Program source code usage.

Last updated 9-10-2001

For best viewability of this documentation you should
set your text format to wrap to window rather than margins.

See Install.doc for device installation information

See Updt_USB.doc for update notes. Be sure to always review update information in that file.

See important programing notes at end of this file.

NOTE:
Search on "Extended Explanations" (at end of this file) for more info on each. Most commonly used functions are marked with a " * ". They are also the functions that I've tried to more fully implement in the sample application(s) code listings.

====================================

DLL function prototypes and explanations

====================================

Highlight function and then use "search" feature to locate description of one of the function names listed below:
tip: if argument in function prototype is greyed, it is no longer used by DLL but the argument is still required as a placeholder.

Inititalization:
DLL_Init

Device handling and enumeration:
DLL_ReInitOneDev
DLL_GetDeviceIDList
DLL_MarkOneDevAsInUse
DLL_ShowGlobalDevIDList
DLL_InstanceInitDeviceList
DLL_ShowDeviceDescriptor
DLL_ReleaseDeviceFromIDList
DLL_ShowInterfaceDescriptors
DLL_ShowEndpointDescriptors
DLL_ShowConfigurationDescriptor

DLL debug related:
DLL_WriteStrToDebugFile
DLL_StopDebugOutput

DLL_SetSpecialDebugOptions

Scan related:
DLL_StartScan_DIO
DLL_EndScan_DIO
DLL_ReadScanData_DIO
DLL_ReadEndOfScanData
DLL_ReadScanDataWithDigin.
DLL_GetEndOfScanBuffSize

Other I/O:
DLL_SetRate
DLL_ResetDevice
DLL_ReadWriteOneDevice

Misc:
DLL_GetDevVersion
DLL_GetVersionInfo
DLL_ZeroSharedDevArray
DLL_ShowSharedDevArray
DLL_ShowSharedGUIDStringArray

==================================
Brief Explanations (extended explanations follow):
==================================
****
Initialization
****

========== *
BOOLEAN
DLL_Init(HWND hAppMainWin, *HMODULE _hLibSETUPAPI, HMODULE _hLibUSER32*,
SHORT sDebugLevel, BYTE* numDevices, *BYTE *DeviceStatus*,
USHORT *LLABS_DeviceIDList);

1.) DLL checks for presence of Device Driver.
2.) DLL checks for installed 301/302 devices.
3.) DLL creates memory-mapped files to contain list of Device ID's of all installed 301/302 devices, their GUID strings, and current status.
4.) DLL initializes pointers. DLL sets it's pointers to equal those of the calling application.

==========

==========

BOOLEAN
DLL_WriteStrToDebugFile(char* szStrToPrint);

DLL writes string to debug file (if debugging enabled) at current location.

==========

==========

BOOLEAN
DLL_StopDebugOutput(void);

DLL stops sending debug information to file.

==========

==========

BOOLEAN
DLL_SetSpecialDebugOptions(usDevList(0), bDbgLvl(0), bDbgLvlThisDev, iDbgFlags);

Set special options for DLL debug file output.

==========

****

Device handling and enumeration

****

========== *

BOOLEAN
DLL_GetDeviceIDList(void);

DLL looks for all installed 301/302 devices. It then updates it's list to reflect the presence of devices that have been installed or removed since the last call to this function.

==========

========== *

BOOLEAN
DLL_InstanceInitDeviceList(BOOLEAN fDisplayLists);

DLL compares it's list of installed 301/302 devices with the list of requested devices in the calling applications Device ID list

==========

========== *

BOOLEAN
DLL_ReleaseDeviceFromIDList(USHORT* LLABSDeviceIDList, BOOLEAN fAnyHandle);

DLL marks the Device ID's in it's list as DEVICE_ATTACHED_BUT_NOT_IN_USE (1)

==========

==========

BOOLEAN

DLL_ShowGlobalDevIDList(char* pDevListBuff);
DLL copies data to to memory pointed to by pDevListBuff.
===========

===========
BOOLEAN
DLL_ShowDeviceDescriptor(char* pDynamicDeviceDescArray, USHORT usDevID);
DLL copies data to to memory pointed to by pDynamicDeviceDescArray.
===========

===========
BOOLEAN
DLL_ShowConfigurationDescriptor(char* pDynamicConfigArray, USHORT usDevID);
DLL copies data to to memory pointed to by pDynamicConfigArray.
===========

===========
BOOLEAN
DLL_ShowInterfaceDescriptors(BYTE* pDynamicArray, USHORT usDevID);
DLL copies data to to memory pointed to by pDynamicArray.
===========

===========
BOOLEAN
DLL_ShowEndpointDescriptors(BYTE* pDynamicEndpointDescArray, USHORT usDevID,
SHORT whichEndpoint);
DLL copies data to to memory pointed to by pDynamicEndpointDescArray.
===========

===========
BOOLEAN
DLL_ReInitOneDev(USHORT usDevID)
DLL reinitializes one device.
===========

===========
BOOLEAN
DLL_MarkOneDevAsInUse(USHORT usDevID)
Call this function after calling **DLL_ReinitOneDev().**

****

Device read/write and rate handling

****

========== *

BOOLEAN
DLL_ReadWriteOneDevice(BOOLEAN fReadWrite, BYTE* DynamicDataArray,
        USHORT usDevID, UINT* numBytes, BYTE* pbDigIn, UINT uiReserved);

DLL reads or writes data to device

==========

========== *

BOOLEAN
DLL_SetRate(BYTE whichDevice, UINT sRate);

BOOLEAN
DLL_SetRate(USHORT usDevID, DOUBLE dblRate);

DLL sets it's rate variables to match those of the applications

****

Scanning

****

========== *

BOOLEAN
DLL_StartScan_DIO(USHORT usDevID,
        PSCAN_OBJECT_DIO pScanObj_DIO, BYTE bStartMode);

DLL starts scanning the device.

==========

========== *

BOOLEAN
DLL_ReadScanData_DIO(USHORT usDevID, UINT* DynaVoltageBuff,
        USHORT numPointsToRead, PSCAN_OBJECT_DIO pScanObj_DIO);

DLL Reads the requested number of bytes from the device.

==========

========== *

BOOLEAN
DLL_ReadScanDataWithDigin(USHORT usDevID, UINT* DynaVoltageBuff, BYTE* pbCurChan
        USHORT numPointsToRead, PSCAN_OBJECT_DIO pScanObj_DIO);

Note: Only make this call with Device IDs greater than 154.

DLL Reads the requested number of bytes from the device. Returns current digital input with each scan.

==========

========== *

BOOLEAN
DLL_EndScan_DIO(USHORT usDevID, BOOLEAN fReserved,
    UINT uiReserved, BYTE bReserved);
DLL stops scanning the device. Pretty self-explanatory. This just ends the scan. The scan will take a little while to end, since a system calibration is done automaticlly by the DLL at the end of each scan.
==========

==========
BOOLEAN
DLL_GetEndOfScanBuffSize(USHORT usDevID,UINT* uiNumPoints);
DLL gets current number of scans in buffer. This is needed to properly size the buffer passed in the following function.
==========

==========
BOOLEAN
DLL_ReadEndOfScanData(USHORT usDevID, UINT* DynaVoltageBuff, USHORT numPointsToRead);
DLL fills DynaVoltageBuff with number of scans listed in numPointsToRead. Depending on the scan rate and number of devices being scanned, scan data can continue to be placed in the buffer, before the scan completely ends. If the data is needed, it can be retrieved using the combination of this and the preceding function.

****
Misc device I/O
****

==========
BOOLEAN
DLL_ResetDevice(USHORT usDevID, BYTE bReserved);
DLL resets device usDevID. Note that after a device is reset, you will need reinitilize it using the signon process.
==========

==========
BOOLEAN
DLL_ZeroSharedDevArray(VOID);
DLL sets to zero all members in DevID list. This sets to zero all members in the Device ID array. This is a utility function and not normally used. If more than one application, or multiple threads in the same application are running and this is done unpredictable results could occur
==========

==========
BOOLEAN
DLL_ShowSharedDevArray(VOID);
DLL displays (using Windows message box) DevID list.
==========

==========
BOOLEAN
DLL_ShowSharedGUIDStringArray(VOID);
DLL displays (using Windows message box) DevID GUID strings.
==========

==========
BOOLEAN
DLL_GetDevVersion(USHORT usDevID, USHORT* usDevVersion, BOOLEAN fDisplayVersion);
DLL displays (using Windows message box) Device version, and passes the value to app.
==========

==========
BOOLEAN
DLL_GetVersionInfo(USHORT usDevID, BOOLEAN fDisplayVersion,
USHORT* usDevVersion, char* cpDLLVersion, char* cpDrvrVersion);
DLL displays (using Windows message box) DLL, DeviceDriver, and hardware version info (using Windows message box) Device version, and passes the information to app.
==========

========================
=== Extended Explanations =
========================

========================
=== Extended Explanations =
========================

========================
=== Extended Explanations =
========================

****
Initialization
****

========= *

BOOLEAN
DLL_Init(HANDLE hAppMainWin, *HMODULE _hLibSETUPAPI, HMODULE _hLibUSER32*,
SHORT sDebugLevel, BYTE* numDevices, *BYTE *DeviceStatus*,
USHORT *LLABS_DeviceIDList);

1.) DLL checks for presence of Device Driver.
2.) DLL checks for installed 301/302 devices.
3.) DLL creates memory-mapped files to contain list of Device ID's of all installed 301/302 devices, their GUID strings, and current status.
4.) DLL initializes pointers. DLL sets it's pointers to equal those of the calling application.
5.) DLL has capability for creating a file that contains runtime debug information as well as various levels of runtime messaging to the application. These features are controlled by entries within the LL_USB.INI file installed in the "Windows" directory during driver installation. See that file for more information. If the DLL doesn't find that file, it uses default values of "no debug output" and "basic signon messaging and critical error messaging being sent to a modeless dialog box". That box requires no user interraction and does not interrupt normal program flow.
6.) The following arguments are now ignored by DLL, but kept in place for backward compatibility

HANDLE _hLibSETUPAPI
HANDLE _hLibUSER32
BYTE *DeviceStatus

7.) **Important note:** *LLABS_DeviceIDList is a pointer to a USHORT (16-bit unsigned short integer) list of 64 Device IDs. That pointer must be kept global until the application has unloaded. That pointer is shared by the DLL and used by the DLL in it's unload function. Even though an application should call the **DLL_ReleaseDeviceFromIDList()** function before closing, the DLL uses the same list passed during the call to this function when it does a final inspection of any I/O pipes or memory allocated for devices that the application was using. It would be an error for example to create a array as a private member to a class and then pass a pointer to that array to this function, since it may no longer be within scope when the DLL is unloaded either by an explicit call by the application to the WIN-API function to unload a DLL, or part of the runtime cleanup done automatically by some applications as they exit.

==========

==========

BOOLEAN
DLL_WriteStrToDebugFile(char* szStrToPrint);

DLL writes string to debug file (if debugging enabled) at current location.

==========

==========

BOOLEAN
DLL_StopDebugOutput(void);

DLL stops sending debug information to file.

==========

==========

BOOLEAN
DLL_SetSpecialDebugOptions(USHORT* usDeviceIdsToEnableDisable,
BYTE* bPerDevDbgLvl, BYTE bDefaultDbgLvl, UINT uiFlags);

Set special options for DLL debug file output. Pass a list of 64 device ID's that your application is using along with the debug level for each. The DLL will write debug file information pertaining to the Devices listed in that list, based on the values in the other arrays. Be sure to use the same index for each list. For example if you have 2 devices to debug, pass a 64 member usDeviceIdsToEnableDisable list with all but the first two set to zero. If you want the first device to be debugged at 2 and the second at 5 then put that device Id in index 0, and place a 2 at index 0 and a 5 at index 1 in the bPerDevDbgLvl list. Set bDefaultDbgLvl to a default debug value for non-device specific debug output. For example, during enumeration and similar functions in the DLL, there is not specific device ID being handled. This function can only be called after initialization in order to fine-tune the debug file output over the standard debug file output parameter passed in the call to **DLL_Init().** The uiFlags variable isn't yet used.

==========

****

Device handling and enumeration

****

========== *

BOOLEAN
DLL_GetDeviceIDList(void);

DLL looks for all installed 301/302 devices. It then updates it's list to reflect the presence of devices that have been installed or removed since the last call to this function. This is a call the application can make if it has received the WM_DEVICECHANGE message with notification that a device was added. With this call, the DLL will locate any new 301/302 devices that may have been added (or removed). You can then inspect the list of devices using:

DLL_ShowGlobalDevIDList
DLL_ShowSharedDevArray

or use **DLL_InstanceInitDeviceList** (described below) with a list of devices you would like to access and see if they are now available.

==========

========== *

BOOLEAN
DLL_InstanceInitDeviceList(BOOLEAN fDisplayLists);

DLL compares it's list of installed 301/302 devices with the list of requested devices in the calling applications Device ID list. The DLL has a copy of that list from the call to that should have previously been made to **DLL_Init()** If a matching Device ID is found, then the DLL marks that device in it's list as being "IN_USE", thus preventing another application from attempting to access it. If the ID in the calling application isn't found in the DLL's list of installed devices, then that Device ID in the application's DevID array is marked with a zero. Therefore it is important for the calling application to save a copy of it's list before the call which it can compare with the shared list after the call to to determine which devices are available for access. Note that there is a Boolean parameter passed. Set that parameter to TRUE only if you would like the DLL use a Message Box to display a copy of it's and the application's lists.

==========

========== *

BOOLEAN
DLL_ReleaseDeviceFromIDList(USHORT* LLABSDeviceIDList, BOOLEAN fAnyHandle);

DLL marks the Device ID's in it's list as no longer in use by this application. This is an important call for the calling application to make before exiting. Memory mapped files are created and maintained by the DLL to reflect the availability and current status of all installed 301/302 devices. When an application connects to a device, that device isn't available to another application until the original application releases it's connection to it. If an application exits without freeing the devices that it was connected to, that/those device(s) will no longer be available again until they are released by a call to this function It is definitely best to just release the devices when the application exits.

Handling of devices in this manner is required for smooth interraction between mulitple applications which may try and access the same devices.

==========

==========

BOOLEAN
DLL_ShowGlobalDevIDList(char* pDevListBuff);

DLL copies data to memory pointed to by pDevListBuff. The string is null terminated and contains complete list of installed device ID's. Application can then use whatever means to display or otherwise deal with that string. Allocated memory for string must be 2048 bytes.

==========

==========

BOOLEAN
DLL_ShowDeviceDescriptor(char* pDynamicDeviceDescArray, USHORT usDevID);

DLL copies data to memory pointed to by pDynamicDeviceDescArray. The string is null terminated and contains Device Descriptor entries for this device ID. Application can then use whatever means to display or otherwise deal with that string. This and the following three functions can be a valuable asset in interracting with our USB devices. Allocated memory for string must be 2048 bytes.

==========

==========

BOOLEAN
DLL_ShowConfigurationDescriptor(char* pDynamicConfigArray, USHORT usDevID);

DLL copies data to memory pointed to by pDynamicConfigArray. The string is null terminated and contains data showing Configuration Descriptor entries for this device ID. Application can then use whatever means to display or otherwise deal with that string. Allocated memory for string must be 2048 bytes.

==========

==========

BOOLEAN
DLL_ShowInterfaceDescriptors(BYTE* pDynamicArray, USHORT usDevID);

DLL copies data to memory pointed to by pDynamicArray. The string is null terminated and contains data showing Interface Descriptor entries for this device ID. Application can then use whatever means to display or otherwise deal with that string. Allocated memory for string must be 2048 bytes.

==========

==========

BOOLEAN

DLL_ShowEndpointDescriptors(BYTE* pDynamicEndpointDescArray, USHORT usDevID,
SHORT whichEndpoint);

DLL copies data to memory pointed to by pDynamicEndpointDescArray. The string is null terminated and contains data showing Endpoint Descriptor entries for this device ID. Application can then use whatever means to display or otherwise deal with that string. Allocated memory for string must be 2048 bytes.

==========

==========

BOOLEAN

DLL_ReInitOneDev(USHORT usDevID)

DLL reinitializes one usDevID. This call is used to reinitialize a device that failed to initialize along with other devices when more than one device is connected. If only one device is connected, use normal initialization function for preliminary initialization. If a single device fails for some reason and needs to be reinitialized it may be easier to use this function than using the various steps required to reinitialize all devices using the normal call to **DLL_Init()**.

==========

==========

- following call being phased out.

BOOLEAN

DLL_MarkOneDevAsInUse(USHORT usDevID)

Call this function after calling **DLL_ReinitOneDev().** This function marks the various lists in the DLL to show that this device belongs to the calling application.

****

Device read/write and rate handling

****

========= *

BOOLEAN

DLL_ReadWriteOneDevice(BOOLEAN fReadWrite, BYTE* DynamicDataArray,
USHORT usDevID, UINT* numBytes, BYTE* pbDigIn, UINT uiReserved);

The DLL reads or writes data to device. DLL requires constant to determine if it is to do a read or write, an array of byte size data, the Device ID, and the number of bytes to read or write. At this time all reads require a 32-byte array and all writes require an 8-byte array.

==========

========= *

BOOLEAN
DLL_SetRate(BYTE whichDevice, UINT sRate);
BOOLEAN
DLL_SetRate(USHORT usDevID, DOUBLE dblRate);

The DLL sets it's rate variables to match those of the applications. This is required to make operation of the DLL more efficient. Since the rate change is performed by sending tokens using the generic **DLL_ReadWriteOneDevice** the DLL is unaware of a rate change until this call is made. Of course this function should only be passed a rate that was succefully written to the device.

****

Scanning

****

========== *

BOOLEAN
DLL_StartScan_DIO(USHORT usDevID,
    PSCAN_OBJECT_DIO pScanObj_DIO, *BYTE bStartMode*);

DLL starts scanning the device. Updating the various members of the app's struct PSCAN_OBJECT_DIO, with each call to the function below. Note that there is no thread running in the DLL, so update of the structured data is only done with each call to **DLL_ReadScanData_DIO.**
BYTE bStartMode arg is required but value is ignored
===========

========== *

BOOLEAN
DLL_ReadScanData_DIO(USHORT usDevID, UINT* DynaVoltageBuff,
    USHORT numPointsToRead, PSCAN_OBJECT_DIO pScanObj_DIO);

This function uses the following structered data format:

```
typedef struct _SCAN_OBJECT_DIO {
        BYTE                SO_bScanType;                   // (APP WRITES)
Definitions below
        #define SINGLE_CHAN_SCAN                    1
        #define MULTI_CHAN_SCAN                     2
        #define MULTI_CHAN_CAL_SCAN                 3
        #define SINGLE_CHAN_DIGIN_SCAN              4
        #define MULTI_CHAN_DIGIN_SCAN               5
        #define MULTI_CHAN_CAL_DIGIN_SCAN           6

        DOUBLE              SO_dblRate;             // (APP WRITES) device rate

        UINT                SO_uiTotalPointsReadByDrvr;  // (DLL WRITES) points read by
driver.
        UINT                SO_uiPointsInBuffer;    // (DLL WRITES) points in
DLL/Driver shared buffer.

        UINT                SO_uiStatus;            // (DLL WRITES) Definitions
below
          #define SCAN_STATUS_UKNOWN                      0x00    // Driver status unknown
          #define SCAN_RUNNING                            0x01    // Driver scan thread is
running
          #define SCAN_DATA_REQUEST_SUCCEEDED                     0x02    // Pending
driver to device data request succeeded
          #define SCAN_ENDING                                     0x04    // DLL has
requested driver to end scanning
          #define SCAN_HALTED                                     0x08    // Driver scan
thread was halted (possible I/O err, device unplug, etc)
          #define SCAN_CHECKSUM_ERROR                     0x10    // Driver's data
checksum doesn't agree with device's checksum
          #define SCAN_VOLTAGE_BUFFER_WRAP                0x20    // The circular data
buffer shared between DLL and Driver has wrapped
          #define SCAN_MICRO_CODE_BUFF_WRAP               0x40    // The device's
(hardware) internal buffer has wrapped
          #define SCAN_DEVICE_IO_ERROR                    0x80    // Driver received an I/O
error
          #define SCAN_DEVICE_REMOVED                     0x100   // The USB cable has
been disconnected
          #define SCAN_DEVICE_NOT_FOUND                   0x200   // The DLL was unable
to find the requested device to begin the scan

          UINT              SO_uiSizeVoltageArray;        // (APP WRITES) The size of
app's voltage array.
          UINT              SO_uiCurHead;                 // (DLL WRITES) The device
driver increments whenever it adds a byte to DLL/Driver buffer
          UINT              SO_uiCurTail;                 // (DLL WRITES) The DLL
increments this whenever it reads a byte DLL/Driver buffer
          BYTE              SO_bDigitalInput;             // (DLL WRITES) Current digital
input value. Driver updates this every 10th point read
          BYTE              SO_bChecksum;                 // (DLL WRITES)  Driver's
current checksum
          BYTE              SO_bDeviceDisconnect;   // (DLL WRITES) required but value is
ignored (backward compatibility)

          clock_t           SO_clkScanStartTime;          // not currently changed
          clock_t           SO_clkScanEndTime;            // not currently changed

          BYTE              SO_bScanArg;                  // (APP WRITES) Number of
channels to scan.
} SCAN_OBJECT_DIO, *PSCAN_OBJECT_DIO;
```

DLL Reads the requested number of points (3 bytes per point) from the device. If the requested number of points is unavailable, the DLL returns FALSE. Otherwise, the DLL returns TRUE and OR's SCAN_DATA_REQUEST_SUCCEEDED onto the existing SO_Status member of the SCAN_OBJECT_DIO struct. Note the various flag bits for the SO_Status member. Flags are OR'd on by the DLL. It is best to call this function as frequently as possible, and inspect the flags with each call in case there was an I/O error in the device. The DLL will also return FALSE if an error occurred during the read.

Note: Application's SCAN_OBJECT_DIO struct is only updated when this function is called.

Note that 10 points are read (see speicial note below for exception) with each read of the device by the device driver. Therefore it is most efficient if you also call this function with numPointsToRead being a multiple of 10. If once you return, you discover that SO_uiPointsInBufferis growing too quickly, it is then most efficient to quickly make multiple calls to this function with requests for 10 points each than to call for a larger number of points, unless your application has a separate running thread making the calls. This is especially true for VisualBasic developers since application GUI events have no way of being handled while the main application thread is away in a function call.

===================
Note:
===================

There are 3 entries in LL_USB.INI to help improve driver capability to maintain stability against other drivers that may behave incorrectly. They are explained below. **Use these entries only if you receive frequent "microcode buffer wrap" errors**.

=====================
ScanSystemBoostLevel (default = 0)
=====================

Use this feature only if you experience "microcode buffer wrap" errors

Explanation:

Improves thread time-slicing handling within the device driver. Any device driver has the opportunity to "take over" the system for a specified period of time. To do so, it's able to set it's internal thread priority to various levels. However, a mis-behaved driver can still "hog the system" using certain techniques which can prevent other drivers from getting their time slice. We've found that some game programs and many software applications that "run in the background" are notorious for that, which can prevent our driver from getting enough time slices to make the required data transfers from our hardware in the allocated time. This is particularly obvious at high scan rates. A failure to do the required transfers shows as the "microcode buffer wrap" flag being set while scanning. The flag is found as one of the bits (bit-5) of the SO_Status member of the SCAN_OBJECT structure that is used for communication between the DLL and your application. We've incorporated a way around this issue by creating an option to pass a larger packet size request to the "lower USB driver" which has even a higher priority level than our driver is allowed.

Following is an overview of the feature:

Without using the new "packet boost" feature, our driver reads 32 bytes of data (one packet) at a time from our hardware. That typically translates to 30 bytes of data, and 2 bytes of flags and digital input, which in single channel scan mode, translates to 10 points per read (3-bytes-per-point). However with use of the new packet boost option controlled from the LL_USB.INI file a user can now boost the driver time slicing capability by a factor of up to 100. What this actually does at the driver level, is it allows our driver to make more use of the power of the system driver below the level of ours which has a higher priority level than possible with our own driver. However, this comes with a small price and the price is that your data will be ready for you to read proportionally to the "boost" factor. For example, if you set up a single-channel scan at 1000 Hz without the use of the boost feature, your data will actually be read in 10-point packages and you can retrieve your data from the DLL with a call every 10 milliseconds. If you set the boost factor to 50, your data will be ready to be read from the DLL every 500 milliseconds, at which time

500 points worth of data will be available. No data is missed, it just takes longer to get the larger packet. Keep this in mind when scanning at slower rates, since if a scan is set to 100Hz and the boost value is set to 100, it will take 10 seconds before one read is complete and that read will contain 10000 points.
NOTE: The ending of the scan process will also take longer as the boost factor is increased.

A good example of an application that can "take control of the system" is the game "Pinball" that ships with WinME and Win2k. That game appears to interract with a low-level driver that completely takes over the system, possibly in order to make possible the smooth graphics that it offers to the user. Running that game on a WinME or Win2k machine while doing a single-channel scan at 1000Hz will usually create a "microcode buffer wrap" within a matter of seconds. However with "ScanSystemBoostLevel" set to 50, such a wrap won't occur.

Advantage: Takes advantage of premium slicing given only to "HUB" type drivers.
Disadvanges:
1.) Causes ending of scan to take longer, since each read request by the lower driver has to return with it's packet before for each device before the scanning threads can end.
2.) Data retrieval is available at slower rate dependent on the setting of this value. The actual scan rate is unchanged, it's just that each time you read, you will get back much more data, but less frequently. For example in single-channel scan at 1000Hz with boost of 10, you can read the data every 100ms and you'll get 100 points with each read, since the boost causes driver to get 10 packets, of10 points each, with every read. If you attempt to read every 10ms (the default without use of this flag) your data will be ready only every 10th read.

=====================
StackedIrpCntOverride (default = 0)
=====================

Use this feature ONLY if you experience "microcode buffer wrap" errors, and only attempting to correct the problem by setting value above (ScanSystemBoostLevel) to it's maximum (100).

Overrides default values calculated by DLL based on scan rate and other variables. The term "IRP" stands for "I/0 Request Packet". Our driver communicates with lower drivers (HUB and kernel) in order to communicate with our device. For each read of our device, we pass an "IRP" to the lower driver. Due to processing of the IRPs and data returned from the request, it is usually more efficient to send many at the same time so the lower driver is always busy. However if too many are sent, then time slicing can be taken away from our driver and given to the lower driver.

Advantage: Assures that lower USB driver will always be busy requesting data from our device.
Disadvanges: Too many stacked IRPs can take away from our driver's time slicing and give it to the lower drivers.

=====================
DoRecordDrvrTimerInfo (default = 0)
=====================

Use this feature ONLY for improving debug file output, when driver time-slicing may be questionable.

This should never be set except when debugging time slicing issues in driver which are usually indicated by "microcode buffer wrap" errors. When set, this causes device driver to record timing information and pass it to the DLL which can then write that information to a debug file when the debug file output feature is enabled.

Advantage: Allows display of driver timing issues.
Disadvanges: Puts extra load on driver to record timing information.

==================
End of Note:
==================

==========

========== *

BOOLEAN
DLL_ReadScanDataWithDigin(USHORT usDevID, UINT* DynaVoltageBuff, BYTE* pbDiginBuff, BYTE* pbCurChan
          USHORT numPointsToRead, PSCAN_OBJECT_DIO pScanObj_DIO);
Note: Only make this call with Device IDs greater than 154.

The DLL returns the requested number of data points if available as well as the current digital input with each point. See **DLL_ReadScanData_DIO()** above for pScanObj_DIO information. Call this function with two array pointers. The first is to receive the current 24-bit conversion, and the second is to receive the current digital input. This is the function to call when you've used one of the Digital-Input scan tokens to start the scan. The device will include a digital input with each data point. The DLL will return the digital input that corresponds to the data point using the same index. For example if you do a MULTI_CHAN_DIGIN_SCAN type scan, then DynaVoltageBuff[0] will contain datapoint for channel 0 and pbDiginBuff[0] will contain the digital input that was read at the same time that point was read. To further explain:

if you're doing a non-cal scan on 3 channels and you request 12 points and 12 points are available, then on return your arrays will look like this:
DynaVoltageBuff [chan**0**, chan**1**, chan**2**, chan**0**, chan**1**, chan**2**, chan**0**, chan**1**, chan**2**, chan**0**, chan**1**, chan**2**]

pbDiginBuff [diginForChan**0**, diginForChan**1**, diginForChan**2**, diginForChan**0**, diginForChan**1**, diginForChan**2**, diginForChan**0**, diginForChan**1**, diginForChan**2**, diginForChan**0**, diginForChan**1**, diginForChan**2**]

if you're doing a calibration scan on 3 channels and you request 12 points and 12 points are

available, then on return your arrays will look like this, where "offs" means channel 7:
DynaVoltageBuff [chan**0**, chan**1**, chan**2**, **offs**, chan**0**, chan**1**, chan**2**, **offs**, chan**0**, chan**1**, chan**2**, **offs**]

pbDiginBuff [diginForChan**0**, diginForChan**1**, diginForChan**2**, diginForChan**offs**, diginForChan**0**, diginForChan**1**, diginForChan**2**, diginForChan**offs**, diginForChan**0**, diginForChan**1**, diginForChan**2**, diginForChan**offs**]

I've also added pbCurChan to help in case you should get confused which channel you're requesting data for. You can call this function (**DLL_ReadScanDataWithDigin()**) anytime - even with numPointsToRead = 0 - to get that information. Your variable which that pointer points to should always equal zero when you return from this call. If it does not, then the number of points you're requesting isn't correct for the number of channels you've requested to scan. That value is set on entry into the call, not on exit. Each read should start with channel zero regardless of the number of channels being read.
===========

========== *
BOOLEAN
DLL_EndScan_DIO(USHORT usDevID, BOOLEAN fReserved,
UINT uiReserved, BYTE bReserved);
Note: All but the usDevID parameter are currently and temporarily disregarded.
DLL stops scanning the device. Note that this call can take a little while to complete. That is because a system calibration is done at the end of each scan.
===========

===========
BOOLEAN
DLL_GetEndOfScanBuffSize(USHORT usDevID,UINT* uiNumPoints);
DLL gets current number of points in buffer. This is needed to properly size the buffer passed in the following function.
===========

===========
BOOLEAN
DLL_ReadEndOfScanData(USHORT usDevID, UINT* DynaVoltageBuff, USHORT numPointsToRead);
DLL fills DynaVoltageBuff with number of scans listed in numPointsToRead. Depending on the scan rate and number of devices being scanned, scan data can continue to be placed in the buffer, before the scan completely ends. If the data is needed, it can be retrieved using the combination of this and the preceding function.

****

Misc device I/O

****

==========
BOOLEAN
DLL_ResetDevice(USHORT usDevID, BYTE bReserved);
DLL resets device usDevID. The DLL does a reset of the device. Occasionally the USB interface becomes corrupted and a device reset is required. This is apparent if the DLL finds the device during preliminary signon, but is unable to wrire and/or read from the device.
==========

==========
BOOLEAN
DLL_ZeroSharedDevArray(VOID);
DLL sets to zero all members in DevID list. See notes below for **DLL_ShowSharedDevArray.**

==========

==========
BOOLEAN
DLL_ShowSharedDevArray(VOID);
DLL displays (using Windows message box) DevID list.

These are utility functions made available for access to the DLL's memory mapped files that reflects the installed devices. These functions are used only for special purposes. The DLL creates memory-mapped files when it's **DLL_Init** entry point is called by the first application that loads it. The DLL maintains a list of, and status of, all devices that are installed, in use, or removed. The DLL uses the memory-mapped files to interract with one or more applications that may be concurrently accessing the DLL. Note that one should excercise caution when one is making a call to **DLL_ZeroSharedDevArray**. Once that call is made, the DLL will have cleared it's list of installed devices, and erroneous results could occur if an attempt is made to access devices previously listed, without a new call to **DLL_Init**, or **DLL_GetDeviceIDList**.
==========

==========
BOOLEAN
DLL_ShowSharedGUIDStringArray(VOID);
DLL displays (using Windows message box) DevID GUID strings. See notes above for: **DLL_ShowSharedDevArray.**
==========

==========
BOOLEAN
DLL_GetDevVersion(USHORT usDevID, USHORT* usDevVersion,
BOOLEAN fDisplayVersion);
DLL displays (using Windows message box) Device version if fDisplayVersion is TRUE. Either way it sets app's variable pointed to by usDevVersion to the device version. This version is the device (hardware) version and has nothing to do with DLL or system driver version. That information is of course available internally and via file properties using the Windows file manager.
==========

==========
BOOLEAN

DLL_GetVersionInfo(USHORT usDevID, BOOLEAN fDisplayVersion,
    USHORT* usDevVersion, char* cpDLLVersion, char* cpDrvrVersion);
DLL displays (using Windows message box) DLL, DeviceDriver, and hardware version info (using Windows message box) Device version, and passes the information to app. Please make string storage to accomodate 32 bytes for cpDLLVersion and cpDrvrVersion
==========

==============================
Programming notes
==============================

1. -)
A device can only be accessed by one application at a time. Once an application is connected to a device then that device is no longer available for use by another application. Anytime a call to **DLL_GetDeviceIDList()** is made the DLL will update it's lists, adding any new devices that have been connected and marking as removed, any devices that have been disconnected since the last call to that function.

2. -)
When the debug option is set in LL_USB.INI the DLL creates a subdirectory called M30x_DBG in the folder from which the application is running. The naming convention used for the debug files is a product of the calling application's main form handle, combined with the particuliar file number if multiple files are created. High scanning rates are not possible when debugging is enabled. Since writing the debug information (depending on the level of debugging) is a time consuming process, DLL through-put is greatly reduced. Enabling debug file output is good for trouble-shooting, but should be kept disabled when not troubleshooting.

3. -)
Since the source code file is rather lengthy, I've attempted to group the functions in that file into different groups depending on the type of function that they provide. I've placed a listing of the title of the groupings at the top of the file. If you highlight the particuliar title and then do a edit/find, you can quickly find your way around the various functions within that file. The same more or less also applies to the CPP header file.

5. -)

The configuration file is easily readable, and the vaious line contents are well explained at the end of the file. The DLL does no reading of that file, so the application programmers are of course free to make their own version of that file to suit the need. The current configuration file has many lines that are disregarded by the sample application code I've provided. Those lines exist only to make the file more readable. However when editing the file note that those particuliar lines <u>must exist</u> even thought the data (or just blank line) is disregarded by the application. You are of course free to create your own variety of config file to suite your custom application.

7. -)

Scan buffer size is 32,763 bytes. The current number of scans in the buffer is available in the SO_ScansInBuffer member of the SCAN_OBJECT structure. If scans-in-buffer exceeds the buffer size, then a wrap of the head and tail buffer pointers will occur creating invalid data. Since these sample applications have only one thread running, such a wrap can occur if one takes too long to enter a digital output during a scan operation. That is because no data is being removed from the scan buffer during that funcion. For example at 600Hz, the scan buffer would wrap in approximately 17 seconds (32,768 / (600 * 3-bytes-per-conv)).

8. -)

If the application fails to initialize and a second attempt fails, close the application, disconnect the device, reconnect the device, reopen the app and then try again to initialize. If initialization continues to fail, set the debug option to it's maximum value (5), attempt another initialization, quit the application, and review the contents of the debug file for an indication of what the error might be.

9. -)

Explanation of SCAN_OBJECT structure members:

SCAN_VOLTAGE_BUFFER_WRAP
SCAN_MICRO_CODE_BUFF_WRAP

With each call to **DLL_ReadScanData(),** the application's structure is updated. Note that the struct is only updated when this function is called. If you want your struct updated, but don't want to read any data, just call the function with numPointsToRead set to zero and the function will return false, but your struct will be updated, since it updates that struct every time it returns, whether it returns TRUE or FALSE. The only exception is if the function is called with an invalid DevID. Note that the "flags" of the SO_Status member are only OR'd on. They are never cleared. It is up to the application to clear the flags if and when it has had an opportunity to review them. In regard to the buffer wrap flags, they reflect the status of two different buffers. The hardware has an internal buffer where it stores data which is then read by the device driver. If for some reason the device driver fails to access the hardware frequently enough the hardware's buffer will wrap. On the other hand, if you fail to call **DLL_ReadScanData()** frequently enough, then the DLL's buffer will wrap. Our device driver's thread runs at a maximum priority level. However it is possible for another driver to "hog" the system, which still gives our driver (and others) less of a timeslice even at it's high priority level. This can also happen due to a computer hardware problem. We recently encountered it on a laptop computer when there was a problem with the cooling fan. Whenever the fan turned on, our hardware's buffer would wrap. When we contacted the manufacturer they repaired the error, which fixed the problem. If you see the SCAN_VOLTAGE_BUFFER_WRAP flag set, then you simply need to make more frequent calls to **DLL_ReadScanData().** Either of these situations may make the data appear scrambled.

NOTE: All functions return (some are passed) BOOLEAN data type. BOOLEAN is a 1-byte (unsigned char) data type (same as bool and boolean). TRUE = 1, FALSE = 0 It is important not to confuse data type BOOLEAN with type BOOL. Although they are both TRUE/FALSE types, BOOL is defined as a 32-bit data type. Replacing type BOOLEAN with BOOL in function calls can cause erroneous behavior. Some of the application and/or class wizards in MFC or OWL may set up your classes using the BOOL data type. You may then be inclined to pass or return the same to/from the various DLL function calls.

To confuse matters even more, note that VB uses a data type of Boolean which is a 16-bit signed data type which at one time may have matched the C++ BOOL data type, but no longer does, nor does it match any of the others. If set to True, it is equal to -1 as a signed value and of course 128 as an unsigned value. Therefore it becomes especially important in VB that when prototyping your DLL functions that return or pass a BOOLEAN data type, to prototype using a Byte data type, in place of any reference to BOOLEAN.

For example the following is incorrect for C++:

```
BOOL fMyFlagOut = FALSE, fMyFlagIn;
fMyFlagIn = DLL_InstanceInitDeviceList(fMyFlagOut );
```

And the following is incorrect for VisualBasic:

```
Dim  fMyFlagOut as Boolean: fMyFlagOut = False;
Dim  fMyFlagIn as Boolean
fMyFlagIn  = DLL_InstanceInitDeviceList(fMyFlag )
```

while the following is fine for C++:

```
BOOLEAN bMyFlagOut = FALSE, bMyFlagIn;
bMyFlagIn = DLL_InstanceInitDeviceList(bMyFlagOut );
```

or

```
BOOLEAN bMyFlagIn;
bMyFlagIn = DLL_InstanceInitDeviceList( 0 );
```

or

```
DLL_InstanceInitDeviceList(0 );
```

And the following is fine for VisualBasic:

```
Dim  bMyFlagOut as Byte: bMyFlagOut = 0
Dim  bMyFlagIn as Byte
bMyFlagIn = DLL_InstanceInitDeviceList(bMyFlagOut );
```

or

```
Dim  bMyFlagIn as Byte
bMyFlagIn = DLL_InstanceInitDeviceList( 0 );
```

This file last updated **9-10-2001**

Tim Van Dusen.

Lawson Labs, Inc.