

```

// DUET.SYS main program: Display Usb Enumeration Transactions, Version 1.0.
//
//
// Copyright (C) 2001, Intel Corporation
// All rights reserved.
// Permission is hereby granted to merge this program code with other program
// material to create a derivative work. This derivative work may be distributed
// in compiled object form only. Any other publication of this program, in any form,
// without the explicit permission of the copyright holder is prohibited.
//
// Derived from:
// Main program for filter driver
// Copyright (C) 1999 by Walter Oney
// All rights reserved. Used with permission
//
//
// DUET.SYS uses Walter Oney's filter structure and adds an "intercept" function on the USB system calls.
// The calls during enumeration are displayed, with their results, on the debug console.
// Using this software presumes that the PC Host can correctly read the first Device_Descriptor (prior to
// the Set_Address) since this filter is keyed off the VID, PID value.
//
// Send questions and comments to John.Hyde@intel.com

#include "stdcls.h"
#include "driver.h"
#include "usbdi.h"

VOID DriverUnload(IN PDRIVER_OBJECT fido);
NTSTATUS AddDevice(IN PDRIVER_OBJECT DriverObject, IN PDEVICE_OBJECT pdo);
NTSTATUS PassDownPower(IN PDEVICE_OBJECT fido, IN PIRP Irp);
NTSTATUS PassDownPnp(IN PDEVICE_OBJECT fido, IN PIRP Irp);
NTSTATUS PassDownWmi(IN PDEVICE_OBJECT fido, IN PIRP Irp);
NTSTATUS CheckTheseFunctions(IN PDEVICE_OBJECT fido, IN PIRP Irp);
NTSTATUS InterceptReply(IN PDEVICE_OBJECT fido, IN PIRP Irp, int Context);
VOID DecodeDescriptor (unsigned char* pByte, USHORT Type);

static char* urbname[] = {
    "SELECT_CONFIGURATION",
    "SELECT_INTERFACE",
    "ABORT_PIPE",
    "TAKE_FRAME_LENGTH_CONTROL",
    "RELEASE_FRAME_LENGTH_CONTROL",
    "GET_FRAME_LENGTH",
    "SET_FRAME_LENGTH",
    "GET_CURRENT_FRAME_NUMBER",
    "CONTROL_TRANSFER",
    "BULK_OR_INTERRUPT_TRANSFER",
    "ISOCH_TRANSFER",
    "GET_DESCRIPTOR_FROM_DEVICE",
    "SET_DESCRIPTOR_TO_DEVICE",
    "SET_FEATURE_TO_DEVICE",
    "SET_FEATURE_TO_INTERFACE",
    "SET_FEATURE_TO_ENDPOINT",
    "CLEAR_FEATURE_TO_DEVICE",
    "CLEAR_FEATURE_TO_INTERFACE",
    "CLEAR_FEATURE_TO_ENDPOINT",
    "GET_STATUS_FROM_DEVICE",
    "GET_STATUS_FROM_INTERFACE",
    "GET_STATUS_FROM_ENDPOINT",
    "RESERVED0",
    "VENDOR_DEVICE",
    "VENDOR_INTERFACE",
    "VENDOR_ENDPOINT",
    "CLASS_DEVICE",
    "CLASS_INTERFACE",
    "CLASS_ENDPOINT",
    "RESERVED1",
    "RESET_PIPE",
    "CLASS_OTHER",
    "VENDOR_OTHER",
    "GET_STATUS_FROM_OTHER",
    "CLEAR_FEATURE_TO_OTHER",
    "SET_FEATURE_TO_OTHER",
    "GET_DESCRIPTOR_FROM_ENDPOINT",
    "SET_DESCRIPTOR_TO_ENDPOINT",
    "GET_CONFIGURATION",
    "GET_INTERFACE",
    "GET_DESCRIPTOR_FROM_INTERFACE",
    "SET_DESCRIPTOR_TO_INTERFACE",

```

```

};

BOOLEAN IsWin98();
UNICODE_STRING servkey;

////////////////////////////////////////////////////////////////

#pragma INITCODE

extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath) {
    KdPrint((DRIVERNAME "Entering DriverEntry: DriverObject %8.8lX\n", DriverObject));

    // Insist that OS support at least the WDM level of the DDK we use
    if (!IoIsWdmVersionAvailable(1, 0)) {
        KdPrint((DRIVERNAME "Expected version of WDM (%d.%2.2d) not available\n", 1, 0));
        return STATUS_UNSUCCESSFUL; }

    // See if we're running under Win98 or NT:
    if (IsWin98()) KdPrint((DRIVERNAME "Running under Windows 98\n"));
    else KdPrint((DRIVERNAME "Running under NT\n"));

    // Save the name of the service key
    servkey.Buffer = (PWSTR) ExAllocatePool(PagedPool, RegistryPath->Length + sizeof(WCHAR));
    if (!servkey.Buffer) {
        KdPrint((DRIVERNAME " - Unable to allocate %d bytes for copy of service key name\n", RegistryPath->Length +
sizeof(WCHAR)));
        return STATUS_INSUFFICIENT_RESOURCES; }
    servkey.MaximumLength = RegistryPath->Length + sizeof(WCHAR);
    RtlCopyUnicodeString(&servkey, RegistryPath);

    // Initialize function pointers
    DriverObject->DriverUnload = DriverUnload;
    DriverObject->DriverExtension->AddDevice = AddDevice;
    for (int i = 0; i < arraysize(DriverObject->MajorFunction); ++i) DriverObject->MajorFunction[i] =
CheckTheseFunctions;
    DriverObject->MajorFunction[IRP_MJ_POWER] = PassDownPower;
    DriverObject->MajorFunction[IRP_MJ_PNP] = PassDownPnp;
    DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] = PassDownWmi;

    return STATUS_SUCCESS;
}

////////////////////////////////////////////////////////////////

BOOLEAN IsWin98() {
#ifdef _X86_
    return !IoIsWdmVersionAvailable(1, 0x10);
#else
    return FALSE;
#endif
}

////////////////////////////////////////////////////////////////

#pragma PAGEDCODE

VOID DriverUnload(IN PDRIVER_OBJECT DriverObject) {
    PAGED_CODE();
    KdPrint((DRIVERNAME "Entering DriverUnload: DriverObject %8.8lX\n", DriverObject));
    RtlFreeUnicodeString(&servkey);
}

////////////////////////////////////////////////////////////////

NTSTATUS AddDevice(IN PDRIVER_OBJECT DriverObject, IN PDEVICE_OBJECT pdo) {
    PAGED_CODE();
    KdPrint((DRIVERNAME "Entering AddDevice: DriverObject %8.8lX, pdo %8.8lX\n", DriverObject, pdo));
    // See Walter Oney's FILTER program for the magic in this routine

    PDEVICE_OBJECT fido;
    NTSTATUS status = IoCreateDevice(DriverObject, sizeof(DEVICE_EXTENSION), NULL, FILE_DEVICE_UNKNOWN, 0, FALSE,
&fido);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVERNAME "IoCreateDevice failed - %X\n", status));
        return status; }
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;

    __try {
        IoInitializeRemoveLock(&pdx->RemoveLock, 0, 0, 0);

```

```

pdx->DeviceObject = fido;
pdx->Pdo = pdo;
PDEVICE_OBJECT fdo = IoAttachDeviceToDeviceStack(fido, pdo);
if (!fdo) {
    KdPrint((DRIVERNAME "IoAttachDeviceToDeviceStack failed\n"));
    status = STATUS_DEVICE_REMOVED;
    __leave; }
pdx->LowerDeviceObject = fdo;
fido->Flags |= fdo->Flags & (DO_DIRECT_IO | DO_BUFFERED_IO | DO_POWER_PAGABLE | DO_POWER_INRUSH);
fido->DeviceType = fdo->DeviceType;
fido->Characteristics = fdo->Characteristics;
fido->Flags &= ~DO_DEVICE_INITIALIZING; }
__finally { // cleanup side effects
    if (!NT_SUCCESS(status)) {
        if (pdx->LowerDeviceObject) IoDetachDevice(pdx->LowerDeviceObject);
        IoDeleteDevice(fido); }
}
return status;
}

```

//

```

VOID RemoveDevice(IN PDEVICE_OBJECT fido) {
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    if (pdx->LowerDeviceObject) IoDetachDevice(pdx->LowerDeviceObject);
    IoDeleteDevice(fido);
}

```

//

```

#pragma LOCKEDCODE
extern "C" void __declspec(naked) __cdecl _chkesp() {
    _asm je okay
        ASSERT(!DRIVERNAME "Stack pointer mismatch");
okay:
    _asm ret
}

```

//

#pragma LOCKEDCODE

```

NTSTATUS CompleteRequest(IN PIRP Irp, IN NTSTATUS status, IN PULONG info) {
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = ULONG(info);
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

```

//

```

NTSTATUS PassDownPower(IN PDEVICE_OBJECT fido, IN PIRP Irp) {
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    PoStartNextPowerIrp(Irp); // must be done while we own the IRP
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
    IoSkipCurrentIrpStackLocation(Irp);
    status = PoCallDriver(pdx->LowerDeviceObject, Irp);
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return status;
}

```

//

```

NTSTATUS PassDownPnp(IN PDEVICE_OBJECT fido, IN PIRP Irp) {
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    ULONG fcn = stack->MinorFunction;
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
    IoSkipCurrentIrpStackLocation(Irp);
    status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    if (fcn == IRP_MN_REMOVE_DEVICE) {
        IoReleaseRemoveLockAndWait(&pdx->RemoveLock, Irp);
        RemoveDevice(fido); }
    else IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return status;
}

```

```
////////////////////////////////////////////////////////////////
```

```
NTSTATUS PassDownWmi(IN PDEVICE_OBJECT fido, IN PIRP Irp) {
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
    IoSkipCurrentIrpStackLocation(Irp);
    status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return status;
}
```

```
////////////////////////////////////////////////////////////////
```

```
NTSTATUS CheckTheseFunctions(IN PDEVICE_OBJECT fido, IN PIRP Irp) {
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
    ULONG IoControlCode = stack->Parameters.DeviceIoControl.IoControlCode;
    bool Intercept = false; char Type = 0; USHORT USBfunction = 0; unsigned char Configuration;
    if (IoControlCode == IOCTL_INTERNAL_USB_SUBMIT_URB) {
        PURB urb = (PURB)stack->Parameters.Others.Argument1;
        USBfunction = urb->UrbHeader.Function;
        switch (USBfunction) {
            case URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE:
                Type = urb->UrbControlDescriptorRequest.DescriptorType;
                if (Type == 1) KdPrint((DRIVERNAME "Request: Get Device Descriptor\n"));
                if (Type == 2) KdPrint((DRIVERNAME "Request: Get Configuration Descriptor\n"));
                Intercept = true; break;
            case URB_FUNCTION_SELECT_CONFIGURATION:
                Configuration = urb->UrbSelectConfiguration.ConfigurationDescriptor->bConfigurationValue;
                KdPrint((DRIVERNAME "Command: Set Configuration to %2.2x\n", Configuration)); break;
            case URB_FUNCTION_GET_DESCRIPTOR_FROM_ENDPOINT:
                KdPrint((DRIVERNAME "Request: Get Report Descriptor\n")); Intercept = true; break;
            case URB_FUNCTION_CLASS_ENDPOINT:
                if (urb->UrbControlVendorClassRequest.Request == 10)
                    KdPrint((DRIVERNAME "Command: SET_IDLE = %4.4x\n", urb->UrbControlVendorClassRequest.Value)); break;
        }
    }
    if (Intercept) {
        // Need to intercept the reply so that the results can be displayed
        // KdPrint((DRIVERNAME "Intercepting %s\n", urbname[USBfunction]));
        int Context = USBfunction | (Type << 16); // InterceptReply needs the original Request
        IoCopyCurrentIrpStackLocationToNext(Irp);
        IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE)InterceptReply, (VOID*)Context, TRUE, TRUE, TRUE);
        return IoCallDriver(pdx->LowerDeviceObject, Irp);
    }
    else {
        // Pass request down without additional processing
        // KdPrint((DRIVERNAME "Passing down %s\n", urbname[USBfunction]));
        IoSkipCurrentIrpStackLocation(Irp);
        status = IoCallDriver(pdx->LowerDeviceObject, Irp);
        IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
        return status;
    }
}
```

```
////////////////////////////////////////////////////////////////
```

```
NTSTATUS InterceptReply(IN PDEVICE_OBJECT fido, IN PIRP Irp, int Context) {
    // The urb function gets changed to CONTROL_TRANSFER so recover the USBfunction and Type from original Request
    USHORT USBfunction = Context & 0x0fff;
    USHORT Type = Context >> 16;
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    if (Irp->PendingReturned) IoMarkIrpPending(Irp);
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    PURB urb = (PURB)stack->Parameters.Others.Argument1;
    unsigned char* pByte = (unsigned char*)urb->UrbControlDescriptorRequest.TransferBuffer;
    unsigned long Length = urb->UrbControlDescriptorRequest.TransferBufferLength;

    KdPrint((DRIVERNAME "Response: Raw Data is %x (%dd) bytes = ", Length, Length));
    for (int i = 0; i < (int)Length; i++) {if ((i & 15) == 0) KdPrint((" \n")); KdPrint((" %2.2x", *pByte++);}
    KdPrint((" \n"));
    if (USBfunction == URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE) {
        pByte = (unsigned char*)urb->UrbControlDescriptorRequest.TransferBuffer; // Reset data pointer
        DecodeDescriptor(pByte, Type);
    }
    // KdPrint((DRIVERNAME "Intercept of %s completed\n", urbname[USBfunction]));
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return STATUS_SUCCESS;
}
```

```
}
```

```
VOID DecodeDescriptor (unsigned char* pByte, USHORT Type) {
    unsigned char LowByte;
    unsigned char* pEnd = 0;
    unsigned char* pLengthLow = pByte + 2;
    unsigned char* pLengthHi = pByte + 3;
    if (Type == 1) pEnd = pByte + 18; // Device Descriptor
    if (Type == 2) pEnd = pByte + *pLengthLow + (*pLengthHi << 8); // Configuration Descriptor
    bool Invalid = false;
    do {
        switch (Type){
        case 1:
            KdPrint((DRIVERNAME "Response: Decoded Device Descriptor\n"));
            KdPrint((DRIVERNAME " bLength = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bDescriptorType = %2.2x\n", *pByte++));
            LowByte = *pByte++;
            KdPrint((DRIVERNAME " bcdUSB = %4.4x\n", LowByte+(*pByte++ << 8) ));
            KdPrint((DRIVERNAME " bDeviceClass = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bDeviceSubClass = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bDeviceProtocol = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bMaxEP0Size = %2.2x\n", *pByte++));
            LowByte = *pByte++;
            KdPrint((DRIVERNAME " VendorID = %4.4x\n", LowByte+(*pByte++ << 8) ));
            LowByte = *pByte++;
            KdPrint((DRIVERNAME " ProductID = %4.4x\n", LowByte+(*pByte++ << 8) ));
            LowByte = *pByte++;
            KdPrint((DRIVERNAME " bcdDevice = %4.4x\n", LowByte+(*pByte++ << 8) ));
            KdPrint((DRIVERNAME " iManufacturer = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " iProduct = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " iSerialNumber = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bNumConfigurations = %2.2x\n", *pByte++));break;
        case 2:
            KdPrint((DRIVERNAME "Response: Decoded Configuration Descriptor\n"));
            KdPrint((DRIVERNAME " bLength = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bDescriptorType = %2.2x\n", *pByte++));
            LowByte = *pByte++;
            KdPrint((DRIVERNAME " wTotalLength = %4.4x\n", LowByte+(*pByte++ << 8) ));
            KdPrint((DRIVERNAME " bNumInterfaces = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bConfigValue = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " iConfiguration = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bmAttributes = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " MaxPower = %2.2x\n", *pByte++)); break;
        case 3: // Will not get strings in an enumeration sequence
            KdPrint((DRIVERNAME "Response: Decoded Get String Descriptor\n")); Invalid = true; break;
        case 4:
            KdPrint((DRIVERNAME "Response: Decoded Interface Descriptor\n"));
            KdPrint((DRIVERNAME " bLength = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bDescriptorType = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bInterfaceNum = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bAlternateSetting = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bNumEndpoints = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bInterfaceClass = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bDeviceSubClass = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bDeviceProtocol = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " iInterface = %2.2x\n", *pByte++));break;
        case 5:
            KdPrint((DRIVERNAME "Response: Decoded Endpoint Descriptor\n"));
            KdPrint((DRIVERNAME " bLength = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bDescriptorType = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bEndpointAddress = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bmAttributes = %2.2x\n", *pByte++));
            LowByte = *pByte++;
            KdPrint((DRIVERNAME " wMaxpacketSize = %4.4x\n", LowByte+(*pByte++ << 8) ));
            KdPrint((DRIVERNAME " bInterval = %2.2x\n", *pByte++)); break;
        case 33:
            KdPrint((DRIVERNAME "Response: Decoded HID Descriptor\n"));
            KdPrint((DRIVERNAME " bLength = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " bDescriptorType = %2.2x\n", *pByte++));
            LowByte = *pByte++;
            KdPrint((DRIVERNAME " HIDversion# = %4.4x\n", LowByte+(*pByte++ << 8) ));
            KdPrint((DRIVERNAME " CountryCode = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " HIDDescriptorCount = %2.2x\n", *pByte++));
            KdPrint((DRIVERNAME " HIDReportType = %2.2x\n", *pByte++));
            LowByte = *pByte++;
            KdPrint((DRIVERNAME " HIDReportLength = %4.4x\n", LowByte+(*pByte++ << 8) )); break;
        default: Invalid = true;
        }
    }
    pByte++; Type = *pByte--; // Look at the next descriptor if any
}
```

```
    if (Invalid) {KdPrint((DRIVERNAME "Invalid descriptor type %2.2 found\n", Type)); break;}  
  } while (pByte < pEnd);  
}
```