
    // -- AUDIO out
    // Streaming out interface descriptor (with no endpoint, required).
    USBInterfaceDescriptor audStreamingOutNoIsochronous;
    // Streaming out interface descriptor.
    USBInterfaceDescriptor audStreamingOut;
    // Audio class descriptor for the streaming out interface.
    AUDStreamingInterfaceDescriptor audStreamingOutClass;
    // Stream format descriptor.
    AUDFormatTypeOneDescriptor1 audStreamingOutFormatType;
    // Streaming out endpoint descriptor.
    AUDEndpointDescriptor audStreamingOutEndpoint;
    // Audio class descriptor for the streaming out endpoint.
    AUDDataEndpointDescriptor audStreamingOutDataEndpoint;

} __attribute__ ((packed)) CompositeDriverConfigurationDescriptors;
```

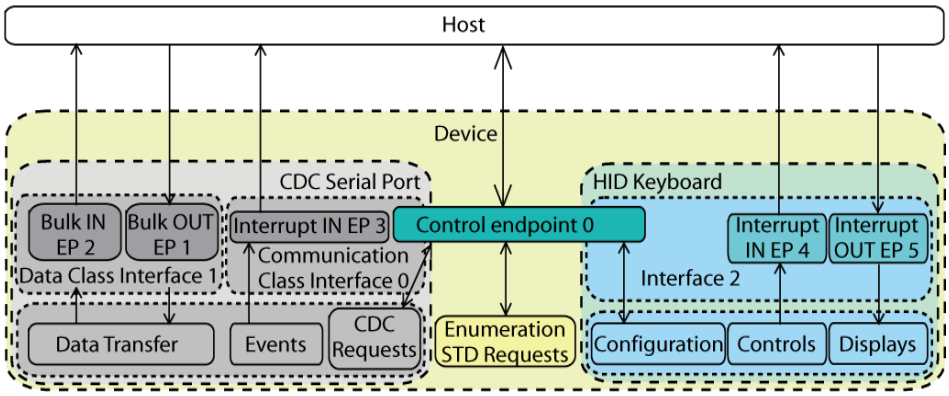
4.7.2.3 *Device Requests*

These requests is dispatched by device request handler and handled in function request handler. See Section 4.4.1 on page 11 for general request handler procedure. See Figure 4-2 and Section 4.4.2 on page 11 and all the related application notes for the detailed implementation.

4.7.3 CDC + HID

4.7.3.1 *Architecture*

Figure 4-5. CDC HID Composite Device Architecture



4.7.3.2 Descriptors

The actual descriptor structure used is defined as following, for more detailed information, please refer to Section 4.3.4 on page 11 and all related applicatoin notes.

```
typedef struct {

  // Standard configuration descriptor.
  USBConfigurationDescriptor configuration;

  // --- CDC 0
  // IAD 0
  USBInterfaceAssociationDescriptor cdcIAD0;
  // Communication interface descriptor
  USBInterfaceDescriptor cdcCommunication0;
  // CDC header functional descriptor.
  CDCHeaderDescriptor cdcHeader0;
  // CDC call management functional descriptor.
  CDCCallManagementDescriptor cdcCallManagement0;
  // CDC abstract control management functional descriptor.
  CDCAbstractControlManagementDescriptor cdcAbstractControlManagement0;
  // CDC union functional descriptor (with one slave interface).
  CDCUnionDescriptor cdcUnion0;
  // Notification endpoint descriptor.
  USBEndpointDescriptor cdcNotification0;
  // Data interface descriptor.
  USBInterfaceDescriptor cdcData0;
  // Data OUT endpoint descriptor.
  USBEndpointDescriptor cdcDataOut0;
  // Data IN endpoint descriptor.
  USBEndpointDescriptor cdcDataIn0;

  // --- HID
  USBInterfaceDescriptor hidInterface;
  HIDDescriptor hid;
  USBEndpointDescriptor hidInterruptIn;
  USBEndpointDescriptor hidInterruptOut;

} __attribute__ ((packed)) CompositeDriverConfigurationDescriptors;
```

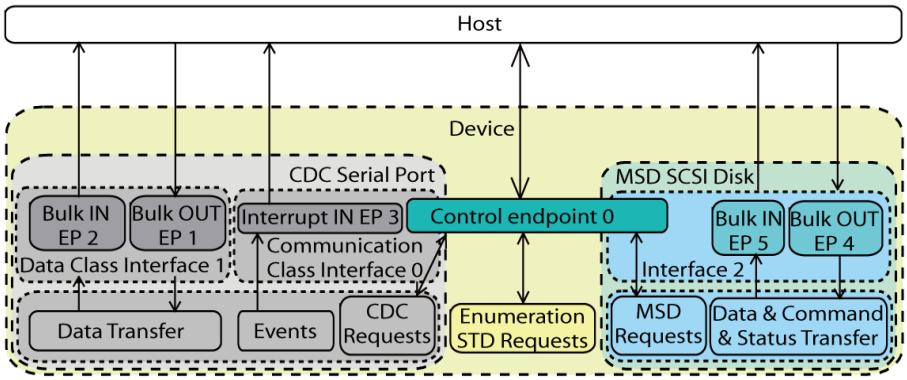
4.7.3.3 Device Requests

These requests is dispatched by device request handler and handled in function request handler. See Section 4.4.1 on page 11 for general request handler procedure. See Figure 4-2 and Section 4.4.2 on page 11 and all the related application notes for the detailed implementation.

4.7.4 CDC + MSD

4.7.4.1 Architecture

Figure 4-6. CDC MSD Composite Device Architecture



4.7.4.2 Descriptors

The actual descriptor structure used is defined as blow, for more detailed information, please refer to Section 4.3.4 on page 11 and all related application notes.

```
typedef struct {

  // Standard configuration descriptor.
  USBConfigurationDescriptor configuration;

  // --- CDC 0
  // IAD 0
  USBInterfaceAssociationDescriptor cdcIAD0;
  // Communication interface descriptor
  USBInterfaceDescriptor cdcCommunication0;
  // CDC header functional descriptor.
  CDCHeaderDescriptor cdcHeader0;
  // CDC call management functional descriptor.
  CDCCallManagementDescriptor cdcCallManagement0;
  // CDC abstract control management functional descriptor.
  CDCAbstractControlManagementDescriptor cdcAbstractControlManagement0;
  // CDC union functional descriptor (with one slave interface).
  CDCUnionDescriptor cdcUnion0;
  // Notification endpoint descriptor.
  USBEndpointDescriptor cdcNotification0;
  // Data interface descriptor.
  USBInterfaceDescriptor cdcData0;
  // Data OUT endpoint descriptor.
  USBEndpointDescriptor cdcDataOut0;
  // Data IN endpoint descriptor.
  USBEndpointDescriptor cdcDataIn0;
```

```
    // --- MSD
    // Mass storage interface descriptor.
    USBInterfaceDescriptor msdInterface;
    // Bulk-out endpoint descriptor.
    USBEndpointDescriptor msdBulkOut;
    // Bulk-in endpoint descriptor.
    USBEndpointDescriptor msdBulkIn;

} __attribute__ ((packed)) CompositeDriverConfigurationDescriptors;
```

4.7.4.3 *Device Requests*

These requests is dispatched by device request handler and handled in function request handler. See Section 4.4.1 on page 11 for general request handler procedure. See Figure 4-2 and Section 4.4.2 on page 11 and all their related application notes for the detailed implementation.

4.7.4.4 *MSD State Machine, LUN & SCSI Commands*

See section 4.4 to section 4.6, AT91 *Mass Storage Driver Implementation*. The general code procedure for MSD class is re-used.

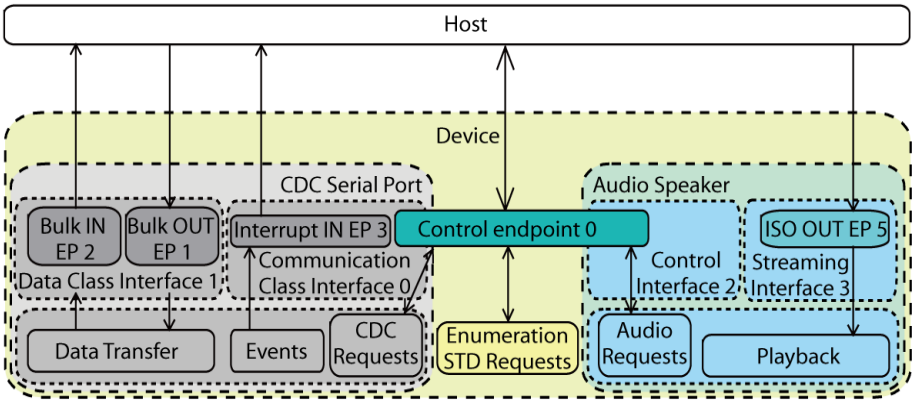
4.7.4.5 *Driver class diagram*

See figure 4-5, AT91 *Mass Storage Driver Implementation*.

4.7.5 CDC + Audio

4.7.5.1 *Architecture*

Figure 4-7. CDC Audio Composite Device Architecture



4.7.5.2 *Descriptors*

The actual descriptor structure used is defined as following, for more detailed information, please refer to Section 4.3.4 on page 11 and all related application notes.

```
typedef struct {

  // Standard configuration descriptor.
  USBConfigurationDescriptor configuration;

  // --- CDC 0
```

```
    // IAD 0
    USBInterfaceAssociationDescriptor cdcIAD0;
    // Communication interface descriptor
    USBInterfaceDescriptor cdcCommunication0;
    // CDC header functional descriptor.
    CDCHeaderDescriptor cdcHeader0;
    // CDC call management functional descriptor.
    CDCCallManagementDescriptor cdcCallManagement0;
    // CDC abstract control management functional descriptor.
    CDCAbstractControlManagementDescriptor cdcAbstractControlManagement0;
    // CDC union functional descriptor (with one slave interface).
    CDCUnionDescriptor cdcUnion0;
    // Notification endpoint descriptor.
    USBEndpointDescriptor cdcNotification0;
    // Data interface descriptor.
    USBInterfaceDescriptor cdcData0;
    // Data OUT endpoint descriptor.
    USBEndpointDescriptor cdcDataOut0;
    // Data IN endpoint descriptor.
    USBEndpointDescriptor cdcDataIn0;

    // --- AUDIO
    // IAD 1
    USBInterfaceAssociationDescriptor audIAD;
    // Audio control interface.
    USBInterfaceDescriptor audInterface;
    // Descriptors for the audio control interface.
    AUDDSpeakerDriverAudioControlDescriptors audControl;
    // -- AUDIO out
    // Streaming out interface descriptor (with no endpoint, required).
    USBInterfaceDescriptor audStreamingOutNoIsochronous;
    // Streaming out interface descriptor.
    USBInterfaceDescriptor audStreamingOut;
    // Audio class descriptor for the streaming out interface.
    AUDStreamingInterfaceDescriptor audStreamingOutClass;
    // Stream format descriptor.
    AUDFormatTypeOneDescriptor1 audStreamingOutFormatType;
    // Streaming out endpoint descriptor.
    AUDEndpointDescriptor audStreamingOutEndpoint;
    // Audio class descriptor for the streaming out endpoint.
    AUDDataEndpointDescriptor audStreamingOutDataEndpoint;

} __attribute__ ((packed)) CompositeDriverConfigurationDescriptors;
```

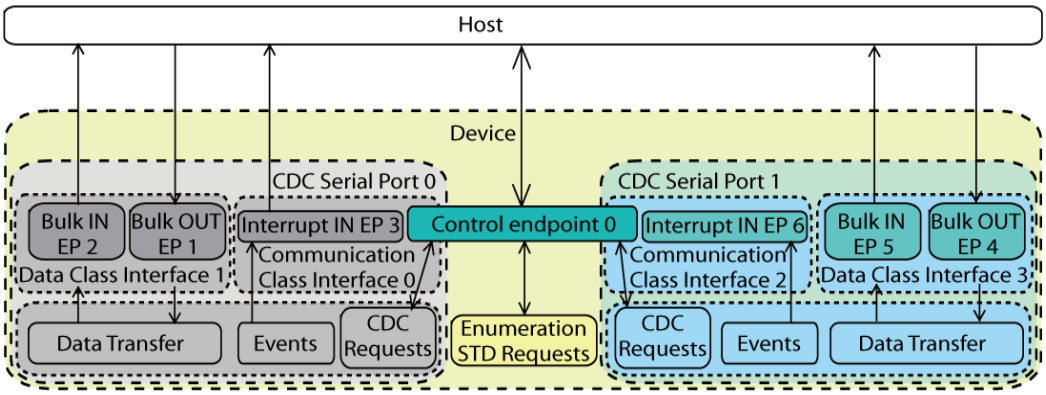
4.7.5.3 *Device Requests*

These requests are dispatched by device request handler and handled in function request handler. See Section 4.4.1 on page 11 for general request handler procedure. See Figure 4-2 and Section 4.4.2 on page 11 and all the related application notes for the detailed implementation.

4.7.6 CDC + CDC (Dual serial ports)

4.7.6.1 *Architecture*

Figure 4-8. Dual CDC Composite Device Architecture



4.7.6.2 *Descriptors*

The actual descriptor structure used is defined as following, for more detailed information, please refer to Section 4.3.4 on page 11 and CDC Implement application notes.

```
typedef struct {

  // Standard configuration descriptor.
  USBConfigurationDescriptor configuration;

  // --- CDC 0
  // IAD 0
  USBInterfaceAssociationDescriptor cdcIAD0;
  // Communication interface descriptor
  USBInterfaceDescriptor cdcCommunication0;
  // CDC header functional descriptor.
  CDCHeaderDescriptor cdcHeader0;
  // CDC call management functional descriptor.
  CDCCallManagementDescriptor cdcCallManagement0;
  // CDC abstract control management functional descriptor.
  CDCAbstractControlManagementDescriptor cdcAbstractControlManagement0;
  // CDC union functional descriptor (with one slave interface).
  CDCUnionDescriptor cdcUnion0;
  // Notification endpoint descriptor.
  USBEndpointDescriptor cdcNotification0;
  // Data interface descriptor.
  USBInterfaceDescriptor cdcData0;
```

```
    // Data OUT endpoint descriptor.
    USBEndpointDescriptor cdcDataOut0;
    // Data IN endpoint descriptor.
    USBEndpointDescriptor cdcDataIn0;

    // --- CDC 1
    // IAD 1
    USBInterfaceAssociationDescriptor cdcIAD1;
    // Communication interface descriptor
    USBInterfaceDescriptor cdcCommunication1;
    // CDC header functional descriptor.
    CDCHeaderDescriptor cdcHeader1;
    // CDC call management functional descriptor.
    CDCCallManagementDescriptor cdcCallManagement1;
    // CDC abstract control management functional descriptor.
    CDCAbstractControlManagementDescriptor cdcAbstractControlManagement1;
    // CDC union functional descriptor (with one slave interface).
    CDCUnionDescriptor cdcUnion1;
    // Notification endpoint descriptor.
    USBEndpointDescriptor cdcNotification1;
    // Data interface descriptor.
    USBInterfaceDescriptor cdcData1;
    // Data OUT endpoint descriptor.
    USBEndpointDescriptor cdcDataOut1;
    // Data IN endpoint descriptor.
    USBEndpointDescriptor cdcDataIn1;

} __attribute__ ((packed)) CompositeDriverConfigurationDescriptors;
```

4.7.6.3 *Device Requests*

These requests are dispatched by device request handler and handled in function request handler. See Section 4.4.1 on page 11 for general request handler procedure. See Figure 4-2 and all the related application notes for the detailed implementation.

Please note that they are two entities of the same function in the device as two virtual serial ports and they can be differed by its interface definitions and affected endpoints.

4.8 Main Application

Normally the main application code can be divided into four parts: initialization, interrupt handlers, callback handlers and the main loop. See the source code and the device related application notes for more details.

4.8.1 Initialization

This part initializes all peripherals and drivers, depending on the functions which are included. Normally there are following general initializations: debug initialization, USB Vbus detect (PIO) initialization, and composite device driver initialization; Those initializations are based on the working functions: clock PIO and clock settings for USB Audio device, HID keyboard PIO initial-

ization, AC97 or SPI & SSC for AT73C213 initialization, MSD Media and LUN initialization; Some extra initializations may be needed for timers.

4.8.2 Interrupt Handlers

This part includes the necessary handler for Vbus detect, the media handler for MSD device function, USART handler for CDC serial port, and the SSC handler for Audio device function if needed. One extra handler may be needed for the timer or PIT.

4.8.3 Callback Handlers

Some of the device function requires callback handler to execute the request from host, such as the mute state changed by the host, or data received or sent, so that another transfer can be started.

4.8.4 Main Loop

All driver functions are non-blocked functions, and many of them are called in the main loop, to receive data from the host, to scan the input keys and send report to host (for HID), to run the state machine (for MSD).

4.9 Example Software Usage

4.9.1 File Architecture

In the example program provided with this application note, the actual driver is divided into four parts:

- **AT91 USB Device Framework folders and files:** See section 4.1 *AT91 USB Device Framework*
- **The Driver Function Re-Used files**: files for the device driver that provided with other related application note. See section 4.8.1 AT91 *USB CDC Serial Driver Implementation*, section 5.6.1 AT91 *USB HID Driver Implementation*, section 4.8.1 AT91 *USB Mass Storage Driver Implementation*, AT91 *USB Audio Driver Implementation*. Except the files that suffixed by “Driver” and “DriverDescriptors”, which is replaced by the following part.
- **The Driver file and Function files**: source files for the composite driver, and all its functions
  - **at91lib/usb/device/composite/**: The folder holds the composite driver source files as following
  - *COMPOSITEDDriver.c*: source for the composite driver
  - *COMPOSITEDDriver.h*: header file with definitions for the composite driver
  - *COMPOSITEDDriverDescriptors.c*: source for the composite driver descriptors
  - *COMPOSITEDDriverDescriptors.h*: header file for composite driver descriptors
  - *HIDDFunctionDriver.c*: source for the HID Keyboard device function driver
  - *HIDDFunctionDriver.h*: header file with definitions for the HID Keyboard function
  - *HIDDFunctionDriverDescriptors.h*: HID related definitions in the composite driver descriptors
  - *MSDDFunctionDriver.c*: source for the MSD device function driver
  - *MSDDFunctionDriver.h*: definitions for the MSD driver function
  - *MSDDFunctionDriverDescriptors.h*: MSD related definitions in the composite driver descriptors
  - *AUDDFunctionDriver.c*: source for the Audio device function driver

- *AUDDFunctionDriver.h*: definitions for the Audio driver fucntion
- *AUDDFunctionDriverDescriptors.h*: Audio related definitions in the composite driver descriptors
- *CDCDFunctionDriver.c*: source for the CDC serial function driver
- *CDCDFunctionDriver.h*: definitions for the CDC serial driver function
- *CDCDFunctionDriverDescriptors.h*: CDC related definitions in the composite driver descriptors

- **The main application**: The main application, which uses the driver to bridge the board interface and USB interface, it also performs the state machine
  - **usb-device-composite-project/**: folder for the main application files
  - *main.c*: The source file to initialize and run all integrated functions together.

4.9.2 Compilation

The software is provided with a **Makefile** to build it. It requires the *make* utility, which is available in GNU make. Please refer to the *AT91 USB Framework* application note for more information on general options and parameters of the Makefile.

To build the composite example, just go to the usb-device-composite-project/ directory and run make:

```
make
```

Then, the resulting binary will be located in the usb-device-composite-project/bin/ directory.

There is optional usb_CLASS parameter to assign the functions included:

```
make usb_CLASS=usb_CDCCDC
```

Then the resulting binary runs a device with two USB serial ports. The other available settings are: usb_HIDMSD, usb_HIDAUDIO, usb_CDCAUDIO, usb_CDCHID, usb_CDCMSD). The default setting is usb_CDCMSD.

4.10 Using a Generic Windows Driver

The device functions are generally supported by Microsoft® Windows®, but for composite device, some patches are needed. All the composite devices above are tested under windows XP (SP3) and works fine. For CDC serial port, additional windows driver file (.inf) is required.

4.10.1 Windows Patches for composite

The old version of USB windows driver has bugs to support multi-interface composite devices and must be updated. There are two ways to obtain these updates:

4.10.1.1 Windows Service pack 3

All the fixes for USB generic driver are included in Windows XP Service Pack 3. It can be found at Microsoft web site.

4.10.1.2 Windows Hot fixes

Two hot fixes are necessary for window to recognize the composite device correctly:

- Hot fix: Composite USB devices whose interfaces are not sequentially numbered do not work in Windows XP.

- Hot fix: The driver Usbser.sys may not be load when a USB device uses an IAD to define a function that has multiple interfaces, and this function uses the Usbser.sys driver file in Windows XP.

4.10.2 Windows Driver Files for CDC serial function

In order to make windows recognize the CDC serial device correctly, it is necessary to write a .inf file. Please refer to section 4.9.1.1 AT91 *USB CDC Driver Implementation* for detailed information. Only one thing should be modified to match the composite device function installation. For composite devices, the hardware ID is made up of the Vender ID, the Product ID and (optionally) the Device Release Number and the start interface number of the function. Those values are extracted from the device descriptors provided during the enumeration phase, the following is the example of the modification for the dual-port CDC serial:

```
[AtmelMfg]
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_6119&MI_00 ; 1st
COM device
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_6119&MI_02 ; 2nd
COM device
```

4.10.3 Using the Driver

When a new device is plugged in for the first time, Windows looks for an appropriate specific or generic driver to use it. The composite device will be recognized as “USB Composite Device”. Then Windows search the driver for the functions inside the composite device.

Please refer to the driver function related application notes for more details. The final installation file is as following:

```
; $Id: 6119.inf,v 1.1.2.1 2006/12/05 08:33:25 danielru Exp $

[Version]
Signature="$Chicago$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%ATMEL%
DriverVer=09/12/2006,1.1.1.5

[DestinationDirs]
DefaultDestDir=12

[Manufacturer]
%ATMEL%=AtmelMfg

[AtmelMfg]
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_6127&MI_00
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_6128&MI_00
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_6129&MI_00
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_6119&MI_00
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_6119&MI_02

[USBtoSer.Install]
```

```
include=mdmcpq.inf
CopyFiles=FakeModemCopyFileSection
AddReg=USBtoSer.AddReg
[USBtoSer.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,usbser.sys
HKR,,EnumPropPages32,,"MsPorts.dll,SerialPortPropPageProvider"

[USBtoSer.Install.Services]
AddService=usbser,0x00000002,USBtoSer.AddService

[USBtoSer.AddService]
DisplayName=%USBSer%
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%12%\usbser.sys

[Strings]
ATMEL="ATMEL Corp."
USBtoSerialConverter="AT91 USB to Serial Converter"
USBSer="USB Serial Driver"
```

5. Revision History

Table 5-1.

Document Ref.	Date	Comments	Change Request Ref.
6436A	01-Jul-09	First issue.	

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com
www.atmel.com/AT91SAM

Technical Support
AT91SAM Support
Atmel techincal support

Sales Contacts
www.atmel.com/contacts/

Literature Requests
www.atmel.com/literature