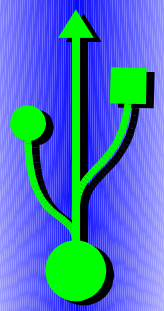
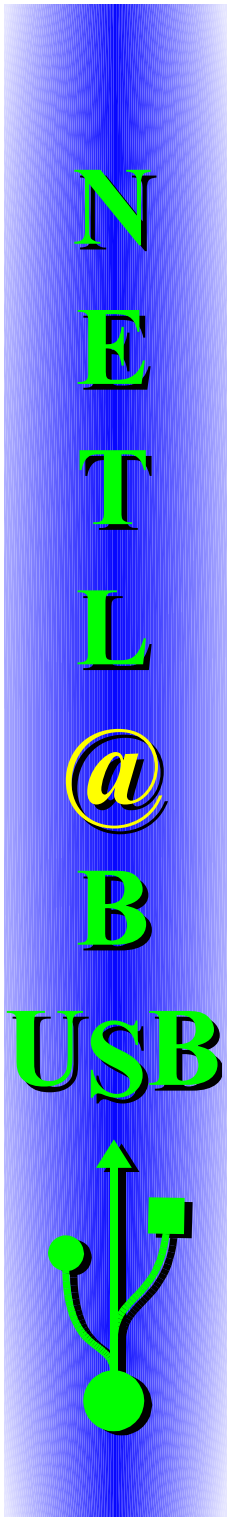
NETL@BUSB

# Developing USB device drivers on OS/2



By Markus Montkowski




# Agenda

- OS/2 and USB
  - OS/2 USB Driver Stack
  - IDC interconnections
  - Device attachment
    - Class driver
    - HID Driver

NETL@B USB

# Hostcontrolers on OS/2

UHCI

OHCI

EHCI (USB 2.0) not yett

NETL@B USB

# Drivers from IBM

Printer

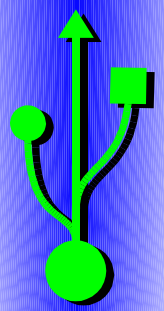
Keyboard / Mouse

Audio

Modem

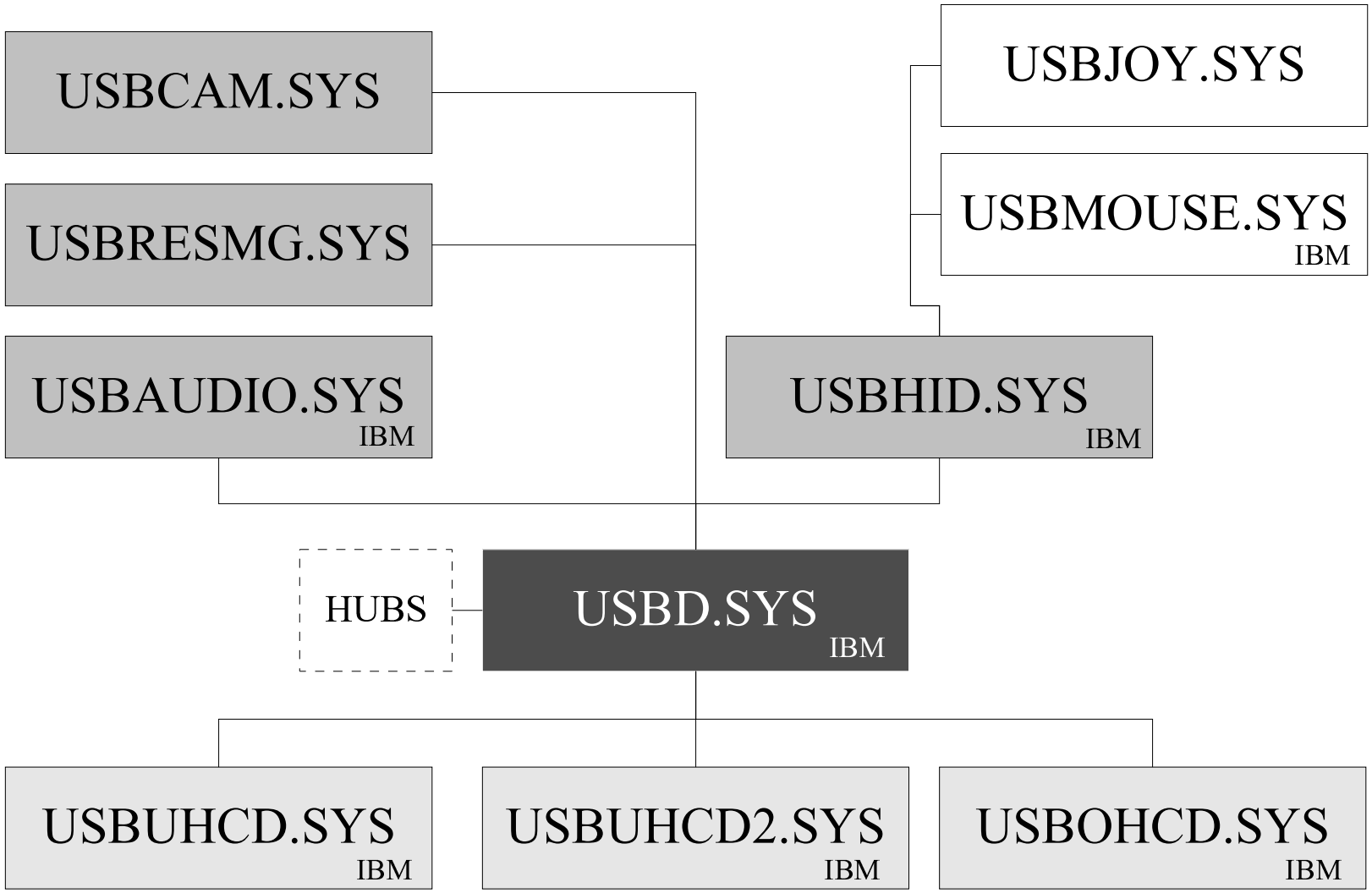
Mass Storage

Ethernet



NETL@B USB

# OS/2 USB Driver Stack



NETL@B USB

# IDC Interconnections

- Attachment
  - To USBD during Init
  - To USBHID during InitComplete
- IDC between USBD and Host Driver
- IDC between USBD and Class Driver
- IDC betweed UDBHID and HID Driver

NETL@BUSB

# IDC between USBD and Host Drv.

HCD → USBD

Category Host

- Register HCD

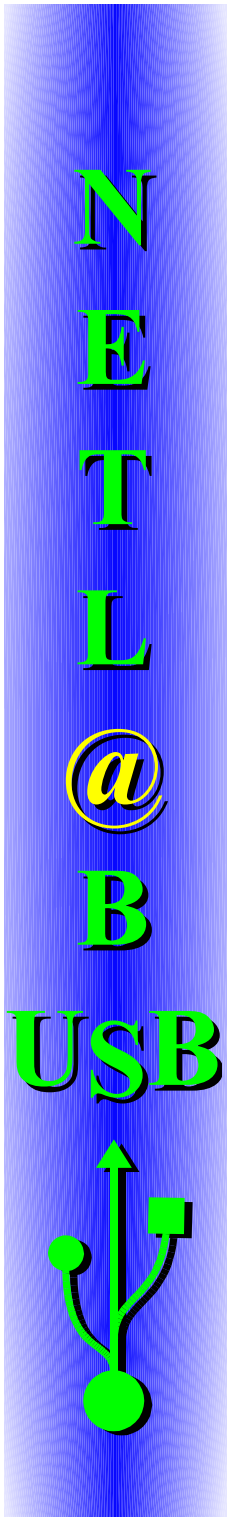
Category USBD

- Process IRQ
- Clear Stalled Pipe

USBD → HCD

Category Host

- Accept IO
- Cancel IO
- Reset Host




# IDC between USBD and Class Drv.

USBD ➔ Class

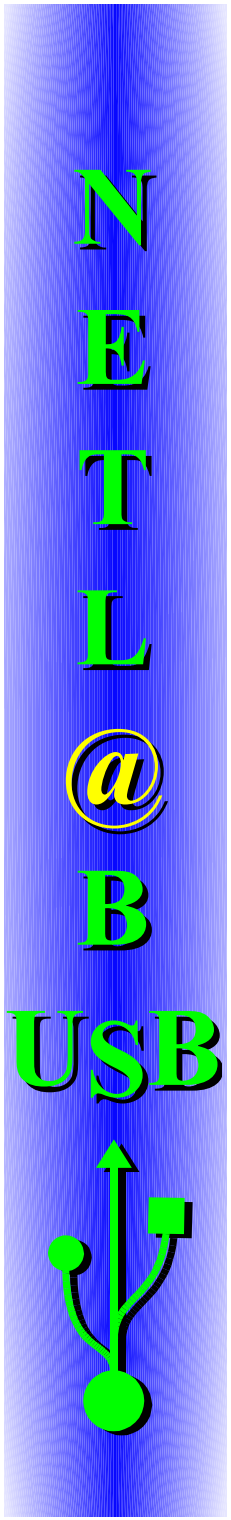
Category Class

- Process IRQ
- Check Service
- Detach Device

Class ➔ USBD

Category USBD

- Register Class
- Set Configuration
- Set Interface
- Accept IO
- Cancel IO




# IDC betweed UDBHID and HID drv.

USBHID ➔ HID

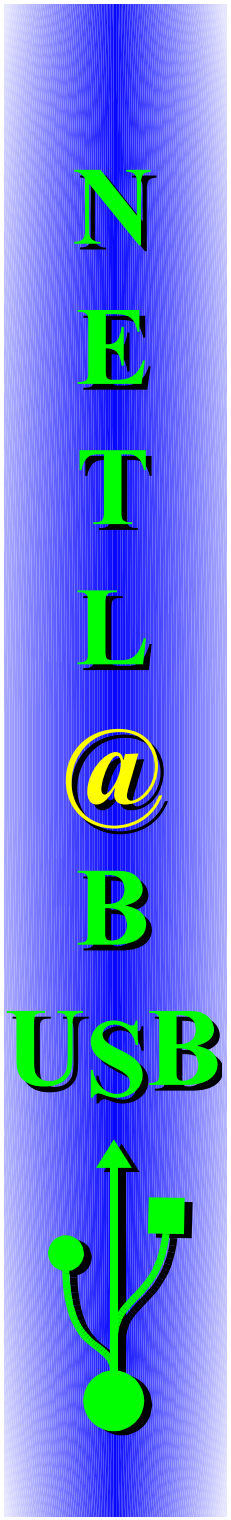
Category Client

- Check Service
- Detach Device
- Process IRQ

HID ➔ USBHID

Category Class

- Accept IO
- Clear stalled dev.




# Device Attachment

**UHCI Detects a change in the port status**

**The virtual RootHub dev. in UHCI signals HubStatusChanged to USBD**

**USBD Gets the new hub status and the status of the ports of the hub**

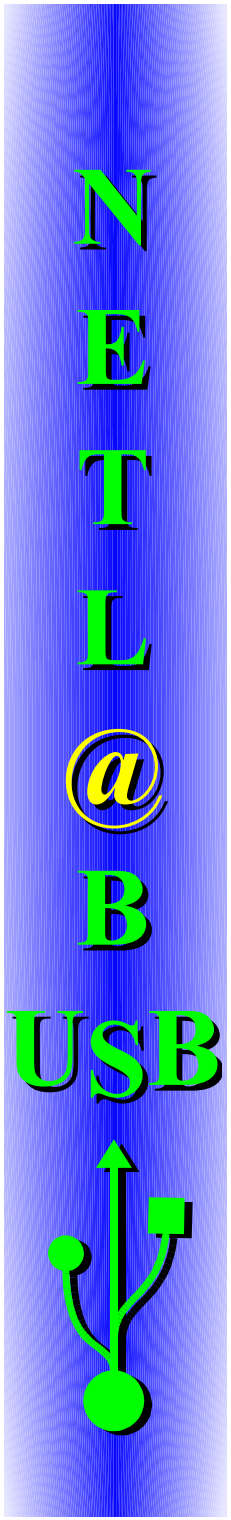
**USBD Gets the DeviceDescriptor**

**USBD Gets the length of the Configuration** (max. 512 Bytes)

**USBD Gets the Configuration**

**USBD looks for driver for this device**

- **Hub devices are handled by USBD itself**
- **Enums all registered classdrivers and calls IDC USB_IDC_FUNCTION_CHKSERV for non Hub devices**

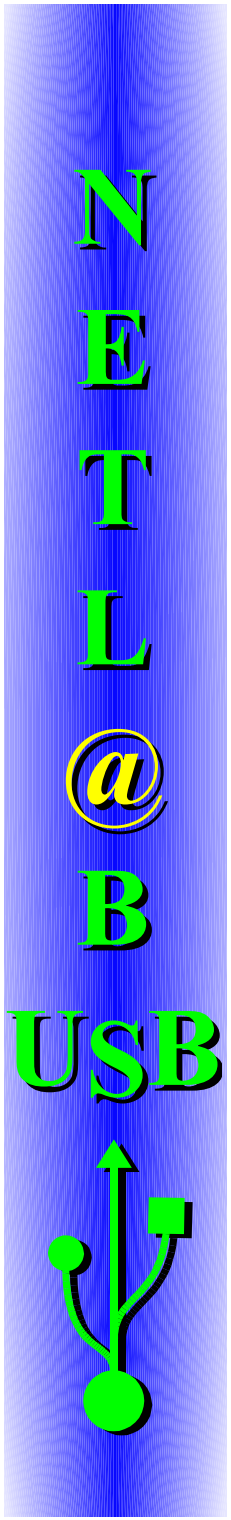



# The IDC entry, heart of an USB driver

Receives USB_IDC_FUNCTION_CHKSERV

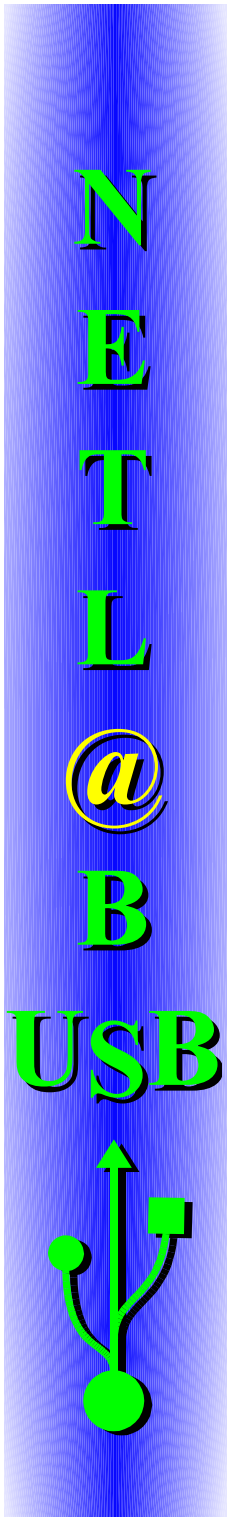
Receives USB_IDC_FUNCTION_DETDEV

Receives USB_IDC_FUNCTION_PRCIRQ




# An IDC Sample function

```
void far IDCEntry (PRP_GENIOCTL pRP)
{
   USHORT status = pRP->rph.Status;
   pRP->rph.Status = 0;
   if (pRP->rph.Cmd != CMDGenIOCTL || !pRP->ParmPacket){
     pRP->rph.Status |= STERR | USB_IDC_PARMERR;
   }
   else if (pRP->Category != USB_IDC_CATEGORY_CLASS){
     pRP->rph.Status |= STERR | USB_IDC_WRONGCAT;
   }
   else
   {
      switch (pRP->Function)
      {
      case  USB_IDC_FUNCTION_PRCIRQ:              // 0x44
            pRP->rph.Status = status;
            IRQSwitch (pRP);
            break;
      case  USB_IDC_FUNCTION_CHKSERV:             // 0x45
            Service (pRP);
            break;
      case  USB_IDC_FUNCTION_DETDEV:              // 0x46
            Detach (pRP);
            break;
      default:
            pRP->rph.Status |= STERR | USB_IDC_WRONGFUNC;
      }
   }
   pRP->rph.Status |= STDON;
}
```



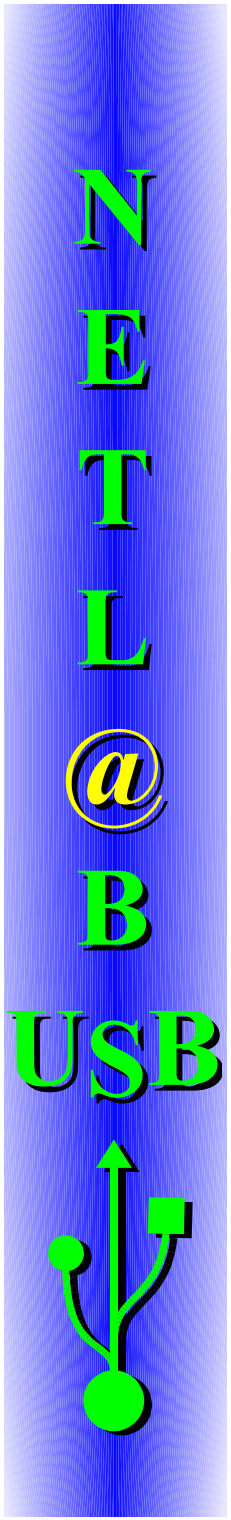

# Checking for device Support

## Class driver

- Check Vendor and Product ID
- Check Number of configurations
- Search for supported interface
- Check interface Needed Endpoints
- Set the Device Interface

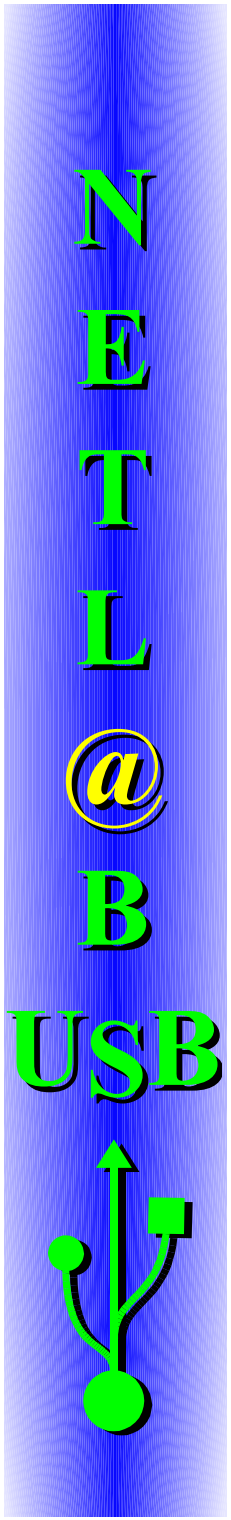
## HID device driver

- Check Usage (Page, Features ...)
- Get Report offsets
- Set Idle Time



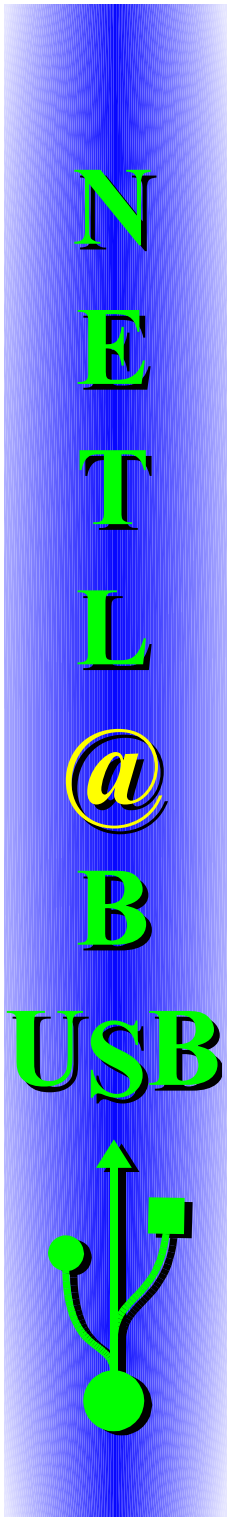

# Check Vendor/Product & configuration

```
void Service (PRP_GENIOCTL pRP_GENIOCTL)
{
  DeviceInfo     FAR   *pDevInfo;
   DeviceEndpoint FAR   *pEndPointD;
   USBSetConf       setConf;
   RP_GENIOCTL      rp;
   USHORT           VsrIndex, usInterface, usIFace;
   UCHAR            ucEndp;

  pDevInfo = ((USBCDServe FAR *)pRP_GENIOCTL->ParmPacket)->pDeviceInfo;
  if (pDevInfo->bConfigurationValue)
  {
    // already configured
    pRP_GENIOCTL->rph.Status = USB_IDC_RC_SERVREJCTD;
    return;
  }
  if ( pDevInfo->descriptor.idVendor  != VENDOR_HANDSPRING ||
       pDevInfo->descriptor.idProduct != PRODUCT_VISOR ||
       pDevInfo->descriptor.bcdDevice != DEVICE_RELEASE ||
       pDevInfo->descriptor.bNumConfigurations != 1)
  {
    pRP_GENIOCTL->rph.Status = USB_IDC_RC_SERVREJCTD;
    return;
  }

....
```




# USB Device Info

```
typedef struct _DeviceInfo
{
   UCHAR              ctrlID;               // (00) controller ID
   UCHAR              deviceAddress;        // (01) USB device address
   UCHAR              bConfigurationValue; // (02) USB device configuration
                                            //  value
   UCHAR              bInterfaceNumber;     // (03) 0 based index in
                                            //  interface array for this item
   UCHAR              lowSpeedDevice;       // (04) 0 for full speed device,
                                            //   nonzero - low speed device
   UCHAR              portNum;              // (05) port number to which
                                            //  device is attached
   USHORT             parentHubIndex;       // (06) index in hub table to
                                            //  parent hub,
                                            //  -1 for root hub device
   HDEVICE            rmDevHandle;          // (08) RM device handle
   SetupPacket        clearStalled;         // (12) setup packet for USBD
                                            //   internal use
   DeviceDescriptor   descriptor;           // (20) Device descriptor
   UCHAR              configurationData[MAX_CONFIG_LENGTH]; // (38) device
                                                  // configuration data
                                                  // (1062)

}  DeviceInfo;
```

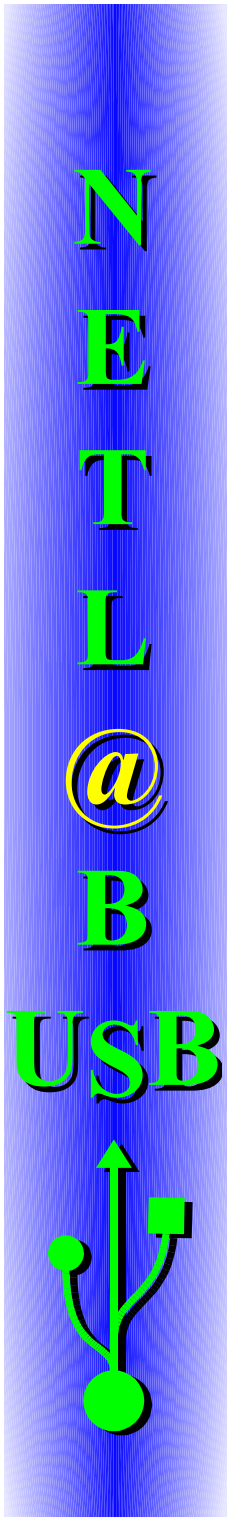



# USB Device Descriptor

```
Typedef struct _device_descriptor_
{
   UCHAR    bLength;              // (00) Size of descriptor in bytes
   UCHAR    bDescriptorType;      // (01) 0x01 - DEVICE Descriptor type
   USHORT   bcdUSB;               // (02) USB Specification Release Number
   UCHAR    bDeviceClass;         // (04) Class Code
   UCHAR    bDeviceSubClass;      // (05) SubClass Code
   UCHAR    bDeviceProtocol;      // (06) Protocol Code
   UCHAR    bMaxPacketSize0;      // (07) Maximum packet size for endpoint 0
   USHORT   idVendor;             // (08) Vendor ID
   USHORT   idProduct;            // (10) Product ID
   USHORT   bcdDevice;            // (12) Device release number
   UCHAR    iManufacturer;        // (14) Index of string descriptor
                                  //      describing manufacturer
   UCHAR    iProduct;             // (15) Index of string descriptor
                                  //      describing product
   UCHAR    iSerialNumber;        // (16) Index of string descriptor
                                  //      describing device's serial number
   UCHAR    bNumConfigurations;   // (17) Number of possible configurations
                                  // (18)
}  DeviceDescriptor;
```

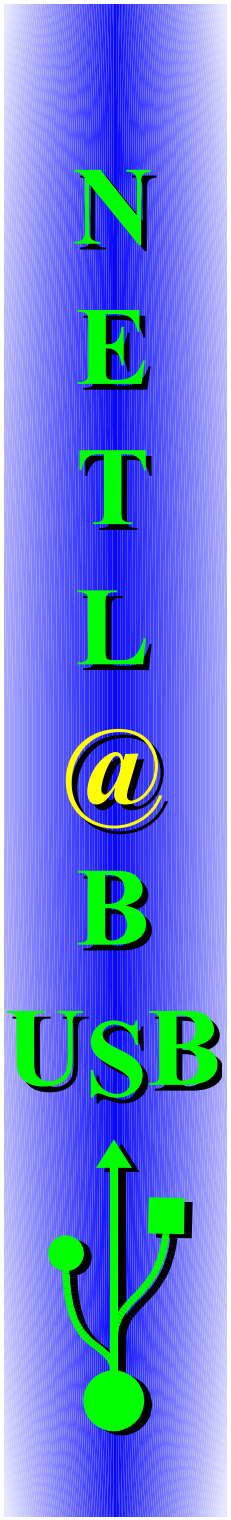
# Search for supported interface

```
...

  if (!(usIFace=SearchConfiguration((PUCHAR)&pDevInfo->configurationData,
                                    pDevInfo->descriptor.bNumConfigurations,
                                    INTERFACE_CLASS_VENDOR,
                                    INTERFACE_SUBCL_RESERVED,
                                    INTERFACE_PROTOCOL_RESERVED)))
  {
    pRP_GENIOCTL->rph.Status = USB_IDC_RC_SERVREJCTD;
    return;
  }

  pDevInfo->bConfigurationValue = LOBYTE(usInterface);

  //    Update global device list

  gVisors[VsrIndex].bConfValue = LOBYTE(usInterface);
  gVisors[VsrIndex].bInterface = HIBYTE(usInterface);

....
```



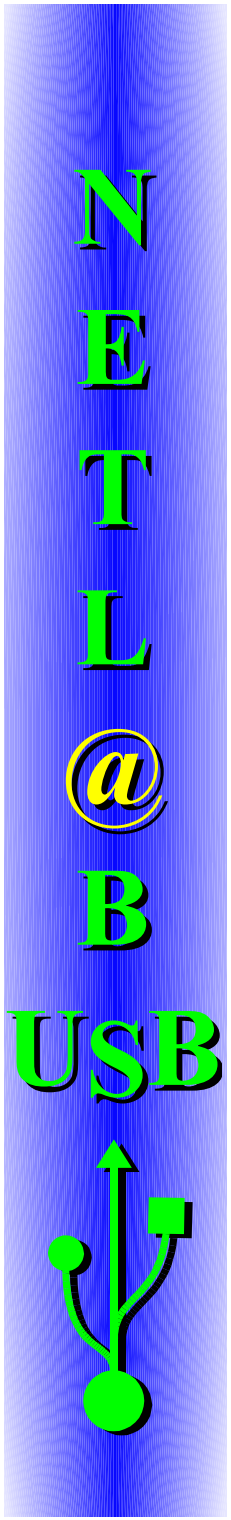

# Check for needed endpoints

```
...
  for(ucEndp=1;ucEndp<=VISOR_NUM_BULKPIPES;ucEndp++) {
    pEndPointD = GetEndpointDPtr ( pDevInfo->configurationData,
                 pDevInfo->descriptor.bNumConfigurations, gVisors[VsrIndex].bConfValue,
                 0, ucEndp  | DEV_ENDPT_DIRIN);
    if ( pEndPointD &&
         ((pEndPointD->bmAttributes & DEV_ENDPT_ATTRMASK) == DEV_ENDPT_BULK))
    {
      gVisors[VsrIndex].bInEndpoint[ucEndp] = ucEndp  | DEV_ENDPT_BULK;
      gVisors[VsrIndex].wMaxInSize[ucEndp]  = pEndPointD->wMaxPacketSize;
    }
    else
    {
      pRP_GENIOCTL->rph.Status = USB_IDC_RC_SERVREJCTD;
      return;
    }
    pEndPointD = GetEndpointDPtr ( pDevInfo->configurationData,
                 pDevInfo->descriptor.bNumConfigurations, gVisors[VsrIndex].bConfValue,
                 0, ucEndp | DEV_ENDPT_DIROUT);
    if ( pEndPointD &&
         ((pEndPointD->bmAttributes & DEV_ENDPT_ATTRMASK) == DEV_ENDPT_BULK)) {
      gVisors[VsrIndex].bOutEndpoint[ucEndp] = ucEndp  | DEV_ENDPT_DIROUT;
      gVisors[VsrIndex].wMaxOutSize[ucEndp]  = pEndPointD->wMaxPacketSize;
    }
    else {
      pRP_GENIOCTL->rph.Status = USB_IDC_RC_SERVREJCTD;
      return;
    }
  }
...
```



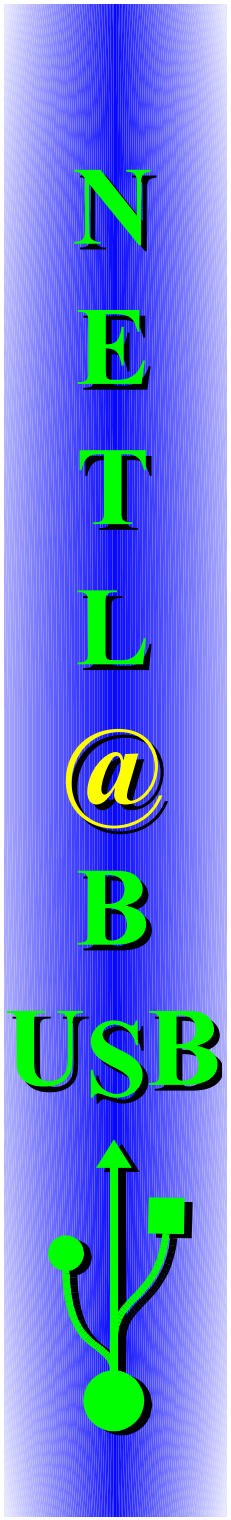

# Set device configuration

```
...
  gVisors[VsrIndex].pDeviceInfo = pDevInfo;
  gVisors[VsrIndex].active = TURNON;
  gNoOfVisors++;

  //   Set Visor Configuration. The request and the request's parameters
  //   are sent to the device in the setup packet.
  setConf.setConfiguration   = &gVisor.setPack;
  setConf.controllerId       = pDevInfo->ctrlID;
  setConf.deviceAddress      = pDevInfo->deviceAddress;
  setConf.classDriverDS      = GetDS();
  // desired configuration
  setConf.configurationValue = gVisors[VsrIndex].bConfValue;
  setConf.irqSwitchValue = VISOR_IRQ_SETCONF;
  setConf.category = USB_IDC_CATEGORY_CLASS;      // IRQ processor category
  setmem ((PSZ)&rp, 0, sizeof(rp));
  rp.rph.Cmd = CMDGenIOCTL;
  rp.Category = USB_IDC_CATEGORY_USBD;
  rp.Function = USB_IDC_FUNCTION_SETCONF;
  rp.ParmPacket = (PVOID)&setConf;

  USBCallIDC (gpUSBDIDC, gdsUSBDIDC, &rp);

  pRP_GENIOCTL->rph.Status = USB_IDC_RC_OK;
}
```




# USB I/O Request Block

```
typedef struct _USBRb {
   UCHAR    controllerId;  // (00) controller ID
   UCHAR    deviceAddress; // (01) USB dev. address. Valid  [1,127], 0 for unconfigured
  UCHAR    endPointId;    // (02) device endpoint ID, valid  [0,15]
   UCHAR    status;        // (03) device status on request complete
   USHORT   flags;         // (04) Low order byte sets  transfer type,
                           //      High order byte gives packet details
   PUCHAR   buffer1;       // (06) Virtual address of data buffer
   USHORT   buffer1Length; // (10) Buffer length in bytes
   PUCHAR   buffer2;       // (12) Virtual address of second data buffer.
   USHORT   buffer2Length; // (16) Buffer length in bytes
   USHORT   serviceTime;   // (18) Required service frequency in ms. Valid [0,255].
   USHORT   maxPacketSize; // (20) maximum packet size to be used for this endpoint
   PUSBIDCEntry   usbIDC;  // (22) Address of IRQ routine to be called for this request
   USHORT   usbDS;         // (26) DS value for IRQ processing routine
   UCHAR    category;      // (28) callers category (used in IRQ extension calls)
   ULONG    requestData1;  // (29) data to be stored within request
   ULONG    requestData2;  // (33) data to be stored within request
   ULONG    requestData3;  // (37) data to be stored within request
   UCHAR    maxErrorCount; // (41) max. error count. Valid  [0,3]. 0 - no error limit.
   struct _USBRb  FAR *nextRb;  // (42) far pointer to chained request block, not used
   UCHAR    altInterface;  // (46) alt interface index support,
                           //      used when USRB_FLAGS_ALT_INTF is on
   // fields used for isohronous requests (USRB_FLAGS_DET_ISOHR is set in 'flags'
   UCHAR    isoFlags;      // (47) ishronous request flags (opening call, regular call,
                          //      last call, cancel call, info call)
   USHORT   isoFrameLength;// (48) # of bytes to be sent in a frame (only opening call)
   USHORT   isoBuffers;    // (50) max no of active buffers( only opening call)
                           // (52)
} USBRB;
```

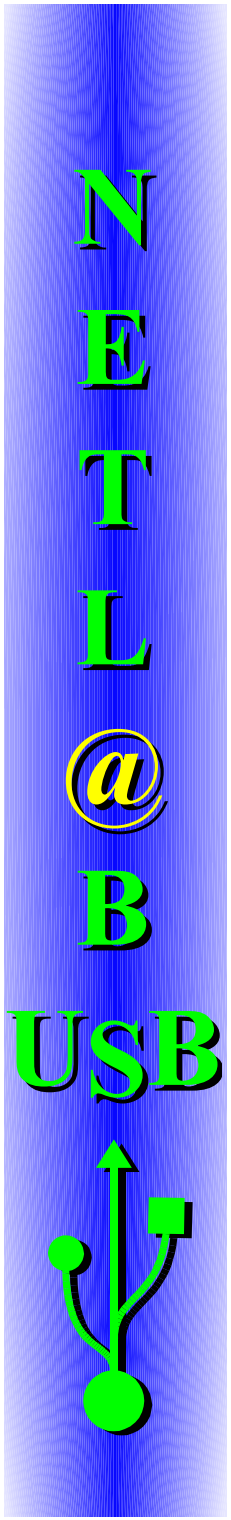



# Setup Packet

```
typedef struct _setup_Packet_{
   UCHAR       bmRequestType;  // (00) Characteristics of request
   UCHAR       bRequest;       // (01) Specific Request
   USHORT      wValue;         // (02) Word-sized field
                               //      (value depends on request)
   USHORT      wIndex;         // (04) typically Index or Offset
   USHORT      wLength;        // (06) Number of bytes to Transfer
}  SetupPacket;
```

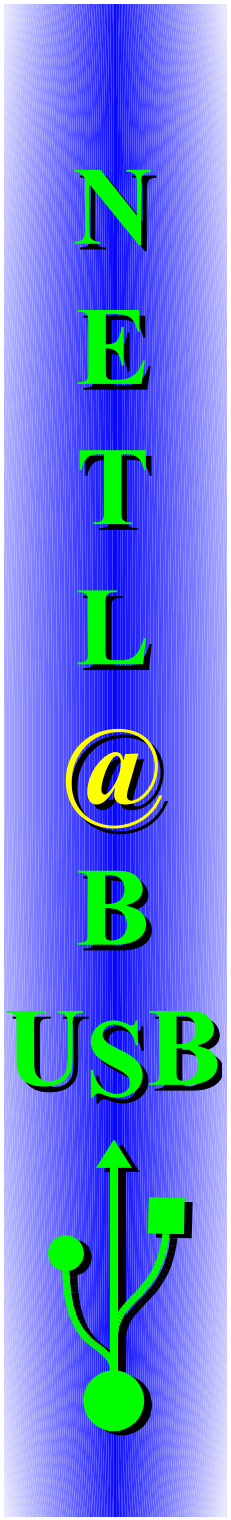
```
D7: Data transfer direction
  0 = Host-to-device
  1 = Device-to-host
D6...5: Type
  0 = Standard
  1 = Class
  2 = Vendor
  3 = Reserved
D4...0: Recipient
  0 = Device
  1 = Interface
  2 = Endpoint
  3 = Other
  4...31 = Reserved
```

```
 0 GET_STATUS
 1 CLEAR_FEATURE
 2 Reserved for future use
 3 SET_FEATURE
 4 Reserved for future use
 5 SET_ADDRESS
 6 GET_DESCRIPTOR
 7 SET_DESCRIPTOR
 8 GET_CONFIGURATION
 9 SET_CONFIGURATION
10 GET_INTERFACE
11 SET_INTERFACE
12 SYNCH_FRAME
```



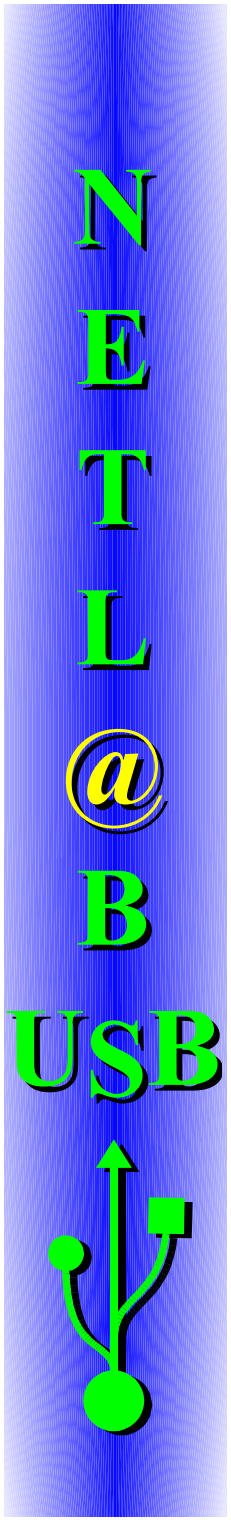

# HID Device attachment

```
void JOYserv (RP_GENIOCTL FAR *pRP_GENIOCTL)
{
   USHORT               index, joyIndex;
   USBHIDServe    FAR *pServData;
   ReportItemData FAR *pItem;
   USHORT         usOffSet;
   // Check for free entry

   if (gNoOfJOYs < MAX_JOYS)
   {
      for (joyIndex = 0; joyIndex < MAX_JOYS; joyIndex++)
        if (!gJOY[joyIndex].active)
          break;
   }
   else
   {
      pRP_GENIOCTL->rph.Status = USB_IDC_RC_SERVREJCTD;
      return;
   }

   pServData = (USBHIDServe FAR *)pRP_GENIOCTL->ParmPacket;


...
```



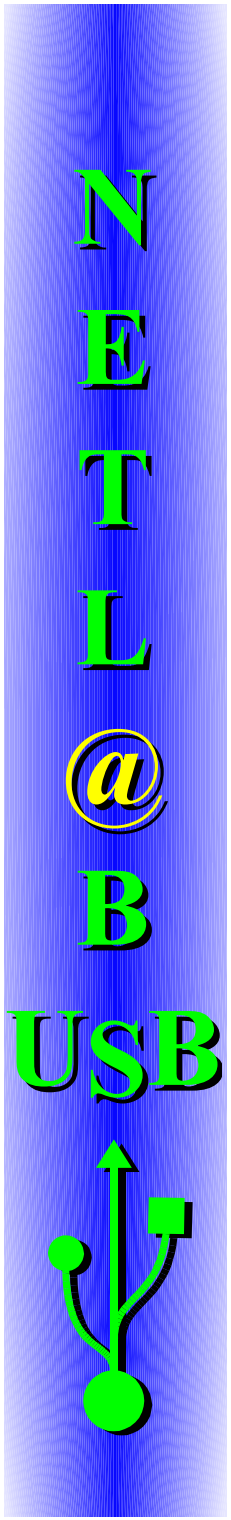

# USBHIDServe

```
typedef struct _USBHIDServe
{
  DeviceInfo FAR           *pDeviceInfo;    // far ptr to device data
  DeviceConfiguration FAR *devConf;         // far ptr to device config. data
  ReportItemData FAR       *itemData;       // ptr to report item data array
  ItemUsage FAR            *itemUsage;      // ptr to extra usage data array
  ItemDesignator FAR       *itemDesignator;// ptr to extra designator data array
  ItemString FAR           *itemString;     // ptr to extra string data array
  USHORT                   reportItemIndex;// starting report item index itemData
  USHORT                   versionFlags;    // specific version flags (HID drafts)
} USBHIDServe;
```




# Check Usage

```
...

  index = pServData->reportItemIndex;
  while (index != LAST_INDEX)
  {
    pItem = pServData->itemData + index;

    if ( pItem->mainType == HID_REPORT_TAGS_MAIN_COLL &&
         pItem->itemFeatures.usagePage == HID_USAGE_PAGE_GDESKTOP &&
         pItem->localFeatures.usageMin == HID_GDESKTOP_USAGE_JOYSTICK &&
         pItem->localFeatures.usageMax == HID_GDESKTOP_USAGE_JOYSTICK )
    {
      break;
    }
    index = pItem->indexToNextItem;
  }

  if (index == LAST_INDEX)
  {
    // no Joystick
      pRP_GENIOCTL->rph.Status = USB_IDC_RC_SERVREJCTD;
      return;
  }

...
```

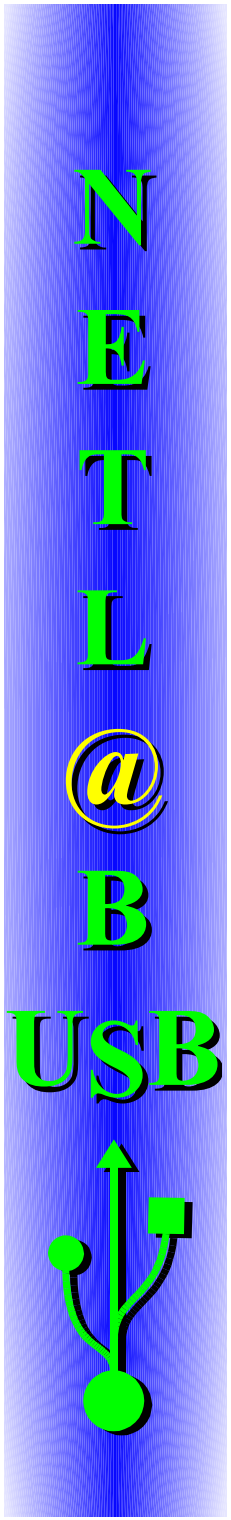



# ReportItemData

```
typedef struct _RepItemData
{
  UCHAR           used;             // 00 nonzero if allocated
  UCHAR           interface;        // 01 interface index
  UCHAR           mainType;         // 02 item type - input, output,
                                    //    feature, collection
  USHORT          itemFlags;        // 03 item flags
  USHORT          parColIndex;      // 05 parent collection index
                                    //    (LAST_INDEX - no parent collection)
  USHORT          indexToNextItem; // 07 index to next main item for this
                                    //    report
  // item features
  ItemFeatures    itemFeatures;     // 09

  // item local data
  LocalFeatures   localFeatures;    // 41
                                    // 59
}  ReportItemData;
```

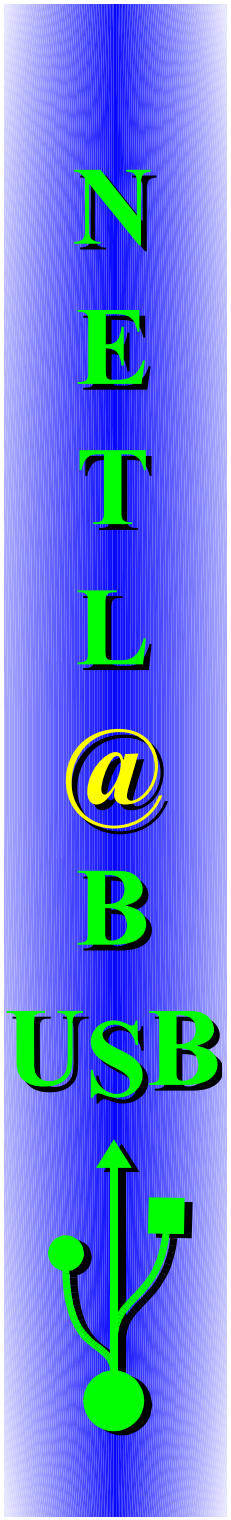
# ItemFeatures

```
typedef struct _item_features
{
  UCHAR         reportID;         // 09 report ID item belongs to
  ULONG         reportSize;       // 10 data size for this item
  ULONG         reportCount;      // 14 element count for current item
  USHORT        usagePage;        // 18 item's usage page
  LONG          logMin;           // 20 logical minimum for this item
  LONG          logMax;           // 24 logical maximum for this item
  LONG          phyMin;           // 28 physical value minimum
  LONG          phyMax;           // 32 physical value maximum
  ULONG         unit;             // 36 units of measurement
  UCHAR         unitExponent      // 40 exponent value
                                  // 41
}  ItemFeatures;
```



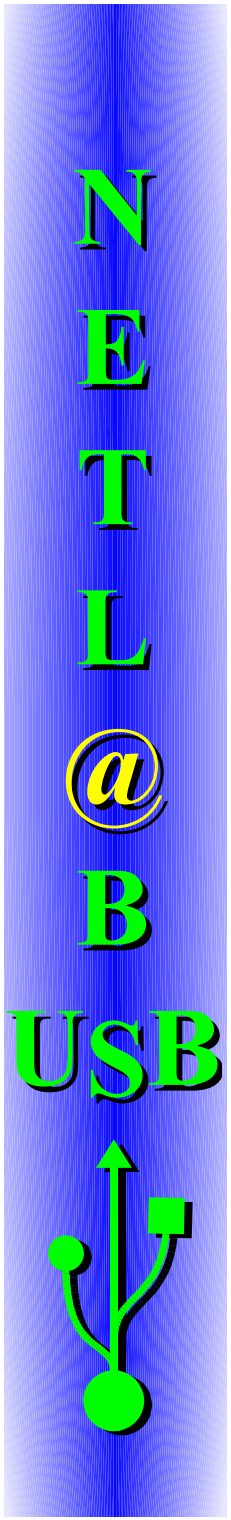

# LocalFeatures

```
typedef  struct   _local_features
{
  // usage information
  USHORT          usagePage;         // 41 local (only this item) usage page
  USHORT          usageMin;          // 43 usage minimum
  USHORT          usageMax;          // 45 usage maximum
  USHORT          indexToUsageList; // 47
  // physical data references
  USHORT          designatorMin;     // 49
  USHORT          designatorMax;     // 51
  USHORT          indexToDesignator;// 53
  // string data references
  UCHAR           stringMin;         // 55
  UCHAR           stringMax;         // 56
  USHORT          indexToStrings;    // 57
                                     // 59
}  LocalFeatures;
```



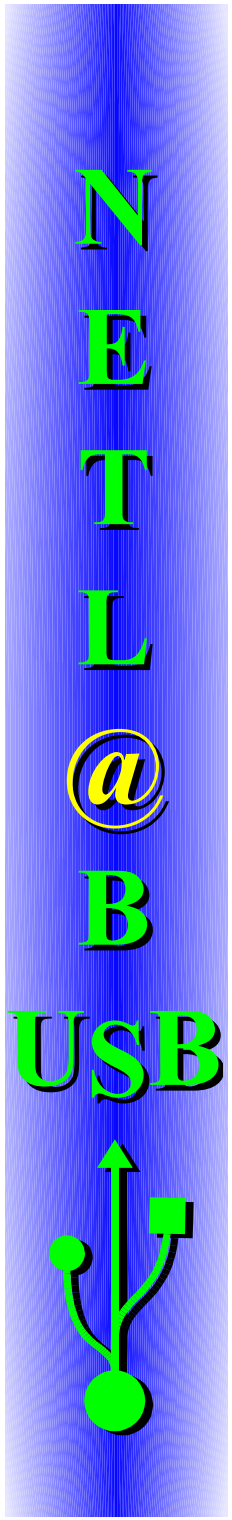

# Check length of report

```
...

  // Check if the total report Length of the device can be handled

  gJOY[joyIndex].ReportLength = 0;
  index = pServData->reportItemIndex;

  while (index != LAST_INDEX)
  {
    pItem = pServData->itemData + index;
    gJOY[joyIndex].ReportLength += pItem->itemFeatures.reportSize*
                                   pItem->itemFeatures.reportCount;
    index = pItem->indexToNextItem;
  }

  gJOY[joyIndex].ReportLength= (gJOY[joyIndex].ReportLength+BITS_IN_BYTE-1)
                               /BITS_IN_BYTE;

  if(gJOY[joyIndex].ReportLength > sizeof(gJOY[joyIndex].buffer))
  {
    // Report is to long
    pRP_GENIOCTL->rph.Status = USB_IDC_RC_SERVREJCTD;
    return;
  }
...
```



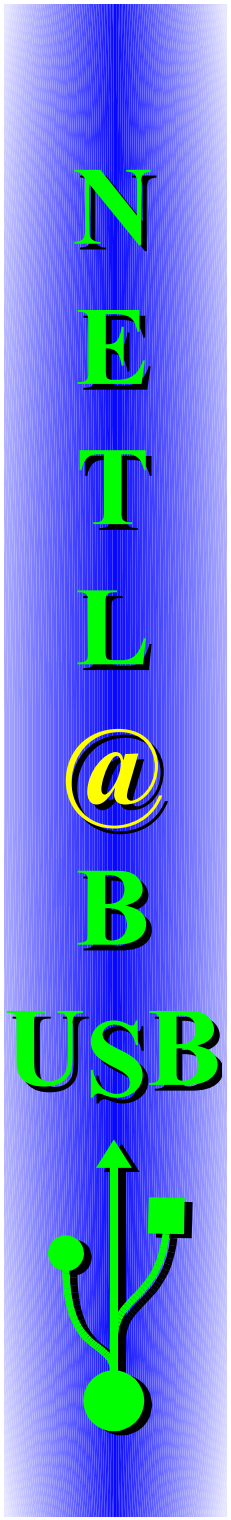

# Parse for needed reports

```
...
  index = pServData->reportItemIndex;
  usOffSet = 0;
  gJOY[joyIndex].ulCapsAxes = 0;
  gJOY[joyIndex].ulCapsSliders = 0;
  setmem((PSZ)&gJOY[joyIndex].DevCapsJoy,0,sizeof(DEVCAPS));
  setmem((PSZ)&gJOY[joyIndex].joyState,0,sizeof(JOYSTATE));
  setmem((PSZ)&gJOY[joyIndex].AxeUnits,0,sizeof(JOYAXEUNIT)*JOYMAX_AXES);
  setmem((PSZ)&gJOY[joyIndex].Items,FULL_BYTE,sizeof(JOYITEM)*JOYMAXITEMS);
  while (index != LAST_INDEX){
    pItem = pServData->itemData + index;

    if ( pItem->mainType == HID_REPORT_TAGS_MAIN_INPUT &&
         pItem->itemFeatures.usagePage == HID_USAGE_PAGE_GDESKTOP) {
      if ( pItem->localFeatures.usageMin >= HID_GDESKTOP_USAGE_X &&
           pItem->localFeatures.usageMax <= HID_GDESKTOP_USAGE_Z ) {
        usOffSet = SetupXYZAxes(joyIndex, pItem, usOffSet);
        gJOY[joyIndex].inInterface = pItem->interface;
      }
      else
      { ... }
    else
    { ... }
      index = pItem->indexToNextItem;
   }
...
```



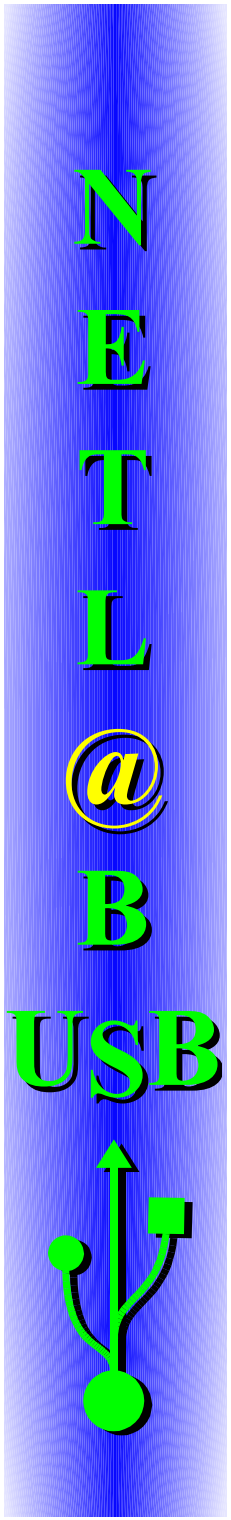

# Last Check and SetIdleTime

```
  if ( (ULONG)0==gJOY[joyIndex].DevCapsJoy.ulButtons ||
       (ULONG)0==gJOY[joyIndex].DevCapsJoy.ulAxes) {
    // No Axes or No buttons
    pRP_GENIOCTL->rph.Status = USB_IDC_RC_SERVREJCTD;
    return;
  }
  gJOY[joyIndex].joyAddr = pServData->pDeviceInfo->deviceAddress;
  gJOY[joyIndex].controllerID = pServData->pDeviceInfo->ctrlID;
  gJOY[joyIndex].interruptPipeAddress =
  GetInterruptPipeAddr( pServData->pDeviceInfo->configurationData,
                        pServData->pDeviceInfo->descriptor.bNumConfigurations,
                        pServData->pDeviceInfo->bConfigurationValue,
                        gJOY[joyIndex].inInterface);
  gJOY[joyIndex].setITpack.bmRequestType = REQTYPE_TYPE_CLASS |
                                           REQTYPE_RECIPIENT_INTERFACE;
  gJOY[joyIndex].setITpack.bRequest = HID_REQUEST_SET_IDLE;
  gJOY[joyIndex].setITpack.wValue = 0x0000;  // all reports only if changed
  gJOY[joyIndex].setITpack.wIndex = gJOY[joyIndex].inInterface;
  gJOY[joyIndex].setITpack.wLength = NULL;
  gJOY[joyIndex].active = TURNON;
  gNoOfJOYs++;

  SetIdleTime (joyIndex, JOY_IRQ_STATUS_IDLESET);

  pRP_GENIOCTL->rph.Status = USB_IDC_RC_OK;
}
```



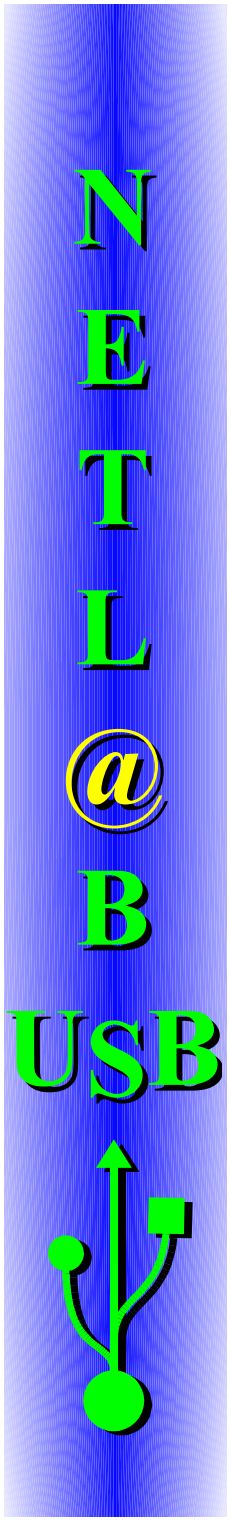

# SetIdleTime

```
void SetIdleTime (USHORT joyIndex, USHORT kbdIRQstatus) {
  USBRB       rbHID;     // I/O request block
  RP_GENIOCTL rpHID;     // request packet

  rbHID.buffer1       = (PUCHAR)&gJOY[joyIndex].setITpack;
  rbHID.buffer1Length = sizeof(gJOY[joyIndex].setITpack);
  rbHID.buffer2       = NULL;
  rbHID.buffer2Length = NULL;
  rbHID.controllerId  = gJOY[joyIndex].controllerID;
  rbHID.deviceAddress = gJOY[joyIndex].joyAddr;
  rbHID.endPointId    = USB_DEFAULT_CTRL_ENDPT;
  rbHID.status        = 0; // not used
  rbHID.flags         = USRB_FLAGS_TTYPE_SETUP;
  rbHID.serviceTime   = USB_DEFAULT_SRV_INTV;
  rbHID.maxPacketSize = USB_DEFAULT_PKT_SIZE;
  rbHID.maxErrorCount = USB_MAX_ERROR_COUNT;
  rbHID.usbIDC        = (PUSBIDCEntry)JOYidc;    // Address of IRQ processor function
  rbHID.usbDS         = GetDS();
  rbHID.category      = USB_IDC_CATEGORY_CLIENT; // set client layer as IRQ processor
  rbHID.requestData1  = JOY_IRQ_STATUS_IDLESET;  // MAKEULONG (kbdIRQstatus,0);
  rbHID.requestData2  = MAKEULONG (joyIndex, 0);
  rbHID.requestData3  = 0;                       // not used
  setmem((PSZ)&rpHID, 0, sizeof(rpHID));
  rpHID.rph.Cmd    = CMDGenIOCTL;
  rpHID.Category   = USB_IDC_CATEGORY_CLASS;
  rpHID.Function   = USB_IDC_FUNCTION_ACCIO;
  rpHID.ParmPacket = (PVOID)&rbHID;

  USBCallIDC (gpHIDIDC, gdsHIDIDC, (RP_GENIOCTL FAR *)&rpHID);
}
```




# Get Report Position

```
USHORT SetupPOVs(USHORT joyIndex, ReportItemData FAR *pItem, USHORT usOffset)
{
  USHORT usCount;
  usCount = 0;
  while( usCount < (USHORT)pItem->itemFeatures.reportCount &&
         gJOY[joyIndex].DevCapsJoy.ulPOVs < MAX_POVS)
  {
    gJOY[joyIndex].Items[JOYOFS_POV0+usCount].bReport    = pItem->itemFeatures.reportID;
    gJOY[joyIndex].Items[JOYOFS_POV0+usCount].usOffse    = usOffset;
    gJOY[joyIndex].Items[JOYOFS_POV0+usCount].usReportSize =
                                              (USHORT)pItem->itemFeatures.reportSize;
    gJOY[joyIndex].AxeUnits[JOYOFS_POV0+usCount].logMin = pItem->itemFeatures.logMin;
    gJOY[joyIndex].AxeUnits[JOYOFS_POV0+usCount].logMax = pItem->itemFeatures.logMax;
    gJOY[joyIndex].AxeUnits[JOYOFS_POV0+usCount].phyMin = pItem->itemFeatures.phyMin;
    gJOY[joyIndex].AxeUnits[JOYOFS_POV0+usCount].phyMax = pItem->itemFeatures.phyMax;
    gJOY[joyIndex].AxeUnits[JOYOFS_POV0+usCount].unit   = pItem->itemFeatures.unit;
    gJOY[joyIndex].AxeUnits[JOYOFS_POV0+usCount].unitExponent =
                                                  pItem->itemFeatures.unitExponent;
    gJOY[joyIndex].DevCapsJoy.ulPOVs++;
    usOffset += pItem->itemFeatures.reportSize;
    usCount++;
  }

  // Just in case the device has more than MAX_POVS Hatswitches
  usOffset += pItem->itemFeatures.reportSize  *
              (pItem->itemFeatures.reportCount- usCount);

  return usOffset;
}
```

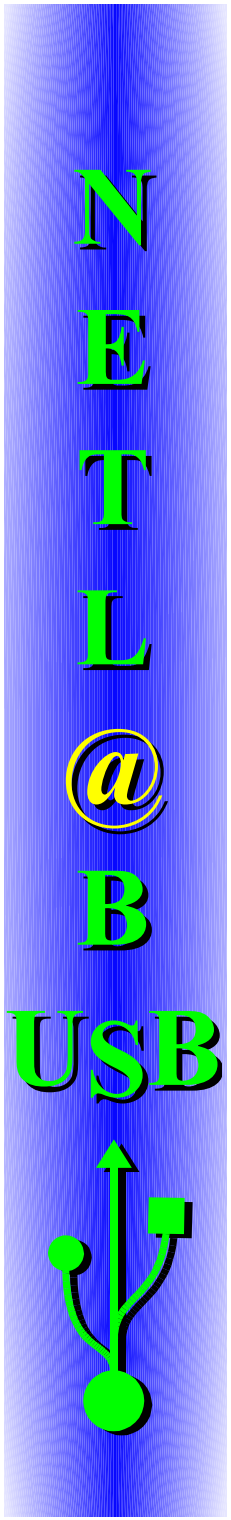



# Parse Report

```
void InterruptDataReceived (RP_GENIOCTL FAR *pRP_GENIOCTL)
{
  USBRB FAR *processedRB;
  BYTE      *pIntData;
  USHORT     joyIndex, i;
  LONG       lValue;
  processedRB = (USBRB FAR *)pRP_GENIOCTL->ParmPacket;
  joyIndex = LOUSHORT (processedRB->requestData2);

  if (gDevice)
    if (joyIndex != gJoyIndex)
      return;

  pIntData = (BYTE *)&gJOY[joyIndex].buffer;

  setmem((PSZ)&gJOY[joyIndex].joyState, 0, sizeof(JOYSTATE));
```

# Parse Report cont.

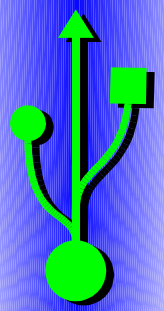
```
...
  i=0;
  while(i< (USHORT)gJOY[joyIndex].DevCapsJoy.ulPOVs)
  {
    lValue = GetLogValue(joyIndex, JOYOFS_POV0+i);
    if(lValue)
    {
      if( gJOY[joyIndex].AxeUnits[JOYOFS_POV0+i].unit)
      {
        // Assume degrees and log 1 as top which is 0°
        lValue = (lValue -1)*
                 (gJOY[joyIndex].AxeUnits[JOYOFS_POV0+i].phyMax-
                  gJOY[joyIndex].AxeUnits[JOYOFS_POV0+i].phyMin)/
                (gJOY[joyIndex].AxeUnits[JOYOFS_POV0+i].logMax-
                 gJOY[joyIndex].AxeUnits[JOYOFS_POV0+i].logMin);
        // Report in hundredths of degrees
        if( gJOY[joyIndex].AxeUnits[JOYOFS_POV0+i].phyMax>=270 &&
            gJOY[joyIndex].AxeUnits[JOYOFS_POV0+i].phyMax<=360)
          lValue *=100;
      }
      else
      {
        // No Units so no physical values translate to degrees
        lValue = (lValue-1) * (36000/gJOY[joyIndex].AxeUnits[JOYOFS_POV0+i].logMax);
      }
    }
    else
    {
      lValue = 0x0000FFFF; // centered
    }
    gJOY[joyIndex].joyState.rgdwPOV[i] = lValue;
    i++;
  }
...
```




# Item Usage, Designator and String

```
typedef struct _item_usage
{
   UCHAR           used;                  // nonzero if allocated
   USHORT          indexToNextUsageData;
   USHORT          usagePage;             // local (only this item) usage page
   USHORT          usageMin;              // usage minimum
   USHORT          usageMax;              // usage maximum
}  ItemUsage;

typedef struct _item_designator
{
   UCHAR           used;                        // nonzero if allocated
   USHORT          indexToNextDesignatorData;
   USHORT          designatorMin;               // designator minimum
   USHORT          designatorMax;               // designator maximum
}  ItemDesignator;

typedef struct _item_strings
{
   UCHAR           used;                    // nonzero if allocated
   USHORT          indexToNextStringData;
   UCHAR           stringMin;               // string minimum
   UCHAR           stringMax;               // string maximum
}  ItemString;
```

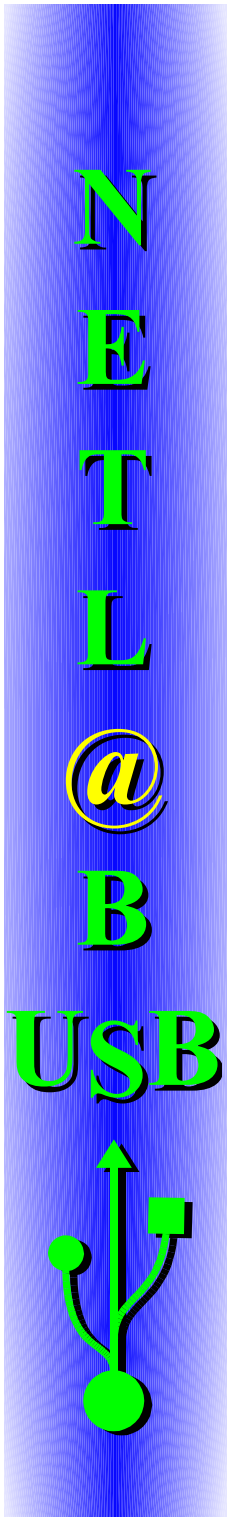
NETL@B USB



# The Brain, the IRQ function

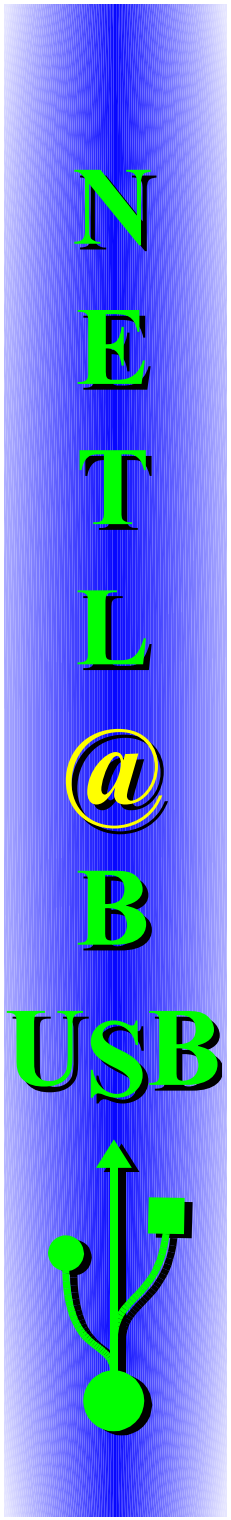
Gets called from the IDC function to process USB_IDC_FUNCTION_PRCIRQ

Each call to a USBCallIDC is followed by an IRQ which gets processed in it



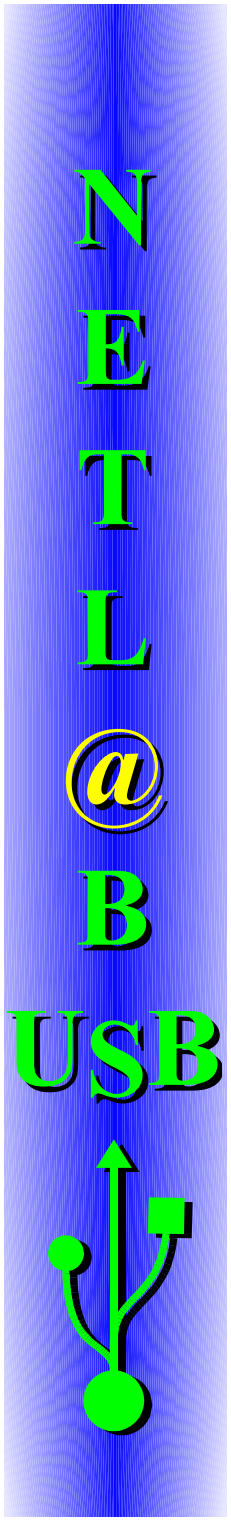

# Sample IRQ function

```
void JOYirq (RP_GENIOCTL FAR *pRP_GENIOCTL)
{
  USBRB FAR *processedRB;
  UCHAR oldCat;

  processedRB = (USBRB FAR *)pRP_GENIOCTL->ParmPacket;

  if(pRP_GENIOCTL->rph.Status != USB_IDC_RC_OK)
  {
    if (processedRB->status & USRB_STATUS_STALLED)
    {
      if( processedRB->requestData1!=JOY_IRQ_STATUS_STALLED )
      {
        oldCat=pRP_GENIOCTL->Category;
        JOYClearStalled(pRP_GENIOCTL);
        pRP_GENIOCTL->Category=oldCat;
      }
      return;
    }
    return;
  }

...
```



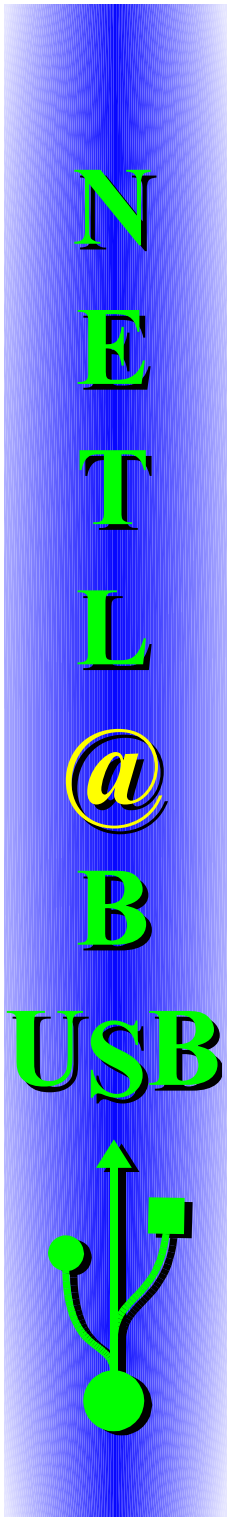

# IRQ function cont.

```
...

  switch (processedRB->requestData1)
  {
    case JOY_IRQ_STATUS_IDLESET:
      pRP_GENIOCTL->rph.Status=STATUS_DONE;    //always ok
      ReadInterruptPipe (pRP_GENIOCTL);
      break;
    case JOY_IRQ_STATUS_DURATION:
      break;
    case JOY_IRQ_STATUS_INTPIPE:
      InterruptDataReceived (pRP_GENIOCTL);
      ReadInterruptPipe (pRP_GENIOCTL);
      break;
    case JOY_IRQ_STATUS_SETACK:
      break;
    case JOY_IRQ_STATUS_STALLED:
      ReadInterruptPipe (pRP_GENIOCTL);
      break;
    default:;
  }
}
```



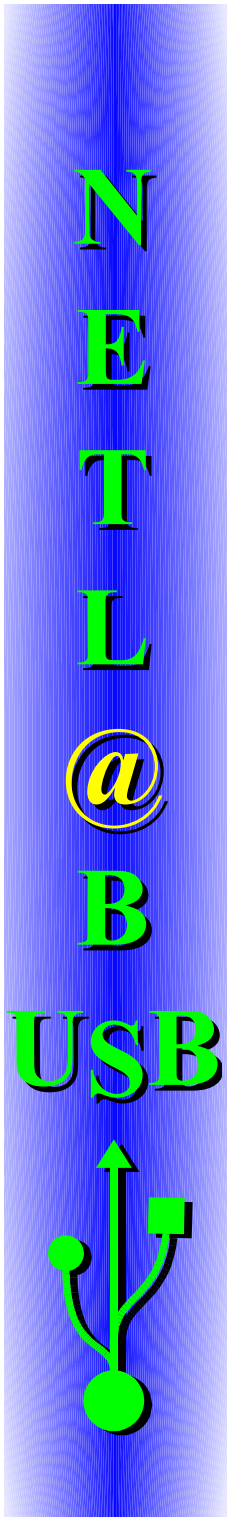

# ReadInterruptPipe

```
void ReadInterruptPipe (PRP_GENIOCTL pRP_GENIOCTL){
   USBRB FAR  *processedRB;
   USBRB       hcdReqBlock;
   RP_GENIOCTL rp_USBReq;
   USHORT      deviceIndex;

   processedRB = (USBRB FAR *)pRP_GENIOCTL->ParmPacket;
   deviceIndex = LOUSHORT(processedRB->requestData2);
   setmem((PSZ)gJOY[deviceIndex].buffer, UI_RESERV, sizeof(gJOY[deviceIndex].buffer));
   hcdReqBlock.controllerId  = processedRB->controllerId;
   hcdReqBlock.deviceAddress = processedRB->deviceAddress;             // use default address
   hcdReqBlock.endPointId    = gJOY[deviceIndex].interruptPipeAddress;
   hcdReqBlock.status        = 0;                                      // not used
   hcdReqBlock.flags         = USRB_FLAGS_TTYPE_IN | USRB_FLAGS_DET_INTRPT;
   if (!(processedRB->flags & USRB_FLAGS_DET_DTGGLEON))
     hcdReqBlock.flags |= USRB_FLAGS_DET_DTGGLEON;
   hcdReqBlock.buffer1       = (PUCHAR)gJOY[deviceIndex].buffer;
   hcdReqBlock.buffer1Length = gJOY[deviceIndex].ReportLength;
   hcdReqBlock.buffer2       = NULL;                       // no additonal data to be sent to/from host
   hcdReqBlock.buffer2Length = 0;                          // to complete this request
   hcdReqBlock.serviceTime   = USB_DEFAULT_SRV_INTV;
   hcdReqBlock.maxPacketSize = USB_DEFAULT_PKT_SIZE;
   hcdReqBlock.maxErrorCount = USB_MAX_ERROR_COUNT;
   hcdReqBlock.usbIDC        = (PUSBIDCEntry)JOYidc;       // Address of IRQ proc.
   hcdReqBlock.usbDS         = GetDS();
   hcdReqBlock.category      = USB_IDC_CATEGORY_CLIENT;    // set USBD layer as IRQ processor
   hcdReqBlock.requestData1  = JOY_IRQ_STATUS_INTPIPE;
   hcdReqBlock.requestData2  = MAKEULONG(deviceIndex, 0); // index in device table
   hcdReqBlock.requestData3  = 0;                          // not used
   setmem((PSZ)&rp_USBReq, 0, sizeof(rp_USBReq));
   rp_USBReq.rph.Cmd    = CMDGenIOCTL;
   rp_USBReq.Category   = USB_IDC_CATEGORY_CLASS;
   rp_USBReq.Function   = USB_IDC_FUNCTION_ACCIO;
   rp_USBReq.ParmPacket = (PVOID)&hcdReqBlock;

   USBCallIDC (gpHIDIDC, gdsHIDIDC, (RP_GENIOCTL FAR *)&rp_USBReq);
}
```



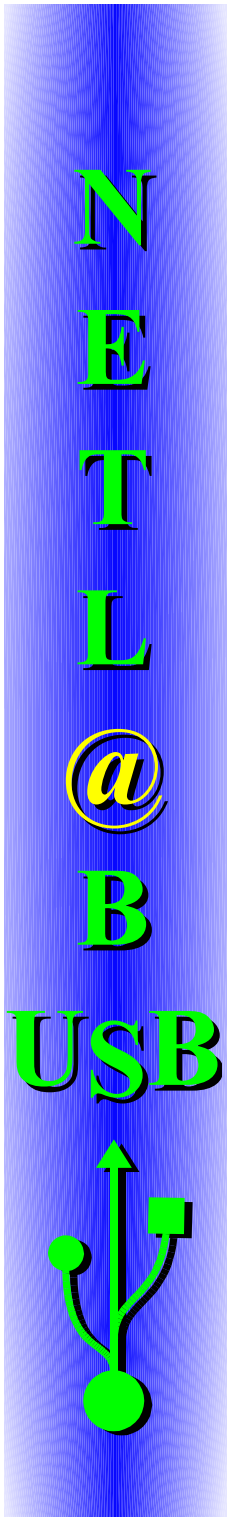

# GetLogValue

```
LONG GetLogValue( USHORT joyIndex, USHORT ItemOfs)
{
  LONG rc = 0;
  USHORT usOffset, usByteOfs, StartBit, usSize;
  BYTE     *pIntData, bRem;
  usOffset = gJOY[joyIndex].Items[ItemOfs].usOffset;
  usSize   = gJOY[joyIndex].Items[ItemOfs].usReportSize;
  pIntData = (BYTE *)&gJOY[joyIndex].buffer;

  // No proper index or Value to long
  if( (FULL_WORD==usOffset) || (usSize>32) )
    return rc;

  StartBit  = usOffset %8;
  usByteOfs = usOffset /8;

  //Check if in bounds of report
  if(usByteOfs>=gJOY[joyIndex].ReportLength)
    return rc;

  if(usSize>1)
  {
    if(!StartBit){
      // probably the easiest
      while(usSize>=8){
        rc *= 256;
        rc += pIntData[usByteOfs++];
        usSize-=8;
      }
      if(usSize){
        rc *= (2*usSize);
        bRem = pIntData[usByteOfs];
        bRem >>=(8-usSize);
        rc += bRem;
      }
    }
...
```




# GetLogValue cont.

```
...
    else{
      if( (StartBit-usSize)<=0){
        // All bits are in this byte
        bRem = pIntData[usByteOfs];
        bRem &= gRightMask[StartBit];
        bRem >>= (8-usSize-StartBit);
        rc = bRem;
      }
      else
      {
        bRem = pIntData[usByteOfs++];
        bRem &=gRightMask[StartBit];
        rc = bRem;
        usSize -= (8-StartBit);
        while(usSize>=8){
          rc *= 256;
          rc += pIntData[usByteOfs++];
          usSize-=8;
        }
        if(usSize){
          rc *= (2*usSize);
          bRem = pIntData[usByteOfs];
          bRem >>=(8-usSize);
          rc += bRem;
        }
      }
    }
  }
  else
  {    // 1 Byte only
    bRem = pIntData[usByteOfs] & gBitMask[StartBit];
    bRem >>= (7-StartBit);
    rc = bRem;
  }
  return rc;
}
```




# Useful information links

General info docs etc www.usb.org

USB device information www.linux-usb.org

Sources for many linux USB drivers
www.sourceforge.net

The OS/2 DDK with sources of USB drivers
service.boulder.ibm.com/ddk/