NUC930 USB Host Core Library

May 16, 2009
Preliminary Released

Contents

1. USB Core Library Overview

The USB Core library is composed of four major parts, which are OHCI driver, EHCI driver, USB driver, and USB hub device driver. Each of these four drivers also represents one of the three-layered USB driver layers. Figure 1-1 presents the driver layers of the USB library.

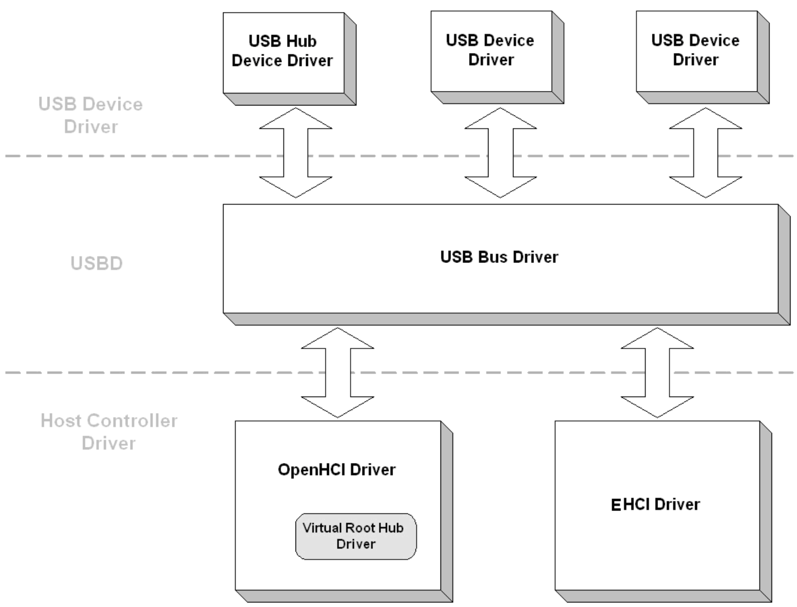

Figure 1-1: USB driver layers of the USB library

2. Data Structures

The USB Core library has included many complicated data structures to describe a USB bus, a device, a driver, various descriptors, and so on. To realize these data structures may be necessary for a USB device driver designer. In the following sections, we will introduce all data structures you may need. These data structures are all defined in header file <usb.h>.

2.1 USB_DEV_T

USB_DEV_T is the data structure used to represent a device instance. Once the host finds that a device presented on a USB bus, the USB system software is notified. The USB system software resets and enables the hub port to reset the device. It then creates a ***USB_DEV_T*** for the newly detected device. For each USB device presented on the bus, even the same device type, USB system software will create a ***USB_DEV_T*** to represent it as an instance.

The contents of all members of ***USB_DEV_T*** are automatically assigned by USB system software. The USB system software will assign a unique device number, read device descriptor and configuration descriptors, and create parent/child relationships. The definition of USB_DEV_T is listed below, and the detailed descriptions can be found in Table 2-1.

```
typedef struct usb_device
{
    INT       devnum;
    INT       slow;
    enum
    {
              USB_SPEED_UNKNOWN = ,
              USB_SPEED_LOW,
              USB_SPEED_FULL,
              USB_SPEED_HIGH
    }         speed;
    struct usb_tt  *tt;
    INT       ttport;
    INT       refcnt;
    UINT32    toggle[2];
```

```
    UINT32  halted[2];
    INT     epmaxpacketin[16];
    INT     epmaxpacketout[16];
    struct usb_device  *parent;
    INT     hub_port;
    USB_BUS_T  *bus;
    USB_DEV_DESC_T  descriptor;
    USB_CONFIG_DESC_T *config;
    USB_CONFIG_DESC_T *actconfig;
    CHAR    **rawdescriptors;
    INT     have_langid;
    INT     string_langid;
    VOID    *hcpriv;
    INT     maxchild;
    struct usb_device  *children[USB_MAXCHILDREN];
} USB_DEV_T;
```

Table 2-1: Members of *USB_DEV_T*

Member	***Description***
devnum	Device number on USB bus; each device instance has a unique device number
slow	Is low speed device speed ? (1: yes; 0: no)
speed	Device speed
refcnt	Reference count (to count the number of users using the device)
toggle[2]	Data toggle; one bit for each endpoint ([0] = IN, [1] = OUT)
halted[2]	Endpoint halts; one bit for each endpoint ([0] = IN, [1] = OUT)
epmaxpacketin[16]	IN endpoints specific maximum packet size (each entry represents for an IN endpoint of this device)
epmaxpacketout[16]	OUT endpoints specific maximum packet size (each entry represents for an OUT endpoint of this device)
parent	Parent device in the bus topology (generally, it should be a hub)
bus	The bus on which this device was presented
descriptor	Device descriptor
config	All of the configuration descriptors

actconfig	The descriptor of the active configuration
rawdescriptors	Raw descriptors for each configuration descriptor (driver can find class specific or vendor specific descriptors from the *rawdescriptors*)
have_langid	Whether string_langid is valid yet
string_langid	Language ID for strings
hcpriv	Host controller private data
maxchild	Number of ports if this is a hub device
children[ ]	Link to the downstream port device if this is a hub device

2.2 Descriptor Structures

In the ***USB_DEV_T*** structure, device descriptor, configuration descriptors, and raw descriptor are included. The USB Driver will acquire these descriptors from device automatically while the device is probed. The USB Driver issues GET_DESCRIPTOR standard device request to acquire the configuration descriptors. It also parses the returned descriptors to create configuration-interface-endpoint descriptor links. Client software can obtain any configuration, interface, or endpoint descriptors by tracing the descriptor link started from ***USB_DEV_T***. As USB Driver cannot understand class-specific and vendor-specific descriptors, it does not create link for these descriptors. If the client software wants to obtain any class-specific or vendor-specific descriptors, it can parse the descriptors stored in raw descriptor, which is the original descriptors list returned from the device. Table2-2, Table 2-3, Table 2-4, and Table 2-5 describe the structures defined for device descriptors, configuration descriptors, interface descriptors, and endpoint descriptors, respectively.

Figure 2-1 presents an overview on the relationship of these data structures. From ***USB_DEV_T*** (device instance structure), ***USB_DEV_DEC_T*** (device descriptor structure) and ***USB_CONFIG_DEC_T*** (configuration descriptor structure), ***USB_IF_DESC_T*** (interface descriptor structure), to ***USB_EP_DESC_T*** (endpoint descriptor structure), all structure entries are linked in top-down order.

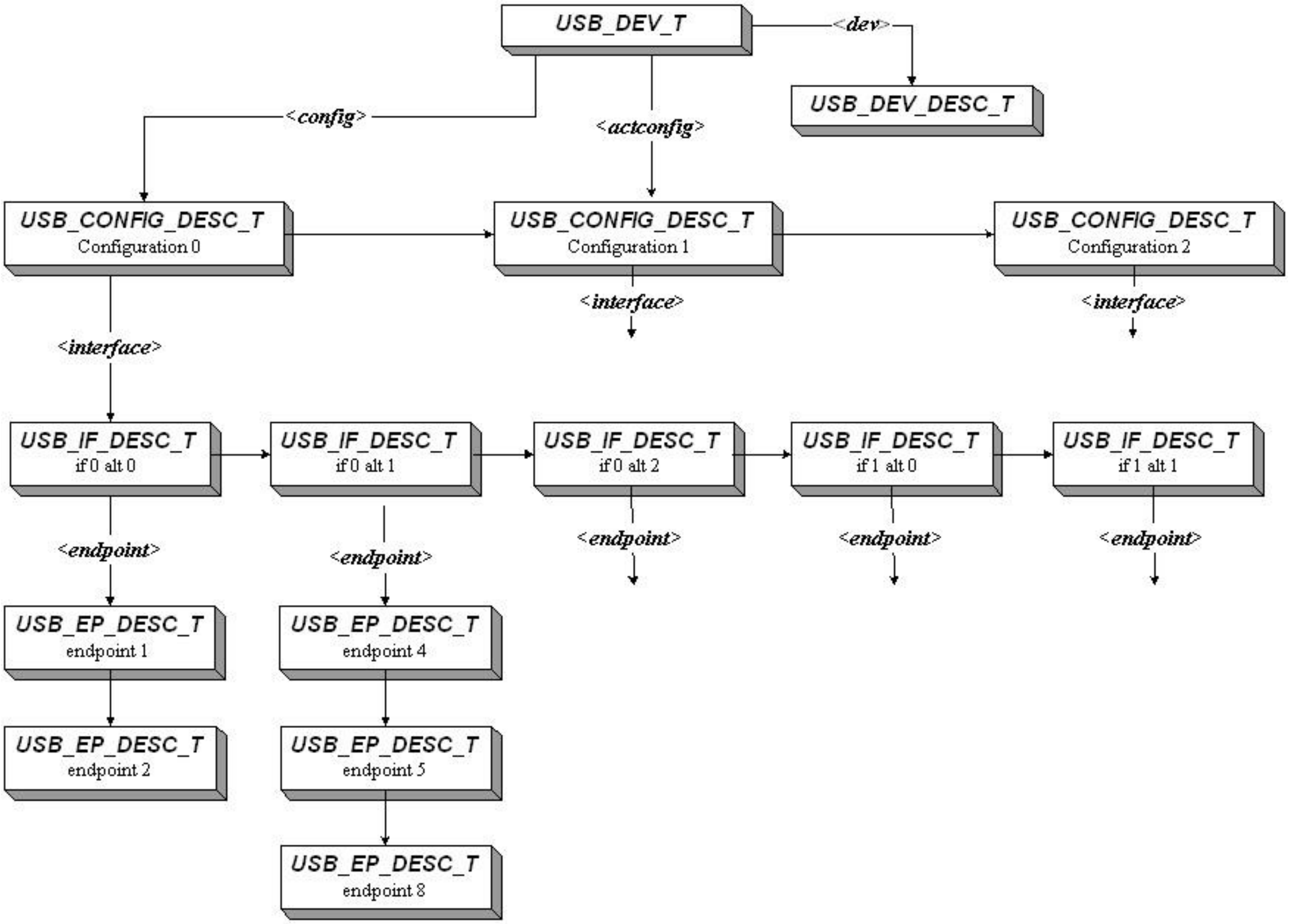

Figure 2-1 Descriptors relationship

```
/* Device descriptor */
typedef struct usb_device_descriptor
{
    __packed UINT8  bLength;
    __packed UINT8  bDescriptorType;
    __packed UINT16 bcdUSB;
    __packed UINT8  bDeviceClass;
    __packed UINT8  bDeviceSubClass;
```

```
    __packed UINT8  bDeviceProtocol;
    __packed UINT8  bMaxPacketSize0;
    __packed UINT16 idVendor;
    __packed UINT16 idProduct;
    __packed UINT16 bcdDevice;
    __packed UINT8  iManufacturer;
    __packed UINT8  iProduct;
    __packed UINT8  iSerialNumber;
    __packed UINT8  bNumConfigurations;
} USB_DEV_DESC_T;
```

Table 2-2: Members of *USB_DEV_DESC_T*

Member	*Description*
bLength	Size of the descriptor in bytes
bDescriptorType	DEVICE descriptor type (0x01)
bcdUSB	USB specification release number in BCD format
bDeviceClass	Device class code
bDeviceSubclass	Device subclass code
bDeviceProtocol	Protocol code
bMaxPacketSize0	Maximum packet size for endpoint zero
idVendor	Vendor ID
idProduct	Product ID
iManufacturer	Device release number in BCD format
iProduct	Index of string descriptor describing product
iSerialNumber	Index of string descriptor describing the serial number
bNumConfigurations	Number of possible configurations

You may have found that the definition of ***USB_DEV_DESC_T*** is fully compliant to the definition of device descriptor defined in USB 1.1 specification. In fact, the USB Driver acquires the device descriptor and fills it into this structure without making any modifications.

```
/* Configuration descriptor information.. */
typedef struct usb_config_descriptor
{
    __packed UINT8   bLength;
    __packed UINT8   bDescriptorType;
    __packed UINT16  wTotalLength;
    __packed UINT8   bNumInterfaces;
    __packed UINT8   bConfigurationValue;
    __packed UINT8   iConfiguration;
    __packed UINT8   bmAttributes;
    __packed UINT8   MaxPower;
    USB_IF_T  *interface;
    UINT8   *extra;
    INT     extralen;
} USB_CONFIG_DESC_T;
```

Table 2-3: Members of *USB_CONFIG_DESC_T*

Member	***Description***
bLength	Size of the descriptor in bytes
bDescriptorType	CONFIGURATION descriptor type (0x02)
wTotalLength	The total length of data returned for this descriptor
bNumInterfaces	Number of interface supported by this configuration
bConfigurationValue	Value to use as an argument to the SetConfiguration() request to select the active configuration
iConfiguration	Index of string descriptor describing this configuration
bmAttributes	Bitmap describing the configuration characteristics
MaxPower	Maximum power consumption of the USB device from the bus in this specific configuration when the device is fully operational (in mA)
interface	Refer to the interface descriptor list (recorded in USB_IF_DESC_T structure format) returned by this configuration
extra	Refer to the memory buffer preserve the raw data of this configuration descriptor itself
extralen	The length of the <*extra*> memory buffer

The ***dev->config*** refers to a list of configurations supported by this device. Client software can access any configuration by indexing the configuration, for example, dev->config[0] is referred to the first configuration of this device. While <***config***> of ***USB_DEV_T*** refers to the configuration list, <***actconfig***> refers to the currently activated configuration. There can be only one configuration activated at the same time.

The structure members from <***bLength***> to <***MaxPower***> are fully compliant to that defined in USB 1.1 specification. The <***interface***> refers to a list of interfaces supported by this configuration. In addition, USB Driver keeps the interface descriptor itself in a dynamically allocated memory buffer, which is referred to by <***extra***>, and the length of this memory buffer is <***extralen***>.

An interface may contain several alternate settings. Each alternate setting has its own set of endpoints. USB Driver creates a single ***USB_IF_DESC_T*** structure for each alternate interface setting and links them in the order that they presented in the returned data of a configuration descriptor.

```
/* Interface descriptor */
typedef struct usb_interface_descriptor
{
    __packed UINT8  bLength;
    __packed UINT8  bDescriptorType;
    __packed UINT8  bInterfaceNumber;
    __packed UINT8  bAlternateSetting;
    __packed UINT8  bNumEndpoints;
    __packed UINT8  bInterfaceClass;
    __packed UINT8  bInterfaceSubClass;
    __packed UINT8  bInterfaceProtocol;
    __packed UINT8  iInterface;
    USB_EP_DESC_T *endpoint;
    UINT8   *extra;
    INT     extralen;
} USB_IF_DESC_T;
```

Table 2-4: Members of *USB_IF_DESC_T*

Member	***Description***
bLength	Size of the descriptor in bytes

bDescriptorType	INTERFACE descriptor type (0x04)
bInterfaceNumber	Number of interface. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration.
bAlternateSetting	Value used to select alternate setting for this interface
bNumEndpoints	Number of endpoints used by this interface (excluding endpoint zero)
bInterfaceClass	Class code
bInterfaceSubClass	Subclass code
bInterfaceProtocol	Protocol code
iInterface	Index of string descriptor describing this interface
endpoint	Refer to the endpoint descriptor list (recorded in USB_EP_DESC_T structure format) of this interface returned by this configuration
extra	Refer to the memory buffer preserve the raw data of this interface descriptor itself
extralen	The length of the <*extra*> memory buffer

The ***dev->config[n]->interface*** refers to a list of interfaces supported by configuration n. The structure members from <***bLength***> to <***iInterface***> are fully compliant to that defined in USB 1.1 specification. The <***endpoint***> refers to a list of endpoints supported by this interface. In addition, USB Driver keeps the interface descriptor itself in a dynamically allocated memory buffer, which is referred to by <***extra***>, and the length of this memory buffer is <***extralen***>.

```
/* Endpoint descriptor */
typedef struct usb_endpoint_descriptor
{
    __packed UINT8  bLength;
    __packed UINT8  bDescriptorType;
    __packed UINT8  bEndpointAddress;
    __packed UINT8  bmAttributes;
    __packed UINT16 wMaxPacketSize;
    __packed UINT8  bInterval;
    __packed UINT8  bRefresh;
    __packed UINT8  bSynchAddress;
    UINT8   *extra;
    INT     extralen;
```

```
} USB_EP_DESC_T;
```

Table 2-5: Members of *USB_EP_DESC_T*

Member	*Description*
bLength	Size of the descriptor in bytes
bDescriptorType	ENDPOINT descriptor type (0x05)
bEndpointAddress	The address of this endpoint
bmAttributes	Transfer type of this endpoint
wMaxPacketSize	The maximum packet size this endpoint is capable of sending or receiving
bInterval	Interval for polling endpoint for data transfers (in milliseconds)
bRefresh	Audio extensions to the endpoint descriptor
bSynchAddress	Audio extensions to the endpoint descriptor
extra	Refer to the memory buffer preserve the raw data of this endpoint descriptor itself
extralen	The length of the <*extra*> memory buffer

2.3 DEV_REQ_T

DEV_REQ_T is used to represent the eight bytes device request in a control transfer. All device requests, including standard device requests, class-specific device requests, and vendor-specific device requests, are written in the ***DEV_REQ_T*** structure, which is also a member of a URB, and transferred to device through the control pipe.

```
typedef struct
{
    __packed UINT8  requesttype;
    __packed UINT8  request;
    __packed UINT16 value;
```

```
    __packed UINT16 index;
    __packed UINT16 length;
} DEV_REQ_T;
```

Table 2-6: Members of *DEV_REQ_T*

Member	***Description***
requesttype	Characteristics of request
request	Specific request
value	Word-sized field that varies according to request
index	Word-sized field that varies according to request
length	Number of bytes to transfer if there is a DATA stage

2.4 USB_DEV_ID_T

When the USB System Software detects a device being attached, it must find out the corresponding device driver for each of its interface from the registered driver list. It can try to invoke the ***probe()*** routine of each registered device driver for each device interface, but this is not efficient and time-consuming. If the USB System Software can make some simple judgment before trying invoking a device driver, it will be better. This is the purpose of ***USB_DEV_ID_T***. The USB Library employ device ID to identify the appropriate device drivers.

When a device driver is registered to USB Driver, it may provide a device ID table, which is structured in ***USB_DEV_ID_T*** format. In the device ID table, driver can specify the characteristics of the USB device interface that the driver would serve. If a driver does not provide a device ID table, then the USB Driver will always try to invoke it when a new device is detected.

The device driver can use device ID table to specify several checks of characteristics, including vendor ID, device ID, release number, device class, device subclass, device protocol, interface class, interface subclass, and interface protocol. The device driver can specify one or more checks. The more checks are specified, the more specific device interface can be identified. Table 2-7 lists the entries of device ID table.

```
typedef struct usb_device_id
{
    UINT16  match_flags;
    UINT16  idVendor;
    UINT16  idProduct;
    UINT16  bcdDevice_lo;
    UINT16  bcdDevice_hi;
    UINT8   bDeviceClass;
    UINT8   bDeviceSubClass;
    UINT8   bDeviceProtocol;
    UINT8   bInterfaceClass;
    UINT8   bInterfaceSubClass;
    UINT8   bInterfaceProtocol;
    UINT32  driver_info;
} USB_DEV_ID_T;
```

Table 2-7: Members of *USB_DEV_ID_T*

Member	***Description***
matchflag	A bitmask of flags, used to determine which of the following items are to be used for matching
idVendor	Used to compare the vendor ID recorded in device descriptor
idProduct	Used to compare the product ID recorded in device descriptor
bcdDevice_lo	Specify the low limit of device release number
bcdDevice_hi	Specify the high limit of device release number
bDeviceClass	Used to compare the class code in device descriptor
bDeviceSubClass	Used to compare the subclass code in device descriptor
bDeviceProtocol	Used to compare the protocol code in device descriptor
bInterfaceClass	Used to compare the class code in interface descriptor
bInterfaceSubClass	Used to compare the subclass code in interface descriptor
bInterfaceProtocol	Used to compare the protocol code in interface descriptor

There are 10 check items can be used to identify a specific type of device. To select which of these check items should be used to identify a device type is controlled by the <***matchflag***> member, which is a 16bits bit-mask flag. Each bit of < ***matchflag*** > is corresponding to one of these check items. The bit-map definition of < ***matchflag*** > is defined as the followings:

```
#define USB_DEVICE_ID_MATCH_VENDOR              0x0001
#define USB_DEVICE_ID_MATCH_PRODUCT             0x0002
#define USB_DEVICE_ID_MATCH_DEV_LO              0x0004
#define USB_DEVICE_ID_MATCH_DEV_HI              0x0008
#define USB_DEVICE_ID_MATCH_DEV_CLASS           0x0010
#define USB_DEVICE_ID_MATCH_DEV_SUBCLASS        0x0020
#define USB_DEVICE_ID_MATCH_DEV_PROTOCOL        0x0040
#define USB_DEVICE_ID_MATCH_INT_CLASS           0x0080
#define USB_DEVICE_ID_MATCH_INT_SUBCLASS        0x0100
#define USB_DEVICE_ID_MATCH_INT_PROTOCOL        0x0200
```

For convenience of driver implementation, the USB library also provides some useful macros that facilitate the development of device driver. These macros are all listed in the followings, you can also define your own macros:

```
/* Some useful macros */
#define USB_DEVICE(vend,prod) \
        { USB_DEVICE_ID_MATCH_DEVICE, vend, prod, 0, 0,
          0, 0, 0, 0, 0, 0, 0 }

#define USB_DEVICE_VER(vend,prod,lo,hi) \
        { USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION, vend,
          prod, lo, hi, 0, 0, 0, 0, 0, 0, 0 }

#define USB_DEVICE_INFO(cl,sc,pr) \
        { USB_DEVICE_ID_MATCH_DEV_INFO, 0, 0, 0, 0, cl,
          sc, pr, 0, 0, 0, 0 }

#define USB_INTERFACE_INFO(cl,sc,pr) \
        { USB_DEVICE_ID_MATCH_INT_INFO, 0, 0, 0, 0, 0,
          0, 0, cl, sc, pr, 0 }
```

2.5 USB_DRIVER_T

The USB library has defined a generalized structure for all USB device drivers. To implement a USB device driver based on this library, you must create such a structure and register it to the USB Driver. Once you have registered your device driver, the USB Driver can determine whether to launch your driver when a new device is attached.

As we will give detail introduction to the implementation of USB device driver, we only briefly describe the members of ***USB_DRIVER_T*** as following:

```
typedef struct usb_device_id
{
    UINT16  match_flags;
    UINT16  idVendor;
    UINT16  idProduct;
    UINT16  bcdDevice_lo;
    UINT16  bcdDevice_hi;
    UINT8   bDeviceClass;
    UINT8   bDeviceSubClass;
    UINT8   bDeviceProtocol;
    UINT8   bInterfaceClass;
    UINT8   bInterfaceSubClass;
    UINT8   bInterfaceProtocol;
    UINT32  driver_info;
} USB_DEV_ID_T;
```

Table 2-8: Members of *USB_DRIVER_T*

Member	*Description*
matchflag	A bitmask of flags, used to determine which of the following items are to be used for matching
idVendor	Used to compare the vendor ID recorded in device descriptor
idProduct	Used to compare the product ID recorded in device descriptor
bcdDevice_lo	Specify the low limit of device release number

bcdDevice_hi	Specify the high limit of device release number
bDeviceClass	Used to compare the class code in device descriptor
bDeviceSubClass	Used to compare the subclass code in device descriptor
bDeviceProtocol	Used to compare the protocol code in device descriptor
bInterfaceClass	Used to compare the class code in interface descriptor
bInterfaceSubClass	Used to compare the subclass code in interface descriptor
bInterfaceProtocol	Used to compare the protocol code in interface descriptor

2.6 URB_T

USB specification has defined four transfer type: control, bulk, interrupt, and isochronous. In the USB library, all these four transfer types are accomplished by URB (USB Request Block). Please refer to Chapter 3 for details about the implementation of each transfer type by using URB.

3. Data Transfer

USB specification defines four transfer types, control, bulk, interrupt, and isochronous. The USB device driver performs data transfers by preparing an URB and transfer it to the underlying USB system software. The URBs are designed to be accommodated with all four transfer types. By configuring the URB, USB device driver can specify the destination device interface and endpoint, the data buffer and data length to be transferred, the callback routine on completion, and other detail information. USB device driver passed the URB to the underlying USB system software, which will interpret the URB and accomplish the data transfers by initiating USB transactions between W90X900 Host Controller and the target device endpoint.

URB has been designed to be accommodated with all four USB data transfer types. Due to the characteristics of different transfer types, various requirements must be satisfied to fulfill the transfer. For example, URB contains <***setup_packet***> for control transfer, <***interval***> for interval transfer, <***start_frame***> and <***number_of_packets***> for isochronous transfer, and <***transfer_buffer***> for all transfers. To implement a USB device driver, the programmers use URBs to accomplish all data transfers to all of the various endpoints.

For a specific endpoint, after delivering a URB to the underlying USB system software, the USB device driver must not deliver another URB to the same endpoint until the current transfer was done by the USB system software. That is, the driver must be blocked in waiting completion of the URB. URB includes a <***complete***> function pointer to solve the block waiting issue. The USB device driver provided a callback function and have <complete> pointer being referred to the callback function. On completion of this URB, the USB system software will invoke the callback function. Thus, the USB device driver was notified with the completion event, and can stop waiting. Note that the callback functions are invoked from an HISR, the execution time must be as short as possible.

3.1 Pipe Control

Before delivering an URB, the USB device driver must determine which device and which endpoint the URB will operate on. This destination device and endpoint is determined by <***pipe***> of URB. <***pipe***> is actually a 32-bits unsigned integer. The USB library defined pipe structure with a 32-bits unsigned integer.

The USB library has defined several useful macros for pipe control. The pipe is defined as the followings:

31	30	29	28	27	26	25	24
Pipe Type		Reserved			Speed	Reserved	
23	22	21	20	19	18	17	16
Reserved				Data0/1	Endpoint		
15	14	13	12	11	10	9	8
	Device						
7	6	5	4	3	2	1	0
Direction	Reserved					Max Size	

Max Size [1 .. 0]

The maximum packet size. This field has been obsoleted. Now the maximum packet size is recorded in <***epmaxpacketin***> and <***epmaxpacketout***> fields of ***USB_DEV_T***.

Direction[7]

Direction of data transfer. 0 = Host-to-Device [out]; 1 = Device-to-Host [in]

Device[8 .. 14]

Device number. This is the unique device address, which is assigned by Host Controller driver by ***SET_ADDRESS*** standard request. With this unique device number, the USB device driver can correctly locate the target device.

Endpoint[15 .. 18]

Endpoint number. This is the endpoint number on the target device, that the pipe is created with. By definition, a pipe corresponds to a unique endpoint on a unique device. By determining the device number and endpoint number, USB device driver can uniquely identify a specific endpoint of a specific device.

Data0/1[19]

Data toggle Data0/Data1. This bit is used to record the current data toggle condition.

Speed[26]
Endpoint transfer speed. 1 = Low speed; 0 = Full speed.

Pipe Type[30 .. 31]
Transfer type. 00 = isochronous; 01 = interrupt; 10 = control; 11 = bulk.

The USB library has provided a lot of macros facilities for USB device driver designer. The device driver can use the facilities to rescuer the trouble of managing bit fields. These macros are listed in the followings:

Transfer Type

```
#define PIPE_ISOCHRONOUS                0
#define PIPE_INTERRUPT                  1
#define PIPE_CONTROL                    2
#define PIPE_BULK                       3

#define usb_pipetype(pipe)     (((pipe) >> 30) & 3)
#define usb_pipecontrol(pipe)  (usb_pipetype((pipe)) == PIPE_CONTROL)
#define usb_pipebulk(pipe)     (usb_pipetype((pipe)) == PIPE_BULK)
#define usb_pipeint(pipe)      (usb_pipetype((pipe)) == PIPE_INTERRUPT)
#define usb_pipeisoc(pipe)     (usb_pipetype((pipe)) == PIPE_ISOCHRONOUS)
```

Maximum Packet Size

```
#define usb_maxpacket(dev, pipe, out)    (out                   \
         ? (dev)->epmaxpacketout[usb_pipeendpoint(pipe)]    \
         : (dev)->epmaxpacketin [usb_pipeendpoint(pipe)] )
```

Direction

```
#define usb_packetid(pipe) (((pipe) & USB_DIR_IN) ?         \
                              USB_PID_IN : USB_PID_OUT)
#define usb_pipeout(pipe)  ((((pipe) >> 7) & 1) ^ 1)
#define usb_pipein(pipe)   (((pipe) >> 7) & 1)
```

Device Number

```
#define usb_pipedevice(pipe)    (((pipe) >> 8) & 0x7f)
#define usb_pipe_endpdev(pipe)  (((pipe) >> 8) & 0x7ff)
```

Endpoint Number

```
#define usb_pipe_endpdev(pipe)   (((pipe) >> 8) & 0x7ff)
#define usb_pipeendpoint(pipe)   (((pipe) >> 15) & 0xf)
```

Data Toggle

```
#define usb_pipedata(pipe)       (((pipe) >> 19) & 1)
#define usb_gettoggle(dev, ep, out)                        \
                    (((dev)->toggle[out] >> ep) & 1)
#define usb_dotoggle(dev, ep, out)                         \
                    ((dev)->toggle[out] ^= (1 << ep))
#define usb_settoggle(dev, ep, out, bit)                   \
                   ((dev)->toggle[out] =                   \
                   ((dev)->toggle[out] & ~(1 << ep)) |     \
                   ((bit) << ep))
```

Speed

```
#define usb_pipeslow(pipe)       (((pipe) >> 26) & 1)
```

Pipe Creation

```
static __inline UINT32 __create_pipe(USB_DEV_T *dev, UINT32 endpoint)
{
    return (dev->devnum << 8) | (endpoint << 15) | (dev->slow << 26);
}

static __inline UINT32 __default_pipe(USB_DEV_T *dev)
{
        return (dev->slow << 26);
}
```

```
/* Create various pipes... */
#define usb_sndctrlpipe(dev,endpoint)                \
           (0x80000000 | __create_pipe(dev,endpoint))
#define usb_rcvctrlpipe(dev,endpoint)                \
           (0x80000000 | __create_pipe(dev,endpoint) | USB_DIR_IN)
#define usb_sndisocpipe(dev,endpoint)                \
           (0x00000000 | __create_pipe(dev,endpoint))
#define usb_rcvisocpipe(dev,endpoint)                \
           (0x00000000 | __create_pipe(dev,endpoint) | USB_DIR_IN)
#define usb_sndbulkpipe(dev,endpoint)                \
           (0xC0000000 | __create_pipe(dev,endpoint))
#define usb_rcvbulkpipe(dev,endpoint)                \
           (0xC0000000 | __create_pipe(dev,endpoint) | USB_DIR_IN)
#define usb_sndintpipe(dev,endpoint)                 \
           (0x40000000 | __create_pipe(dev,endpoint))
#define usb_rcvintpipe(dev,endpoint)                 \
           (0x40000000 | __create_pipe(dev,endpoint) | USB_DIR_IN)
#define usb_snddefctrl(dev)                          \
           (0x80000000 | __default_pipe(dev))
#define usb_rcvdefctrl(dev)                          \
           (0x80000000 | __default_pipe(dev) | USB_DIR_IN)
```

3.2 Control Transfer

IN this section, we will introduce how to make control transfers by URBs. A control transfer is accomplished by sending a device request to the control endpoint of the target device. Depend on the request sent to device, there may be data stage or not.

The URB provided a <***setup_packet***> field to accommodate the device request command. The USB device driver must have the <***setup_packet***> of its URB being referred to an <unsigned char> array, which contained the device request command to be transferred. Note that <***setup_packet***> is designed to be used with control transfer.

If a device request included data stage, the data to be transferred must be referred to by the <***transfer_buffer***> pointer of URB. If the device request required data to be sent from Host to Device, the USB device driver must prepare a DMA buffer (non-cacheable) and fill the data to be transferred into this buffer. Then, the USB device driver have <***transfer_buffer***> pointer refer to this buffer, and specify the length of the buffer with <***transfer_buffer_length***> of the URB. If the device request requires data to be sent from Device to Host, the USB device driver must prepare a DMA buffer to receive the data from Device. Again, the USB device driver used <***transfer_buffer***> and <***Transfer_buffer_length***> to describe its DMA buffer. The <actual_length> is written by USB system software to tell the device driver how many bytes are actually transferred.

The USB device driver also has to prepare a callback function to be invoked by the USB system software. The callback function will be invoked on the completion of URB, in spite of success or fail. Generally, the callback function is responsible for waking up the task that delivered the URB. The callback function may also check the status of the URB to determine the transfer is successful or not. The following is an example of control transfer.

```
static VOID  ctrl_callback(URB_T *urb)
{
    PEGASUS_T   *pegasus = urb->context;

    switch ( urb->status )
    {
        case USB_ST_NOERROR:
            if (pegasus->flags & ETH_REGS_CHANGE)
            {
                pegasus->flags &= ~ETH_REGS_CHANGE;
                pegasus->flags |= ETH_REGS_CHANGED;
                update_eth_regs_async(pegasus);
                return;
            }
            break;
        case USB_ST_URB_PENDING:
            return;
        case USB_ST_URB_KILLED:
            break;
        default:
            printf("Warning - status %d\n", urb->status);
    }
```

```
    pegasus->flags &= ~ETH_REGS_CHANGED;
    if (pegasus->flags & CTRL_URB_SLEEP)
    {
       pegasus->flags &= ~CTRL_URB_SLEEP;
       NU_Set_Events(&pegasus->events, 1, NU_OR);  /* set event */
    }
}

static INT  get_registers(PEGASUS_T *pegasus, UINT16 indx, UINT16 size,
VOID *data)
{
    INT     ret;
    UINT8   *dma_data;

    while (pegasus->flags & ETH_REGS_CHANGED)
    {
        pegasus->flags |= CTRL_URB_SLEEP;
        USB_printf("ETH_REGS_CHANGED waiting...\n");
        NU_Retrieve_Events(&pegasus->events, 1, NU_AND,
                                   (unsigned long *)&ret, NU_SUSPEND);
    }

    dma_data = (UINT8 *)USB_malloc(size, BOUNDARY_WORD);
    if (!dma_data)
        return -ENOMEM;

    pegasus->dr->requesttype = PEGASUS_REQT_READ;
    pegasus->dr->request = PEGASUS_REQ_GET_REGS;
#ifdef LITTLE_ENDIAN
    pegasus->dr->value = 0;
    pegasus->dr->index = indx;
    pegasus->dr->length = size;
#else
    pegasus->dr->value = USB_SWAP16(0);
    pegasus->dr->index = USB_SWAP16(indx);
    pegasus->dr->length = USB_SWAP16(size);
#endif
    pegasus->ctrl_urb.transfer_buffer_length = size;
```

```
    FILL_CONTROL_URB(&pegasus->ctrl_urb, pegasus->usb,
                     usb_rcvctrlpipe(pegasus->usb,0),
                    (UINT8 *)pegasus->dr,
                     dma_data, size, ctrl_callback, pegasus );

    pegasus->flags |= CTRL_URB_SLEEP;
    NU_Set_Events(&pegasus->events, 0, NU_AND);  /* clear event */
    USB_SubmitUrb(&pegasus->ctrl_urb);
    NU_Retrieve_Events(&pegasus->events, 1, NU_AND,
                      (unsigned long *)&ret, NU_SUSPEND);
    memcpy(data, dma_data, size);
out:
    USB_free(dma_data);
    return  ret;
}
```

In the above example, the device driver first prepare the device request command in <*pegasus->dr*>, which was later referred to by <*urb->setup_packet*>. It request a buffer for DMA transfer by ***USB_malloc()***. Note that ***USB_malloc()*** will allocate a non-cacheable memory buffer. It then created a Control-In pipe by using ***usb_rcvctrlpipe*** macro, and the endpoint number is 0. The device driver the use the ***FILL_CONTROL_URB*** macro facility to fill the URB. The callback function is ***ctrl_callback()***, which is provided by the device driver itself. After submitting the URB, the caller task suspend on waiting the <*pegasus->events*> event set. On completion of this URB, the USB system software will invoke ***ctrl_callback()***, and ***ctrl_callback()*** will set the <*pegasus->events*> event to wake up the caller task.

3.3 Bulk Transfer

IN this section, we will introduce how to make bulk transfers by URBs. The URB provided <***transfer_buffer***> and <***transfer_buffer_length***> to accommodate data to be transferred to or from device. The direction of transfer is determined by the direction bit of bulk pipe. The transfer length is unlimited. If you are familiar with OpenHCI specification, you may understand that the maximum transfer size of a bulk transfer is 4096 bytes. If the transfer length of your URB exceeds 4096 bytes, the USB system software will split it into several transfer units smaller than 4096 bytes. Thus, you can specify unlimited transfer buffer length, only the physical memory can limit the size.

The transfer buffer must be non-cacheable. A designer can use ***USB_malloc()*** to acquire a block of non-cacheable memory.

The USB device driver also has to prepare a callback function to be invoked by the USB system software. The callback function will be invoked on the completion of URB, in spite of success or fail. Generally, the callback function is responsible for waking up the task that delivered the URB. The callback function may also check the status of the URB to determine the transfer is successful or not. The following is an example of bulk transfer.

```
/* In Host Controller HISR context */
static VOID write_bulk_callback(URB_T *urb)
{
    PEGASUS_T         *pegasus = urb->context;
    STATUS             previous_int_value;
    DV_DEVICE_ENTRY   *device;

    _PegasusDevice->tx_ready = 1;
    /* Get a pointer to the device. */
    device = DEV_Get_Dev_By_Name("Pegasus");

    /* Lock out interrupts. */
    previous_int_value = NU_Control_Interrupts(NU_DISABLE_INTERRUPTS);
    DEV_Recover_TX_Buffers(device);

    /* If there is another item on the list, transmit it. */
    if (device->dev_transq.head)
    {
        /* Re-enable interrupts */
        NU_Control_Interrupts(previous_int_value);
        /* Transmit the next packet. */
        PegasusTransmit(device, device->dev_transq.head);
    }
    /* Re-enable interrupts. */
```

```
    NU_Control_Interrupts(previous_int_value);

    if (urb->status)
        USB_printf("write_bulk_callback - TX error status: %d\n",
                   urb->status);
}


STATUS  PegasusTransmit(DV_DEVICE_ENTRY *dev, NET_BUFFER *netBuffer)
{
    INT      ret, wait=0;
    UINT8    *buf_ptr;
    INT      totalLength = 0;

    while (!_PegasusDevice->tx_ready)
    {
        NU_Sleep(1);                  /* wait on any outgoing Tx */
        if (wait++ > NU_PLUS_Ticks_Per_Second)
        {
            USB_printf("Can't transmit packet!\n");
            return NU_IO_ERROR;
        }
    }

    buf_ptr = _PegasusDevice->tx_buff + 2;
    do
    {
        memcpy(buf_ptr, netBuffer->data_ptr, netBuffer->data_len);
        totalLength += netBuffer->data_len;
        buf_ptr += netBuffer->data_len;

        /* Move on to the next buffer. */
        netBuffer = netBuffer->next_buffer;
    } while (netBuffer != 0);

    /* The first two bytes record the packet length. */
    buf_ptr = _PegasusDevice->tx_buff;
    buf_ptr[0] = totalLength & 0xff;
    buf_ptr[1] = (totalLength >> 8) & 0xff;
```

```
    FILL_BULK_URB(&_PegasusDevice->tx_urb, _PegasusDevice->usb,
                  usb_sndbulkpipe(_PegasusDevice->usb, 2),
                 (CHAR *)buf_ptr, PEGASUS_MAX_MTU,
                  write_bulk_callback, _PegasusDevice);

    _PegasusDevice->tx_urb.transfer_buffer_length =
            ((totalLength+2) & 0x3f) ? totalLength+2 : totalLength+3;
    _PegasusDevice->tx_ready = 0;
    USB_SubmitUrb(&_PegasusDevice->tx_urb);
    return NU_SUCCESS;
}
```

3.4 Interrupt Transfer

IN this section, we will introduce how to make interrupt transfers by URBs. The URB provided <***transfer_buffer***> and <***transfer_buffer_length***> to accommodate data to be transferred to or from device, and <***interval***> to specify polling interval of the interrupt transfer. The direction of transfer is determined by the direction bit of interrupt pipe. The transfer length is dependent on target interrupt endpoint.

The transfer buffer must be non-cacheable. A designer can use ***USB_malloc()*** to acquire a block of non-cacheable memory.

The USB device driver also has to prepare a callback function to be invoked by the USB system software. The callback function will be invoked if there's data received in one of the interrupt interval. In the callback function, USB device driver can read <***transfer_buffer***> to retrieve received interrupt data. The USB device driver have not to modify URB or resend URB. The USB library will resend the interrupt URB after callback. The interrupt URB will not stop until hardware failure or explicitly deleted by the USB device driver.

```
static VOID  intr_callback(URB_T *urb)
{
```

```
    PEGASUS_T *pegasus = urb->context;
    UINT8    *d;

    if (!pegasus)
        return;

    switch (urb->status)
    {
        case USB_ST_NOERROR:
            break;
        case USB_ST_URB_KILLED:
            return;
        default:
            break;
    }

    d = urb->transfer_buffer;
    if (d[2] & 0x1)
        UART_printf("Rx error - overflow!!\n");
}



    FILL_INT_URB(&_PegasusDevice->intr_urb, _PegasusDevice->usb,
                 usb_rcvintpipe(_PegasusDevice->usb, 3),
                 (CHAR *)&_PegasusDevice->intr_buff[0], 8,
                 intr_callback, _PegasusDevice,
                _PegasusDevice->intr_interval);
    res = USB_SubmitUrb(&_PegasusDevice->intr_urb);
    if (res)
        UART_printf("pegasus_open - failed intr_urb %d\n", res);
```

4. USB Library Provided API

❖ *InitUsbSystem*

Prototype	INT InitUsbSystem (VOID)
Description	Initialize the USB hardware and USB core library. This function must be invoked before any other functions. The USB library will scan device at this time, but the device will not be activated until the corresponding device driver was registered by USB_RegisterDriver().
Input	None
Output	None
Return	0 - Success Otherwise - Failure
See also	
Example code	/* * Initialize USBD, HC driver, hub driver, and register all other * USB device drivers. */ InitUsbSystem(); UMAS_InitUmasDriver(); UsbPrinter_Init();

❖ *UMAS_InitUmasDriver*

Prototype	INT UMAS_InitUmasDriver (VOID)
Description	Initialize the USB mass storage driver. fsInitFileSystem() and InitUsbSystem() must be called prior to this API. Once an USB mass storage device detected, USB core library will initialize it and mount it to NVTFAT file system automatically.
Input	None
Output	None
Return	0 - Success Otherwise - Failure
See also	
Example code	/* * Initialize NVTFAT FAT file system, USB core system, and USB mass storage driver */ fsInitFileSystem(); InitUsbSystem(); UMAS_InitUmasDriver(); .. .

❖ *USB_RegisterDriver*

Prototype	INT USB_RegisterDriver (USB_DRIVER_T *driver)
Description	Register a device driver with the USB library. In this function, USB library will also try to associate the newly registered device driver with all connected USB devices that have no device driver associated with it. Note that a connected USB device can be detected by USB library but may not work until it was associated with its corresponding device driver.
Input	<driver> - The USB device driver to be registered with USB core library
Output	None
Return	0 - Success Otherwise - Failure
See also	*USB_DeregisterDriver*
Example code	see below

```
static USB_DRIVER_T  usblp_driver =
{
    "usblp",
    usblp_probe,
    usblp_disconnect,
    {NULL,NULL},
    {0},
    NULL,
    usblp_ids,
    NULL,
    NULL
};

INT  UsbPrinter_Init() {
       if (USB_RegisterDriver(&usblp_driver)) return -1;
       return 0;
}
```

❖ *USB_DeregisterDriver*

Prototype	VOID USB_DeregisterDriver(USB_DRIVER_T *driver)
Description	Deregister a device driver
Input	<driver> - The device driver to be deregistered
Output	None
Return	0 - Success Otherwise - Failure
See also	*USB_RegisterDriver*

Example code	VOID UsbPrinter_Exit()
	{
	        USB_DeregisterDriver(&usblp_driver);
	}

❖ *USB_AllocateUrb*

Prototype	URB_T *USB_AllocateUrb(INT iso_packets)
Description	Creates an urb for the USB driver to use and returns a pointer to it. The driver should call USB_FreeUrb() when it is finished with the urb.
Input	<iso_packets> - The number of isochronous frames in a single URB. For other transfer types, this value must be zero.
Output	None
Return	NULL - Failure Otherwise - A pointer to the newly allocated URB
See also	*USB_FreeUrb, USB_SubmitUrb, USB_UnlinkUrb*

Example code	`_W99683_Camera->sbuf[i].urb = USB_AllocateUrb(FRAMES_PER_DESC);`
	`if (_W99683_Camera->sbuf[i].urb == NULL)`
	`{`
	`    UART_printf("%s - USB_AllocateUrb(%d.) failed.\n", proc,`
	`                FRAMES_PER_DESC);`
	`    Return -1;`
	`}`

❖ *USB_FreeUrb*

Prototype	VOID USB_FreeUrb(URB_T *urb)
Description	Frees the memory used by a urb
Input	<urb> - pointer to the URB to free
Output	None
Return	None
See also	*USB_AllocateUrb, USB_SubmitUrb, USB_UnlinkUrb*
Example code	

❖ *USB_SubmitUrb*

Prototype	INT USB_SubmitUrb(URB_T *urb)
Description	Submit a URB for executing data transfer
Input	<urb> - Pointer to the URB to be serviced
Output	None
Return	0 - Success Otherwise - Failure
See also	*USB_AllocateUrb, USB_FreeUrb, USB_UnlinkUrb*
Example code	(see below)

```
/* prepare URB */
FILL_BULK_URB(&_PegasusDevice->tx_urb, _PegasusDevice->usb,
            usb_sndbulkpipe(_PegasusDevice->usb, 2),(CHAR *)buf_ptr, PEGASUS_MAX_MTU,
            write_bulk_callback, _PegasusDevice);

/* set the data length to be transferred */
_PegasusDevice->tx_urb.transfer_buffer_length =
                            ((totalLength+2) & 0x3f) ? totalLength+2 : totalLength+3;
_PegasusDevice->tx_ready = 0;

/* submit URB */
if (USB_SubmitUrb(&_PegasusDevice->tx_urb) != 0)
{
    UART_printf("Warning - failed tx_urb %d\n", ret);
    return NU_IO_ERROR;
}
```

❖ *USB_UnlinkUrb*

Prototype	INT USB_UnlinkUrb(URB_T *urb)
Description	Unlink a URB which has been submitted but not finished
Input	<urb> - pointer to the URB to be unlinked
Output	None
Return	0 - Success Otherwise - Failure
See also	*USB_AllocateUrb, USB_FreeUrb, USB_SubmitUrb*
Example code	See below

```
INT  PegasusClose()
{
    _PegasusDevice->flags &= ~PEGASUS_RUNNING;

    if (!(_PegasusDevice->flags & PEGASUS_UNPLUG))
        disable_net_traffic(_PegasusDevice);

    USB_UnlinkUrb(&_PegasusDevice->rx_urb);
    USB_UnlinkUrb(&_PegasusDevice->tx_urb);
    USB_UnlinkUrb(&_PegasusDevice->ctrl_urb);
#ifdef  PEGASUS_USE_INTR
    USB_UnlinkUrb( &_PegasusDevice->intr_urb );
#endif
    return 0;
}
```

❖ *USB_SendControlMessage*

Prototype	INT USB_SendControlMessage(USB_DEV_T *dev, UINT32 pipe, UINT8 request, UINT8 rtype, UINT16 value, UINT16 index, V OID *data, UINT16 size, INT timeout)
Description	Builds a control urb, sends it off and waits for completion. This function sends a simple control message to a specified endpoint and waits for the message to complete, or timeout. Don't use this function from within an interrupt context.
Input	<dev> - pointer to the usb device to send the message to <pipe> - endpoint "pipe" to send the message to <request>- USB message request value <rtypr> - USB message request type value <value> - USB message value <index> - USB message index value <data> - pointer to the data to send <size> - length in bytes of the data to send <timeout>- time to wait for the message to complete before timing out (if 0 the wait is forever)
Output	None
Return	0 - Success Otherwise - Failure
See also	*USB_SendBulkMessage*

Example code	dma_data = USB_malloc(len, BOUNDARY_WORD);
	retval = USB_SendControlMessage(usblp->dev,
	dir ? usb_rcvctrlpipe(usblp->dev, 0) : usb_sndctrlpipe(usblp->dev, 0),
	request, USB_TYPE_CLASS | dir | recip, value, usblp->ifnum, dma_data,
	len, HZ * 5);
	memcpy(buf, dma_data, len);
	USB_free(dma_data);

❖ *USB_SendBulkMessage*

Prototype	INT USB_SendBulkMessage(USB_DEV_T *dev, UINT32 pipe, VOID *data, INT len, INT *actual_length, INT timeout)
Description	Builds a bulk urb, sends it off and waits for completion. This function sends a simple bulk message to a specified endpoint and waits for the message to complete, or timeout. Don't use this function from within an interrupt context.
Input	<dev> - pointer to the usb device to send the message to <pipe> - endpoint "pipe" to send the message to <data> - pointer to the data to send <len> - length in bytes of the data to send <actual_length> - pointer to a location to put the actual length transferred in bytes <timeout>- time to wait for the message to complete before timing out (if 0 the wait is forever)
Output	None
Return	0 - Success Otherwise - Failure
See also	*USB_SendControlMessage*
Example code	see below

```
if (!pb->pipe)
    pipe = usb_rcvbulkpipe (s->usbdev, 2);
else
    pipe = usb_sndbulkpipe (s->usbdev, 2);

ret = USB_SendBulkMessage(s->usbdev, pipe, pb->data, pb->size, &actual_length, 100);
if (ret<0) {
   err("dabusb: usb_bulk_msg failed(%d)",ret);
   if (usb_set_interface (s->usbdev, _DABUSB_IF, 1) < 0) {
       err("set_interface failed");
       return -EINVAL;
   }
}
```

❖ *USB_malloc*

Prototype	VOID *USB_malloc(INT wanted_size, INT boundary)
Description	Allocate a non-cacheable memory block started from assigned boundary. The total size of the USB library managed memory block is 256KB.
Input	<wanted_size> - The wanted size of non-cacheable memory block <boundary> - The start address boundary of the memory block. It cab be BOUNDARY_BYTE, BOUNDARY_HALF_WORD, BOUNDARY_WORD, BOUNDARY32, BOUNDARY64, BOUNDARY128, BOUNDARY256, BOUNDARY512, BOUNDARY1024, BOUNDARY2048, BOUNDARY4096
Output	None
Return	NULL - Failed, there is not enough memory or USB library is not started Otherwise - pointer to the newly allocated memory block
See also	*USB_free*
Example code	(see below)

```
UINT8  *dma_data;

dma_data = USB_malloc(len, BOUNDARY_WORD);
if (dma_data == NULL) {
    NU_printf("usblp_ctrl_msg - Memory not enough!\n");
    return -1;
}
retval = USB_SendControlMessage(usblp->dev,
       dir ? usb_rcvctrlpipe(usblp->dev, 0) : usb_sndctrlpipe(usblp->dev, 0),
        request, USB_TYPE_CLASS | dir | recip, value, usblp->ifnum, dma_data,
       len, HZ * 5);

memcpy(buf, dma_data, len);
USB_free(dma_data);
```

❖ *USB_free*

Prototype	VOID USB_free(VOID *alloc_addr)
Description	Free the memory block allocated by USB_malloc()
Input	<alloc_addr> - pointer to the USB_malloc() allocated memory block to be freed
Output	None
Return	None
See also	*USB_malloc*
Example code	See USB_malloc()