OS/2 USB Stack development Guidelines

Vladimirs Zinovjevs
(Vladimirs_Zinovjevs@exigengroup.lv)

Summary

- OS/2 & USB stack
- Development environment
- USB stack architecture
- Interrupt processing
- Device reservation
- USB filter driver design
- New features in usbmsd driver
- Relations between usbmsd and dasd (os2&dani)
- Known problems & restrictions

OS/2 & USB Stack

- Started development in 1997
- Limited driver support for USB 1.0/1.1
- Added USB 2.0 support in 2002
- Support of several class drivers:
  - HID devices (mice/keyboard)
  - Audio
  - modem/serial convertors
  - Ethernet driver
  - Mass Storage devices
  - Printers

Development Environment

- IBM DDK build tree
- Tools
  - MS C 6.0
  - Masm
- Built-in debug/service tools
  - Serial port printout routines (impacts timing), may control output message level
  - parameter/message processing routines
  - C library routine replacements
  - USB data structure processing routines
- Driver template

Development Environment

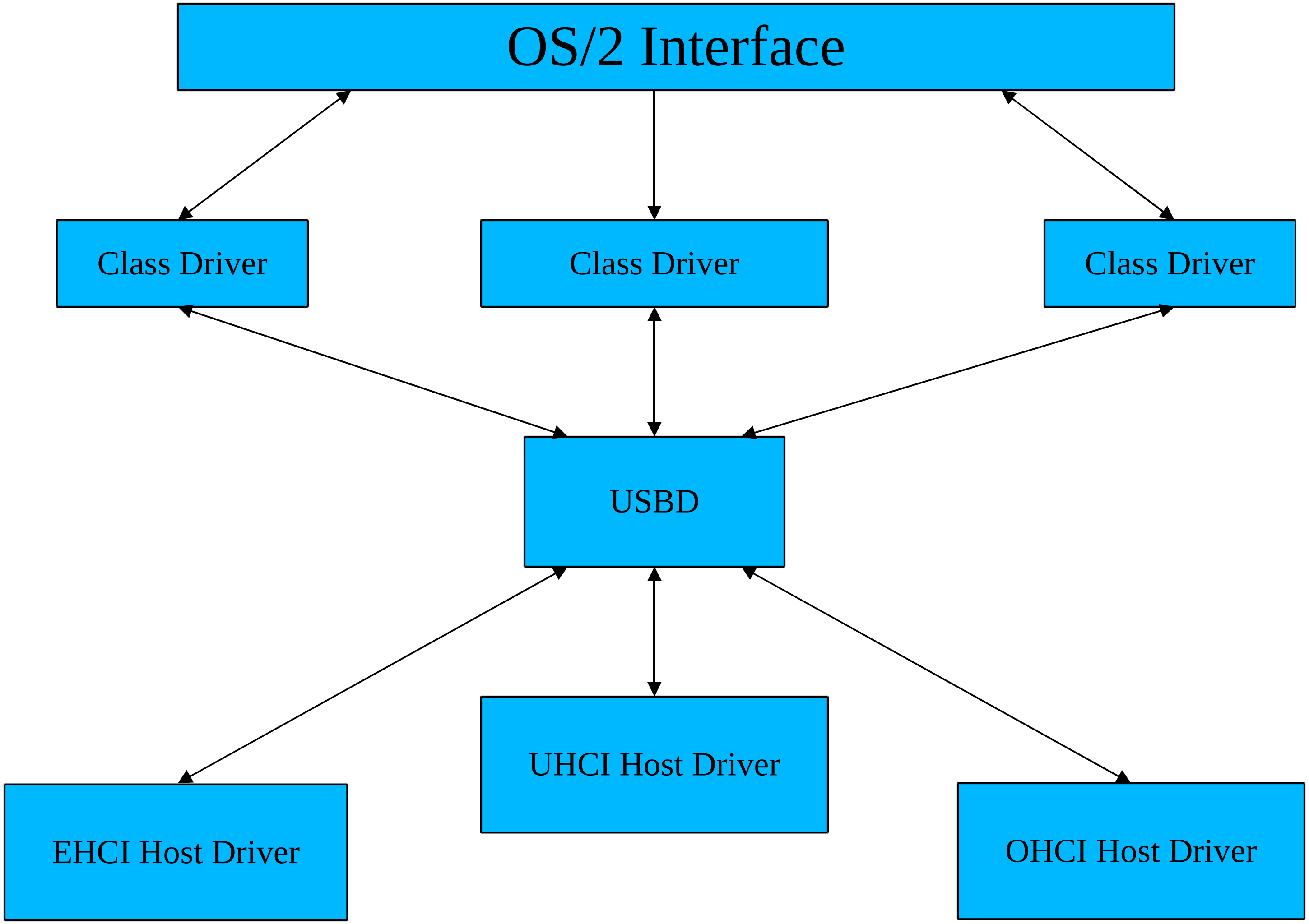
- Template files

– TM_const.c	constant definitons (names)
– TM_data.c	data structures, initializations
– TM_idc.c	IDC processor related routines
– TM_init.c	initialization time routines
– TM_irq.c	IRQ processing routines
– TM_segs.asm	driver's header, segments
– TM_strat.c	strategy router
– TM_.h	master include file
– TM_extrn.h	data structure external definitions
– TM_proto.h	function prototypes
– TM_types	driver's type defenition

- Makefile
- Template can be easily build by commands

```
drive:\ddk\tools\nmake /a DEBUG=1
drive:\ddk\tools\nmake /a
```

USB Stack Architecture



USB Stack Architecture - Interfaces

- External interfaces
  - Mass storage device adapter driver IORB interface
  - Multimedia interface
  - NDIS 2.04 interface
  - Serial/parallel interfaces
  - mice/keyboard USB/regular device IDC interface
- Internal (interstack) interface
  - IDC based
  - Similar to IOCTL interface
  - asynchronuous/synchronuous requests

USB Stack Architecture - Interfaces

Function	Type	Source
REGISTER	sync	class
SETCONF	async	class
SETINTF	async	class
PRCIRQ	sync	host
ACCIO	async	class
CANCEL	sync	class
CLRSTALL	async	class
CMPL_INI	sync	class
APM	sync	USBD
RESET_PORT	sync	class
IDLE	sync	USBD
CANCEL_STATE	sync	class

USB Stack Architecture - Interfaces

- CMPL_INI
    - sent to USBD to start host initialization when adapter driver has received notification from kernel that system is ready to switch from BIOS support to native drivers
- IDLE
    - Sent once after initial device enumeration has been completed

USB Stack Architecture - Interfaces

- RESET_PORT

Last resort to make port working, device address may change as enumeration will be executed again

```
void ResetPort(DeviceList *const pDevice)
{
  USBCancel      cancelRequest; // USB Cancel Request Block
  RP_GENIOCTL    rp_USBReq;     // USBD Request Packet

  #ifdef DEBUG
  if (!pDevice)
    dsPrint(DBG_CRITICAL, "MSD: reset port !pDevice\r\n");
  #endif
  //Check if device is connected
  if (!pDevice->pDeviceInfo) {
    pDevice->errorCode = IOERR_UNIT_NOT_READY;
    return;
  }
  cancelRequest.controllerId = pDevice->pDeviceInfo->ctrlID;
  cancelRequest.deviceAddress = pDevice->pDeviceInfo->deviceAddress;
  cancelRequest.endPointId = 0;
  #ifdef DEBUG
  dsPrint2(DBG_CRITICAL, "MSD: reset port %x %x\r\n",
          cancelRequest.controllerId, cancelRequest.deviceAddress);
  #endif
  setmem((PSZ)&rp_USBReq, 0, sizeof(rp_USBReq));
  rp_USBReq.rph.Cmd = CMDGenIOCTL;
  rp_USBReq.Category = USB_IDC_CATEGORY_USBD;
  rp_USBReq.Function = USB_IDC_FUNCTION_RESET_PORT;
  rp_USBReq.ParmPacket = (PVOID)&cancelRequest;
  USBCallIDC(gpUSBDIDC, gdsUSBIDC, (PRP_GENIOCTL)&rp_USBReq);
}
```

USB Stack Architecture - Interfaces

- CANCEL_STATE

In addition to regular cancel request returns endpoint /request state.

```
void CancelRequestWithState(USHORT prtIndex, USHORT endPoint)  {
  USBCancel   rb;   // USB Cancel Request Block
  RP_GENIOCTL rp;   // IOCtl Request Packet to USBD
  if (gPRT[prtIndex].pDeviceInfo) {
    rb.controllerId  = gPRT[prtIndex].pDeviceInfo->ctrlID;
    rb.deviceAddress = gPRT[prtIndex].pDeviceInfo->deviceAddress;
    rb.endPointId    = (UCHAR)endPoint;
    setmem((PSZ)&rp, 0, sizeof(rp));
    rp.rph.Cmd    = CMDGenIOCTL;
    rp.Category   = USB_IDC_CATEGORY_USBD;
    rp.Function   = USB_IDC_FUNCTION_CANCEL_STATE;
    rp.ParmPacket = (PVOID)&rb;
    USBCallIDC(gpUSBDIDC, gdsUSBDIDC, (PRP_GENIOCTL)&rp);
    if (rp.rph.Status == USB_IDC_RC_WRONGFUNC)
      CancelRequests(prtIndex, endPoint);
  }
}
```

USB Stack Architecture - Interfaces

```
do {
      if (!(gPRT[prtIndex].wFlags & STOP_TRANSMIT)) WriteData (prtIndex);
      do {
        awakeC = DevHelp_ProcBlock((ULONG)(PUCHAR)gPRT[prtIndex].pRPWrite[CURRENT],
                             (pRP->Unit)? // COM# : $USBPRT     in milliseconds
                             (ULONG)((gDCB[pRP->Unit-1].dcb.usWriteTimeout + 1)*10) :
                             gPRT[prtIndex].dwTO[WRITE_IDLE_TO],  WAIT_IS_INTERRUPTABLE);
      } while (awakeC != WAIT_TIMED_OUT && gPRT[prtIndex].pRPWrite[CURRENT]->rph.Status == 0);
      if (awakeC == WAIT_TIMED_OUT) {
        CancelRequestWithState(prtIndex, gPRT[prtIndex].writeEndpoint);
        DevHelp_ProcBlock((ULONG)(PUCHAR)gPRT[prtIndex].pRPWrite[CURRENT], 1000, WAIT_IS_INTERRUPTABLE);
        if ((pRP->Unit == 0 && gPRT[prtIndex].bInfinRetry == TRUE)      ||
           (pRP->Unit >  0 && gDCB[pRP->Unit-1].dcb.fbTimeout & F3_W_INF_TO))
        { // to try to write the data to the USB printer
          continue;
        } else {
          gPRT[prtIndex].pRPWrite[CURRENT]->rph.Status |= STERR | ERROR_I24_WRITE_FAULT;  break;
        }
      } else if (gPRT[prtIndex].wFlags & (FLUSH_OUT_INPROGRESS | WRITE_DATA_ERROR)) {
        gPRT[prtIndex].pRPWrite[CURRENT]->rph.Status &= ~STBUI; break;
      }  else gPRT[prtIndex].pRPWrite[CURRENT]->rph.Status = 0;
    } while (gPRT[prtIndex].wWCount < gPRT[prtIndex].wWReqCount);
```

USB Stack Architecture - Interfaces

- APM

    Power management notification

```
     switch ( pRP_GENIOCTL->Category )
    {
    case USB_IDC_CATEGORY_CLASS:
      switch ( pRP_GENIOCTL->Function )  {
      case USB_IDC_FUNCTION_APM:
        APMService (pRP_GENIOCTL);
        break; //LR0619end
      }
      break;
    }

static void APMService (PRP_GENIOCTL pRP) {
  ULONG       apmState = ((USBAPMNotification FAR *)pRP->ParmPacket)->apmState;
  if (apmState == USB_APM_SUSPEND) {
  } else if (apmState == USB_APM_RESUME) {
  }
}
```

USB Stack Architecture - Tricks

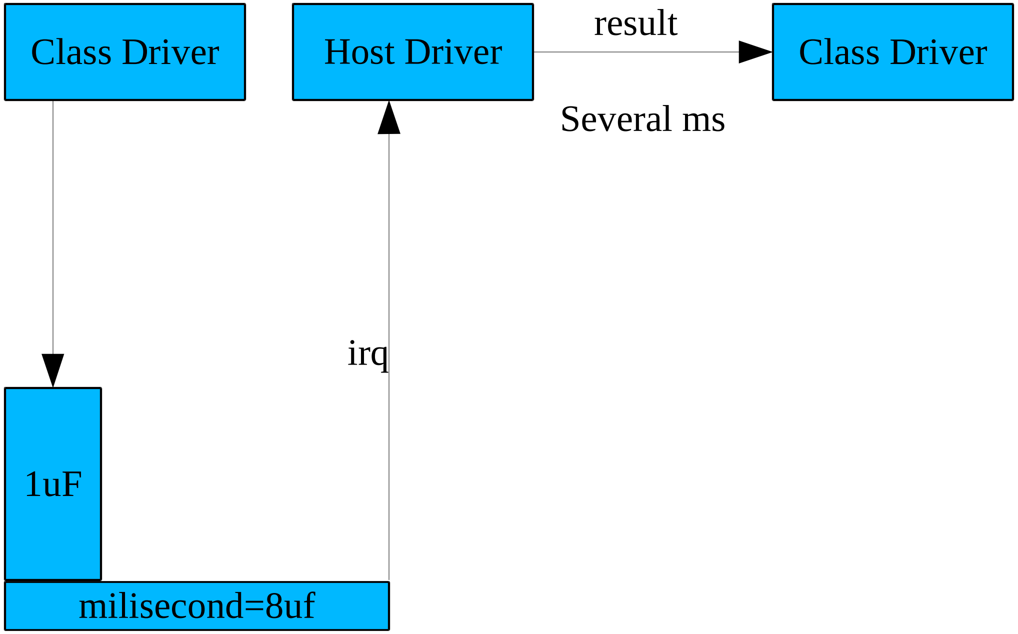
```
void CancelRequests (USHORT prtIndex, USHORT endPoint) {
   USBCancel   rb;   // USB Cancel Request Block
   RP_GENIOCTL rp;   // IOCtl Request Packet to USBD
   if (gPRT[prtIndex].pDeviceInfo)  {
      rb.controllerId = gPRT[prtIndex].pDeviceInfo->ctrlID;
      rb.deviceAddress = gPRT[prtIndex].pDeviceInfo->deviceAddress;
      rb.endPointId = (UCHAR)endPoint;
      setmem((PSZ)&rp, 0, sizeof(rp));
      rp.rph.Cmd = CMDGenIOCTL;
      rp.Category = USB_IDC_CATEGORY_USBD;
      rp.Function = USB_IDC_FUNCTION_CANCEL;
      rp.ParmPacket = (PVOID)&rb;
      USBCallIDC (gpUSBDIDC, gdsUSBDIDC, (PRP_GENIOCTL)&rp);
   }
}
```

USB Stack Architecture - Tricks

```
#define  MAX_BULK_HS_BUFFSIZE      65535
#define  MAX_BULK_BUFFSIZE         16384
          if (pCurrDevice->pDeviceInfo->SpeedDevice == USB_HIGH_DEV_SPEED) {
            // for USB20 driver can send/receive 3*20kb bytes simultaneously
            if (currBuffLen > MAX_BULK_HS_BUFFSIZE)  *length = 61440; // it 3 transfer descriptors
            else {
              if (scatGatIndex != pCurrDevice->cSGList - 1) {
                // adjust data length to be equal maxPacketSize*n, n = 1,2,..
                // otherwise USB device will stall request
                *length = (USHORT)(currBuffLen - (currBuffLen % HS_MAX_PACKET_BULK_SIZE));
              } else // driver sends all data if it is a last item in a gather list
                *length = (USHORT)currBuffLen;
            }
          } else {   // for USB11 driver can send/receive 16kb bytes simultaneously
            if (currBuffLen >= (ULONG)gBuffSize)   *length = gBuffSize;
            else {
              if (scatGatIndex != pCurrDevice->cSGList - 1) {
                // adjust data length to be equal maxPacketSize*n, n = 1,2,..
                // otherwise USB device will stall request
                *length = (USHORT)(currBuffLen - (currBuffLen % FS_MAX_PACKET_BULK_SIZE));
              } else // driver sends all data if it is a last item in a gather list
                *length = (USHORT)currBuffLen;
            }
          }
```

USB Stack Architecture - Tricks

- You shoud merge buffers from scatter gather list into one buffer and send it to USBD.



Interrupt Processing

- Limited processing at IRQ time in host drivers
- Finalizing during task time, calls initiating driver directly
- Original request structure may not match 1-1 to one returned during IRQ (hostID/address/endpoint/requestdata fields are always restored)
- Transfer status are reflected in request's status field and buffer length fields
- New requests are/may be initiated during IRQ notification calls
- Class drivers (also other ones) should not sent any requests to USBD driver during interrupt time (like from timer callback routines)

Device Reservation

- Based on configuration selection
  - Must be set as soon as possible during device attach notification process
  - Configuration must be set via SETCONF call to USBD
- device/interface sharing only between friendly drivers

USB Filter Driver

- Uses the same interfaces as regular driver
- Filter nature only when accepting device for service – may set filter driver IDC/DS addresses as per device basis
- May send requests to USBD driver using special command (CMDIOCTLW instead of CMDGenIOCTL) to bypass filtering
- After registration filter is called instead of host driver for each request, except for REGISTER /PRCIRQ / CMPL_INI
  - May update commands/data to be sent to device
  - May replace one request with one or more other requests to implement support for non-standard devices

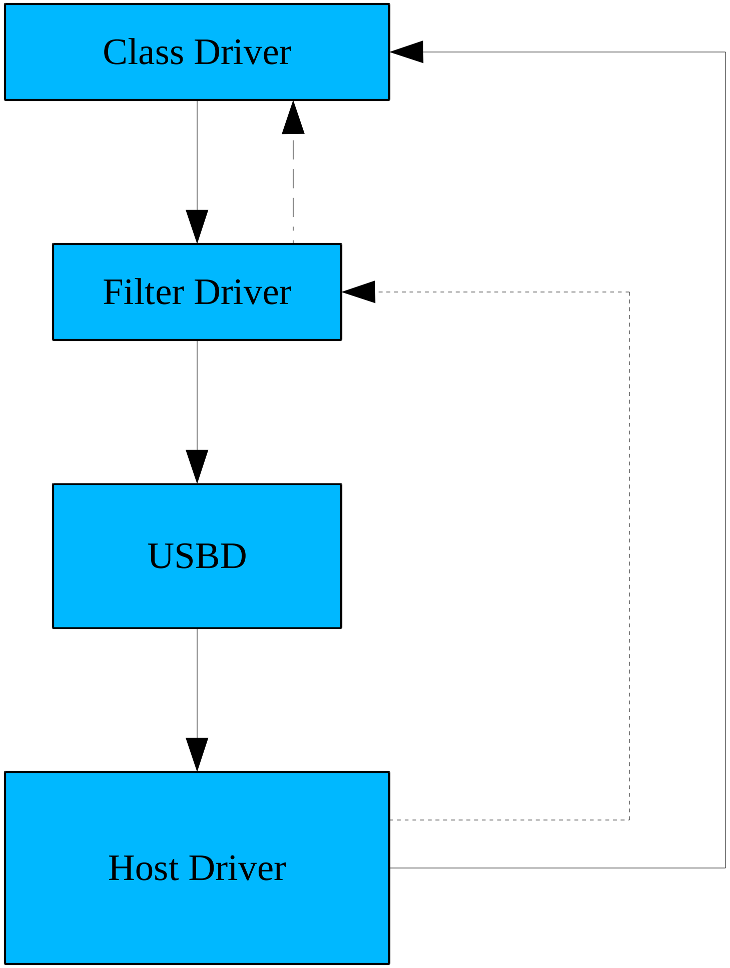
USB Filter Driver

- May update IRQ processing IDC/DS address

```
pRB->usbDS = GetDS();
pRB->usbIDC = (PUSBIDCEntry)&FL_idc;
```

- Post request data processing during IRQ
- Possible timeout problems when replacing single request with several for devices served by drivers that support time-outs for requests (like MSD driver)

USB Filter Driver



Copyright Exigen

USB MSD Driver

- Initial driver supports only first Logical Unit Number 0
- Added multiple LUN support in 2004
- Fixed several problems with device geometry detection (for BOT devices and for UFI devices):
  - fixed CBI-NI protocol support
  - fixed format for UFI
  - ignored incorrect CHS geometry
- Added USB HDD support. The key FIXED_DISKS is ignored now.
- Added possibility to work with USB CDRW devices. A filter driver must be implemented.
- ModeSense10 command can be avoided. (/MS10_OFF)
- Supports non-512 bytes/sector media with Dani filter.

USB MSD & DASD Drivers

The following dasd drivers exists:

- Os2dasd for MCP and ACP
- Os2dasd for Warp4
- Danidasd

At present moment the latest fixes for USB mass storage devices have been inserted only in os2dasd for MCP:

- Eject command can be used for hard drives.
- Driver supports USB HDD with large media
- Can detect partitions created by non-OS2 OS.
- Can work with media formatted by another OS.
- Supports non-OS2 oem names for PRM.

There are the following restrictions in danidasd and os2dasd for Warp4:

- New key CHS can be used. MSD calculates CHS geometry from device geometry. This key helps to support USB drives with capacity more than 40GB.
- Eject command must be rewritten.

Problems

Host problems:

- Does not support physically discontinuous buffers.
- Does not support zero length transfers. It may be a critical point for some protocols.

USBD problems:

- Interrupt processing is incorrect if short packets are in use (more than one transfer descriptor).

Class drivers' problems: