

```

// Main program for LoadEz function driver: EZ-USB auto-loader program, Version 2.0
//
// 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 holders is prohibited.
//
// Send questions and comments to John.Hyde@intel.com
//
// Derived using Walter Oney's WDM Wizard
// Copyright (C) 1999, 2000 by Walter Oney
// All rights reserved. Used with permission
//
// Also uses Walter Oney's Portable FileIO Subsystem
// Copyright (C) 1999, 2000 by Walter Oney
// All rights reserved. Used with permission
//

#include "stdcls.h"
#include "driver.h"
#include <initguid.h>
#include "guids.h"
#include "usbdi.h"
#include "fileIO.h"

NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo);
VOID DriverUnload(PDRIVER_OBJECT fdo);

struct INIT_STRUCT : public _GENERIC_INIT_STRUCT {};

// Need a few global variables
const WCHAR HexCharacter[] = L"0123456789ABCDEF";
BOOLEAN win98 = FALSE;

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

#pragma INITCODE

extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    KdPrint((DRIVERNAME "Version 2.1. (John.Hyde@intel.com)\n"));

    // We require GENERIC.SYS 1.3 or later.
    if (GenericGetVersion() < 0x00010003) {
        KdPrint((DRIVERNAME "Required version (=>1.3) of GENERIC.SYS not installed\n"));
        return STATUS_UNSUCCESSFUL;
    }

    // See if we're running under Win98 or 2000:
    win98 = IsWin98();
    KdPrint((DRIVERNAME "Running under Windows(R) ");
    if (win98) KdPrint(("98\n")); else KdPrint(("2000\n"));

    // Initialize function pointers. LoadEz only needs four
    DriverObject->DriverUnload = DriverUnload;
    DriverObject->DriverExtension->AddDevice = AddDevice;
    DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
    DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;

    return STATUS_SUCCESS;
}

#pragma PAGEDCODE

VOID DriverUnload(PDRIVER_OBJECT DriverObject) {
    PAGED_CODE();
    KdPrint((DRIVERNAME "Unloading Driver\n"));
}

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

NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo) {
    PAGED_CODE();
    KdPrint((DRIVERNAME "Adding Device\n"));
    NTSTATUS status;

    // Create a functional device object to represent the hardware we're managing.

```

```

PDEVICE_OBJECT fdo;
ULONG dxsize = (sizeof(DEVICE_EXTENSION) + 7) & ~7;
ULONG xsize = dxsize + GetSizeofGenericExtension();
status = IoCreateDevice(DriverObject, xsize, NULL, FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE, &fdo);
if (!NT_SUCCESS(status)) {
    KdPrint((DRIVERNAME "IoCreateDevice failed with status %X\n", status));
    return status;
}
KdPrint((DRIVERNAME "Device Object (%8.8x) created\n", fdo));

PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
BOOLEAN ginit = FALSE;

// From this point forward, any error will have side effects that need to be cleaned up.
// Using a try-finally block allows easy modification without losing track of the side effects.
__try
{
    // finish initialization
    pdx->DeviceObject = fdo;
    pdx->Pdo = pdo;

// Link our device object into the stack leading to the PDO
    pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);
    if (!pdx->LowerDeviceObject) {
        KdPrint((DRIVERNAME "IoAttachDeviceToDeviceStack failed\n"));
        status = STATUS_DEVICE_REMOVED;
        __leave;
    }
    // can't attach device

// Set power management flags in the device object
    fdo->Flags |= DO_POWER_PAGABLE;

// Initialize to use the GENERIC.SYS library
    pdx->pgx = (PGENERIC_EXTENSION) ((PUCHAR) pdx + dxsize);
    INIT_STRUCT gis;
    RtlZeroMemory(&gis, sizeof(gis));
    gis.Size = sizeof(gis);
    gis.DeviceObject = fdo;
    gis.Pdo = pdo;
    gis.Ldo = pdx->LowerDeviceObject;
    gis.RemoveLock = &pdx->RemoveLock;
    gis.StartDevice = StartDevice;
    gis.StopDevice = StopDevice;
    gis.RemoveDevice = RemoveDevice;
    RtlInitUnicodeString(&gis.DebugName, LDRIVERNAME);

    status = InitializeGenericExtension(pdx->pgx, &gis);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVERNAME "InitializeGenericExtension failed with status %X\n", status));
        __leave;
    }
    ginit = TRUE;
    GenericRegisterInterface(pdx->pgx, &GUID_INTERFACE_LOADEZ);

// Clear the "initializing" flag so that we can get IRPs
    fdo->Flags &= ~DO_DEVICE_INITIALIZING;
}
__finally
{
    // cleanup side effects
    if (!NT_SUCCESS(status)) { // need to cleanup
        if (ginit) CleanupGenericExtension(pdx->pgx);
        if (pdx->LowerDeviceObject) IoDetachDevice(pdx->LowerDeviceObject);
        IoDeleteDevice(fdo);
    }
}
return status;
}

VOID RemoveDevice(IN PDEVICE_OBJECT fdo) {
    PAGED_CODE();
    KdPrint((DRIVERNAME "Removing Device (%8.8x)\n", fdo));
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS status;
    if (pdx->LowerDeviceObject) IoDetachDevice(pdx->LowerDeviceObject);
    if (pdx->HexFileBuffer) ExFreePool(pdx->HexFileBuffer);
    IoDeleteDevice(fdo);
}

////////////////////////////////////
// Plug-and-Play and Power Management is handled, for the most part, by GENERIC.SYS

```

```

NTSTATUS DispatchPnp(PDEVICE_OBJECT fdo, PIRP Irp) {
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    return GenericDispatchPnp(pdx->pgx, Irp);
}

NTSTATUS DispatchPower(PDEVICE_OBJECT fdo, PIRP Irp) {
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    return GenericDispatchPower(pdx->pgx, Irp);
}

////////////////////////////////////
// Declare the support functions

NTSTATUS SendUrbAndWait(PDEVICE_OBJECT fdo, PURB Urb) {
    PAGED_CODE();
    KEVENT Event;
    IO_STATUS_BLOCK ioStatus;
    NTSTATUS Status = STATUS_SUCCESS;
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    KeInitializeEvent(&Event, NotificationEvent, FALSE);
    PIRP Irp = IoBuildDeviceIoControlRequest(IOCTL_INTERNAL_USB_SUBMIT_URB,
        pdx->LowerDeviceObject, NULL, 0, NULL, 0, TRUE, &Event, &ioStatus);
    PIO_STACK_LOCATION Stack = IoGetNextIrpStackLocation(Irp);
    Stack->Parameters.Others.Argument1 = (PVOID) Urb;
    Status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    if (Status == STATUS_PENDING) {
        KeWaitForSingleObject(&Event, Executive, KernelMode, FALSE, NULL);
        Status = ioStatus.Status;
    }
    return Status;
}

NTSTATUS GetIdentity(PDEVICE_OBJECT fdo) {
    PAGED_CODE();
    // Need to discover the VID and PID currently in use
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    USB_DEVICE_DESCRIPTOR Descriptor;
    NTSTATUS Status;
    URB LocalUrb;
    PURB Urb = &LocalUrb;
    RtlZeroMemory(Urb, sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST));
    Urb->UrbHeader.Length = sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST);
    Urb->UrbHeader.Function = URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE;
    Urb->UrbControlDescriptorRequest.TransferBufferLength = sizeof(Descriptor);
    Urb->UrbControlDescriptorRequest.TransferBuffer = &Descriptor;
    Urb->UrbControlDescriptorRequest.TransferBufferMDL = NULL;
    Urb->UrbControlDescriptorRequest.DescriptorType = 1;
    Status = SendUrbAndWait(fdo, Urb);
    pdx->VID = Descriptor.idVendor;
    pdx->PID = Descriptor.idProduct;
    KdPrint((DRIVERNAME "Current VID = %4.4x and PID = %4.4x\n", pdx->VID, pdx->PID));
    return Status;
}

NTSTATUS SetEZUSBMemory(PDEVICE_OBJECT fdo, USHORT StartLocation, USHORT Length, PVOID DataBytes) {
    PAGED_CODE();
    // Need to build a Vendor Request and send it to the development board
    URB LocalUrb;
    PURB Urb = &LocalUrb;
    RtlZeroMemory(Urb, sizeof(struct _URB_CONTROL_VENDOR_OR_CLASS_REQUEST));
    Urb->UrbHeader.Length = sizeof(struct _URB_CONTROL_VENDOR_OR_CLASS_REQUEST);
    Urb->UrbHeader.Function = URB_FUNCTION_VENDOR_DEVICE;
    Urb->UrbControlVendorClassRequest.TransferFlags = USBD_TRANSFER_DIRECTION_OUT;
    Urb->UrbControlVendorClassRequest.TransferBufferLength = Length;
    Urb->UrbControlVendorClassRequest.TransferBuffer = DataBytes;
    Urb->UrbControlVendorClassRequest.TransferBufferMDL = NULL;
    Urb->UrbControlVendorClassRequest.Request = 0x0A0; // "Anchor Load"
    Urb->UrbControlVendorClassRequest.Value = StartLocation;
    return SendUrbAndWait(fdo, Urb);
}

NTSTATUS EZUSB_Reset(PDEVICE_OBJECT fdo, UCHAR ResetBit) {
    PAGED_CODE();
    // EZUSB_Reset is a special case of SetEZUSBMemory
    return SetEZUSBMemory(fdo, 0x07F92, 1, &ResetBit);
}

```

```

NTSTATUS GetFirmware(PDEVICE_OBJECT fdo) {
    PAGED_CODE();
    // Open the Firmware file and load it into a local buffer
    // First create the name of the file that we need to open
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS Status;
    Status = GetIdentity(fdo);
    if (Status != STATUS_SUCCESS) {
        KdPrint((DRIVERNAME "Could not get VID and PID from IO device\n"));
        return Status;
    }
    // Create a filename from the current VID and PID
    WCHAR Filename[] = L"\\SystemRoot\\system32\\drivers\\vvvvpppp.hex";
    Filename[29] = HexCharacter[(pdx->VID >> 12) & 0x0F];
    Filename[30] = HexCharacter[(pdx->VID >> 8) & 0x0F];
    Filename[31] = HexCharacter[(pdx->VID >> 4) & 0x0F];
    Filename[32] = HexCharacter[pdx->VID & 0x0F];
    Filename[33] = HexCharacter[(pdx->PID >> 12) & 0x0F];
    Filename[34] = HexCharacter[(pdx->PID >> 8) & 0x0F];
    Filename[35] = HexCharacter[(pdx->PID >> 4) & 0x0F];
    Filename[36] = HexCharacter[pdx->PID & 0x0F];
    // Need to use Walter Oney's portable file subsystem here since this driver may be running before
    // the file system is running
    HANDLE FileHandle;
    Status = OpenFile(Filename, TRUE, &FileHandle);
    KdPrint((DRIVERNAME "OpenFile returned with status = %8.8x\n", Status));
    if (!NT_SUCCESS(Status)) return Status;

    pdx->HexFileLength = (ULONG) GetFileSize(FileHandle);
    pdx->HexFileBuffer = (PCHAR) ExAllocatePool(NonPagedPool, pdx->HexFileLength);
    if (!pdx->HexFileBuffer) return STATUS_NO_MEMORY;
    ULONG BytesReturned;
    Status = ReadFile(FileHandle, pdx->HexFileBuffer, pdx->HexFileLength, &BytesReturned);
    if (!NT_SUCCESS(Status)) KdPrint((DRIVERNAME "ReadFile failed with status = %8.8x\n", Status));
    else KdPrint((DRIVERNAME "ReadFile returned %d bytes\n", BytesReturned));
    CloseFile(FileHandle);
    return Status;
}

bool GetNextCharacter(PDEVICE_OBJECT fdo, PCHAR BufferPtr) {
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    *BufferPtr = pdx->HexFileBuffer[pdx->ByteOffset];
    if (pdx->ByteOffset++ < pdx->HexFileLength) return true;
    return false;
}

bool GetNextLine(PDEVICE_OBJECT fdo, PCHAR BufferPtr) {
    PAGED_CODE();
    // Get the next line from the input file, return FALSE if EOF
    // First find the ':'
    do { if (!GetNextCharacter(fdo, BufferPtr)) return false; } while (*BufferPtr != ':');
    // Then read until the end of the line
    do { if (!GetNextCharacter(fdo, ++BufferPtr)) return false; } while (*BufferPtr != 10); // EOL
    return true;
}

SHORT value(char Entry) {
    PAGED_CODE();
    for (SHORT i = 0; i<sizeof(HexCharacter); i++) {if (Entry == HexCharacter[i]) return i;}
    KdPrint((DRIVERNAME "Invalid Hex character '%2.2x' in file\n", Entry)); return 0;
}

////////////////////////////////////
// This is where the work is done in this device driver

NTSTATUS StartDevice(PDEVICE_OBJECT fdo, PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated) {
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    KdPrint((DRIVERNAME "Starting IO device\n"));
    // Starting the device involves downloading firmware into the EZ-USB component

    // Open the HEX file and read it into a local buffer
    NTSTATUS Status = GetFirmware(fdo);
    if (Status != STATUS_SUCCESS) {
        KdPrint((DRIVERNAME "Could not get device firmware (%8.8x)\n", Status));
        return Status;
    }
}

```

```

// Loading is done in two passes since AnchorLoad only operates on internal memory addresses
// Load external memory first using a helper "mover" program
// First ensure that the EZ-USB 8051 is RESET before downloading
    Status = EZUSB_Reset(fdo, 1);
    if (Status != STATUS_SUCCESS) return Status;
// Since we are still here (previous line did not return) no need to check this Status further

    UCHAR MoverCode[] = // 8051 Object code for the Mover program
        {0x90, 0x7F, 0x9E, 0x74, 0x0C0, 0x0F0, 0x90, 0x7F, 0x95, 0x0F0, 0x75, 0x92, 0,
        0x78, 0x24, 0x0E2, 0x60, 0x10, 0x0FF, 8, 0x0E2, 0xF5, 0x83, 8, 0x0E2, 0x0F5, 0x82,
        8, 8, 0x0E2, 0x0F0, 0x0A3, 0x0DF, 0x0FA, 0x80, 0x0FE, 0x0F, 0x0FF, 0x0F0, 0, 0x4D,
        0x6F, 0x76, 0x65, 0x72, 0x20, 0x49, 0x6E, 0x73, 0x74, 0x61, 0x6C, 0x6C, 0x65, 0x64};
    SetEZUSBMemory(fdo, 0, sizeof(MoverCode), MoverCode);
    EZUSB_Reset(fdo, 0); // unRESET the EZUSB CPU so that it executes this code

    UCHAR Record[40];
    char Buffer[80];
    int Pass, i;
    for (Pass = 1; Pass<3; Pass++) {
        pdx->ByteOffset = 0; // Start at the beginning of the Hex data file
        EZUSB_Reset(fdo, 1); // Stop the EZUSB CPU
        USHORT ByteCount, LoadAddress, RecordType;
// Read the HEX records one at a time and load them onto the development board
        while (GetNextLine(fdo, Buffer)) {
            ByteCount = Record[0] = (value(Buffer[1]) << 4) + value(Buffer[2]);
            for (i = 1; i<ByteCount+4; i++) Record[i] = (value(Buffer[1+i+i]) << 4) + value(Buffer[2+i+i]);
            LoadAddress = (Record[1] << 8) + Record[2];
            RecordType = Record[3];
            if (RecordType == 0) {
// A content record (=0) has been located
                if ((Pass == 1) && (LoadAddress > 0x1FFF)) {
//
                    KdPrint((DRIVERNAME "Loading %2.2xH bytes at upper address %4.4xH\n", ByteCount, LoadAddress));
                    SetEZUSBMemory(fdo, 0x24, ByteCount+4, Record);
                    EZUSB_Reset(fdo, 0); // unRESET EZUSB CPU so that it will move the data
                    EZUSB_Reset(fdo, 1); // Stop the EZUSB CPU again
                }
                if ((Pass == 2) && (LoadAddress + ByteCount) < 0x1B3F) {
//
                    KdPrint((DRIVERNAME "Loading %2.2xH bytes at lower address %4.4xH\n", ByteCount, LoadAddress));
                    SetEZUSBMemory(fdo, LoadAddress, ByteCount, &Record[4]);
                }
            }
        }
    }
// All done, allow the EZ-USB CPU to 'renumerate'
    KdPrint((DRIVERNAME "Firmware downloaded!\n"));
    EZUSB_Reset(fdo, 0);
// Interesting philosophical point - the original device that caused this driver to run no longer
// exists, since the driver has downloaded it with a new identity.
// It would be preferable to return UNSUCCESSFUL so the OS will mark us STOPPED. (this also prevents "Surprise
Removal" messages on Win2K)
// However, the USDB.SYS driver will Suspend our I/O device if we do. So return SUCCESS
    return STATUS_SUCCESS;
}

VOID StopDevice(IN PDEVICE_OBJECT fdo, BOOLEAN oktouch /* = FALSE */) {
    PAGED_CODE();
    KdPrint((DRIVERNAME "Device Stopped\n"));
}

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

#pragma LOCKEDCODE

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

```