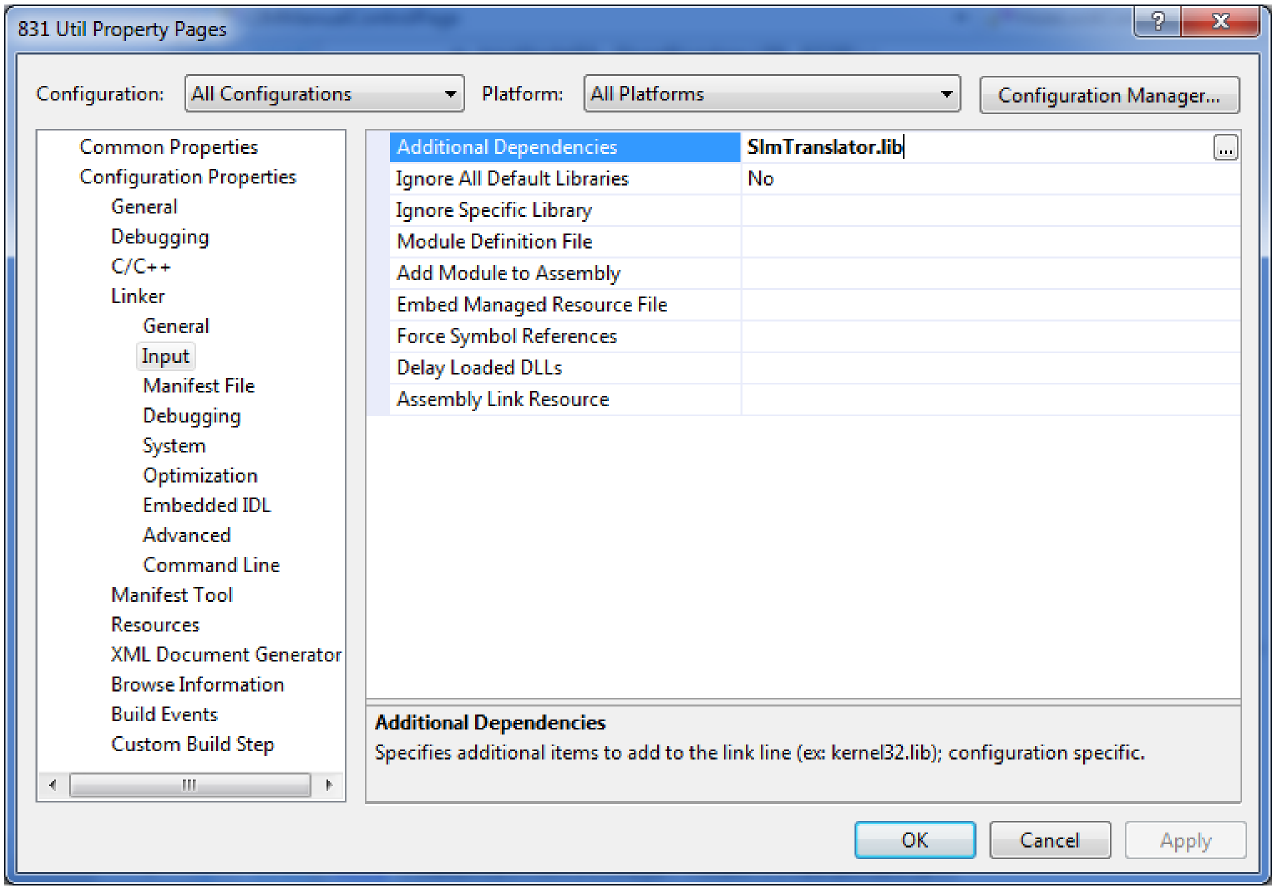
How to use SlmTranslator

Introduction

The SlmTranslator is a dll that provides functions to ‘translate’ data from an 831 or LxT meter into a set of structures which can then be used to manipulate, view, and/or export that data as the user needs. The input data can either be the raw data downloaded from the meter or imported from data stored to a USB thumb drive. For example, SlmUtility-G3 saves the raw downloaded data to a file named slm.ldbin, which is then used as the input file to SlmTranslator.

Adding SlmTranslator to a Project

The SlmTranslator.dll must be linked into your program. In Visual Studio, include the SlmTranslator.lib in your project by either adding it to Project Properties->Linker->Input->Additional Dependencies (as shown in Figure 1) or as a required file in the solution explorer.



Configuration: All Configurations
Platform: All Platforms

Additional Dependencies	SlmTranslator.lib
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.

Figure 1 – Adding SlmTranslator to a Project

Include the following .h files as necessary. (Note that the structures defined in OldStructs.h are deprecated and will be removed in a future version):

```
#include "LxT831.h"
#include "SlmStructs.h"
#include "OldStructs.h" // deprecated structures
```

Then add the following code: (Note: DllImport is defined in LxT831.h)

```
DllImport BOOL StartSlmTranslation(char *pszFileName, int nIndex=1);
DllImport void EndSlmTranslation(void);
DllImport int  GetLDFileType(char *pszFilename, int *pnVersion);
DllImport void SetNotifyWnd(long hWnd, int nMsg);
DllImport BOOL ConvertUSBFileToBinFile(char **pszIn, int nFiles, char *pszOut);
```

The following functions have been deprecated in SlmTranslator.dll version 2.3:

```
DllImport BOOL GetSlmHeader(FileHeader *pFileHeader);
DllImport BOOL GetNextSlmRecord(stSlmRecord *pRec, int nIndex = -1);
DllImport BOOL GetBulkData(stSlmBulkRecord *pSlmBulkRec);
DllImport void GetBulkCounts(DataCnts * pDataCnts);
```

And have been replaced by these extended functions:

```
DllImport BOOL GetSlmHeaderEx(stFileHeader_t &fh);
DllImport BOOL GetNextSlmRecordEx(stSlmRecord_t *pRec, int nIndex = -1);
DllImport void GetBulkCountsEx(stDataCnts_t * pDataCnts);
DllImport BOOL GetBulkDataEx(stSlmBulkRecord_t *pSlmBulkRec);
```

The following function has been added to determine the mode (SLM, RA, FFT) the meter was in when the measurement was created. This call must be preceded by a call to StartSlmTranslation:

```
DllImport int  GetInstrumentMode(void); // -1=N/A, 0=SLM, 1=RA, 2=FFT
```

Please note that these new extended functions take different params from their non-extended counterparts. If you wish to extract RT60 and/or FFT data, you must use the extended functions.

To use the Slmtranslator.dll in other programming languages, refer to the help provided by the distributor.

Using the Translator

The data is retrieved using several structures defined in SlmStructs.h. The main structure used by the translator is stSlmRecord_t which has a member variable, nRecType, to designate the type of data to be retrieved (using the TYPE_* flags defined in LxT831.h) and a union of all the data structures associated with each TYPE that can be retrieved from the file being translated (except the header information, which is retrieved using the stFileHeader_t structure and the GetSlmHeaderEx function).

In order to retrieve the data properly, the calling program must first retrieve the file header, the settings and the preferences structures. These structures will contain all the information necessary to know what type of data is included in the file.

The following is an overview of the order of calling the SlmTranslator functions.
The first function to call is

```
StartSlmTranslation(pszFileName, nIndex);
```

To translate data that has been downloaded from the meter, pszFileName is the name of the file created when the data file(s) were downloaded (i.e slm.ldbin). When multiple data files from the meter are downloaded at the same time, they are all saved into a single download file. The nIndex parameter specifies which record (data file) in the file to translate.

To translate data files stored on a USB flash drive, supply the directory name of the file. For instance, the following is a directory that stores various data files created by the 831:

```
G:\831_Data\831_Data.000
```

The filename that must be used to call StartSlmTranslation is the actual directory name, not the name of any of the files within that directory. The call to retrieve the data stored in the above directory in C++ is:

```
StartSlmTranslation(“G:\\831_Data\\831_Data.000”);
```

The nIndex parameter will default to 1 and is optional in this case since there will never be more than one record in the file when translating data files stored on a USB drive.

Once the translator has been initialized, the next call is to retrieve the file header information using:

```
GetSlmHeaderEx(&p_FileHeader);
```

pFileHeader is a stFileHeader_t structure which is described in SlmStructs.h.

The calling program can ask for specific data structures of which there is only one per file by calling

```
GetNextSlmRecordEx(&SlmRecordData, nDataType);
```

where SlmRecordData is a stSlmRecord_t data structure and nDataType is the desired type of data selected by using one of the TYPE_* flags defined in LxT831.h. For the data types with a variable number of records, repeatedly call GetNextSlmRecordEx with the same datatype flag until the function returns FALSE. When trying to read audio or voice records, pass TYPE_SESSION to GetNextSlmRecordEx, then examine SlmRecordData.nRecType upon return to determine which type it actually is (TYPE_AUDIO, TYPE_VOICE or TYPE_SESSION).

The calling program can also retrieve all data at once by calling GetNextSlmRecordEx in a while loop with a data type of TYPE_ALL. The calling program must then process each data type in a switch statement (`switch (nRecType)`) as the data is returned. Both of these methods are illustrated in the example program.

Some data types have a variable number of records for that type. These include Time History records, Measurement (Interval) records, Exceedence (Event) records, Voice records, and Audio records. There are two ways to retrieve all the records for these data types. The first is to retrieve them one at a time as described above. The second method is to use the bulk data functions to retrieve all the records of a given data type in single call. This necessitates calling GetBulkCountsEx to retrieve the number of records of each data type, allocating the storage required, and then calling GetBulkDataEx with the appropriate TYPE_BULK_* flag set for the desired data type.

```
void GetBulkCountsEx(stDataCnts_t * pDataCnts);

BOOL GetBulkDataEx(stSlmBulkRecord_t * pBulkRec);
```

Once all of the data has been retrieved, the calling program must call

```
EndSlmTranslation();
```

This allows the translator to shut down cleanly and delete any allocated memory used during the translation.

Deleting allocated Storage

The SlmTranslator allocates memory for certain members of various structures. When the calling program is finished with that structure, it must ‘free’ the allocated memory, if any. When the pointer variable is not null, it means memory was allocated for that data in the SlmTranslator. The following function demonstrates which pointers need to be checked and how to deallocate memory.

```
void CTranslateMngr::DeleteAllocated(stSlmRecord_t *p)
{
   int i = 0;
   HANDLE hProcessHeap = GetProcessHeap();

   if ((hProcessHeap != NULL) && (p != NULL))
   {
      switch (p->nRecType)
      {
         case TYPE_OVERALL:
            for (i = 0; i < NUMSPECTRABINS; i++)
            {
               if (p->m_OVData.m_pnSpectralLns[i])
               {
                  HeapFree(hProcessHeap, 0, (void *)(p->m_OVData.m_pnSpectralLns[i]));
                  p->m_OVData.m_pnSpectralLns[i] = NULL;
               }
            }
            break;
         case TYPE_EXCEEDENCE:
            if (p->m_Exceedences.m_ETHData.m_pETHRecs)
            {
               HeapFree(hProcessHeap, 0, (void *)(p->m_Exceedences.m_ETHData.m_pETHRecs));
               p->m_Exceedences.m_ETHData.m_pETHRecs = NULL;
            }
            break;

            case TYPE_AUDIO:
               if (p->m_AudioRec.m_pbyData)
               {
                  HeapFree(hProcessHeap, 0, (void *)(p->m_AudioRec.m_pbyData));
                  p->m_AudioRec.m_pbyData = NULL;
               }
               break;

            case TYPE_VOICE:
               if (p->m_SpeechRec.m_pbyData)
               {
                  HeapFree(hProcessHeap, 0, (void *)(p->m_SpeechRec.m_pbyData));
                  p->m_SpeechRec.m_pbyData = NULL;
               }
               break;

           case TYPE_FFT:
           case TYPE_FFTHISTORY:
              if (p->m_Fft.stLevels.pfLevelAvg)
              {
                 HeapFree(hProcessHeap, 0, (void *)(p->m_Fft.stLevels.pfLevelAvg));
                 p->m_Fft.stLevels.pfLevelAvg = NULL;
              }
              if (p->m_Fft.stLevels.pfLevelMax)
              {
                 HeapFree(hProcessHeap, 0, (void *)(p->m_Fft.stLevels.pfLevelMax));
                 p->m_Fft.stLevels.pfLevelMax = NULL;
              }
              break;

          default:
             break;
      }
   }
}
```

How the data is deleted in other languages is left to the user.

Determining File Types

A new function has been added to the translator to aid the application developer in knowing what type of file the user is trying to translate. The function

```
int GetLDFileType(char *pszFilename, int *pnVersion);
```

will return one of the following types, based on the specified filename

```
#define LDFILETYPE_UNKNOWN       0
#define LDFILETYPE_SLMDL         1
#define LDFILETYPE_BINARY        2
#define LDFILETYPE_USBFILE       3
```

If the filetype is LDFILETYPE_BINARY, the *pnVersion parameter will also be filled in with the binary file version.

If filetype indicates LDFILETYPE_USBFILE, the file (actually a folder) can be converted to a standard binary file by calling

```
BOOL ConvertUSBFileToBinFile(char **pszIn, int nFiles, char *pszOut);
```

And specifying the input files (folder) and output file.

Progress Bars

To use the progress functions in the SlmTranslator, do the following:

(The following are defined in LxT831.h)

```
#define LDM_TRANSLATORNOTIFY          WM_USER+0x7FFF
#define TM_BINFILE_SIZE               1
#define TM_BINFILE_CURLOC             2
#define TM_FILECONVERT_SIZE           3
#define TM_FILECONVERT_CURLOC         4
#define TM_POPULATEREC_COUNT          5
#define TM_POPULATEREC_STEP           6
#define TM_ENDTRANSLATION             7
#define TM_FILEREPAIR_BEGIN           8
#define TM_FILEREPAIR_END             9
#define TM_FILEREPAIR_STEP            10

void SetNotifyWnd(HWND hWnd, int nMsg);
```

1. Call SetNotifyWnd(hWnd, nMsg) where hWnd specifies the HWND of the dialog window that will receive the notifications, and nMsg specifies the message to be sent. The following code demonstrates this call.

```
   // create a progress window
   CTranslatorProgDlg dlg;
   dlg.Create(NULL);
   SetNotifyWnd(dlg.GetSafeHwnd(), LDM_TRANSLATORNOTIFY);
```

2. Call StartSlmTranslation. During translation, LDM_TRANSLATORNOTIFY messages will be sent to the specified window. The following code demonstrates a message handler for this message:

```
LRESULT CTranslatorProgDlg::OnLDNotify(WPARAM wParam, LPARAM lParam)
{
   int nLower = 0;
   int nUpper = 0;

   switch (wParam)
   {
      case TM_BINFILE_SIZE:
         SetWindowText(CString((LPCTSTR)IDS_TRANSLATING_BINFILE));
         m_Progress1.SetRange32(0, (int)lParam);
         m_Progress1.SetPos(0);
         break;
      case TM_BINFILE_CURLOC:
         m_Progress1.SetPos((int)lParam);
         break;

      case TM_FILECONVERT_SIZE:
         SetWindowText(CString((LPCTSTR)IDS_CONVERTING_FILE));
         m_Progress1.SetRange32(0, (int)lParam);
         m_Progress1.SetPos(0);
         break;

      case TM_FILECONVERT_CURLOC:
         m_Progress1.SetPos((int)lParam);
         break;

      case TM_POPULATEREC_COUNT:
         SetWindowText(CString((LPCTSTR)IDS_POPULATING_REC));
         m_Progress1.SetRange32(0, (int)lParam);
         m_Progress1.SetPos(0);
         m_Progress1.SetStep(1);
         m_bHasRecs = true;
         break;

      case TM_POPULATEREC_STEP:
         if (m_bHasRecs)
            m_Progress1.StepIt();
         break;

      case TM_FILEREPAIR_BEGIN:
         SetWindowText(CString((LPCTSTR)IDS_REPAIRING_BINFILE));
         m_Progress1.SetRange32(0, (int)lParam);
         break;

      case TM_FILEREPAIR_STEP:
         m_Progress1.OffsetPos(1);
         break;

      case TM_FILEREPAIR_END:
         break;

      case TM_ENDTRANSLATION:
         m_Progress1.GetRange(nLower, nUpper);
         m_Progress1.SetPos(nUpper);
         break;
   }

   PumpMessages();

   return (LRESULT)(m_bCanceled ? 0 : 1);
}
```

```
When wParam == TM_BINFILE_SIZE, lParam will be the total size of the file.
When wParam == TM_BINFILE_CURLOC, lParam will be the current location.
When wParam == TM_FILECONVERT_SIZE, lParam will be size of file being converted
(sent when converting data files stored on thumbdrive into 'download' format).
When wParam == TM_FILECONVERT_CURLOC, lParam will be the current location (sent
when converting data files stored on thumbdrive into 'download' format).
When wParam == TM_POPULATEREC_COUNT, lParam will be number of records to extract.
wParam == TM_POPULATEREC_STEP indicates another record has been extracted.
wParam == TM_FILEREPAIR_BEGIN indicates a Sound Record repair is being attempted.
wParam == TM_FILEREPAIR_STEP indicates a Sound Record repair progress.
wParam == TM_FILEREPAIR_END indicates a Sound Record repair has finished.
wParam == TM_ENDTRANSLATION indicates translation has ended.
```

Interpretation of Session Log flags

Session Log type (or action) flags will be one of the following values:

```
#define PRM_ACTION_STOP                  0x0001  // Stop action performed
#define PRM_ACTION_RUN                   0x0002  // Run
#define PRM_ACTION_PAUSE                 0x0004  // Pause
#define PRM_ACTION_RESUME                0x0008  // Resume from Pause
#define PRM_ACTION_VOICE                 0x0010  // Voice Recording
#define PRM_ACTION_AUDIO                 0x0020  // Audio Recording {reserved for future}
#define PRM_ACTION_CAL                   0x0040  // Calibration check record
#define PRM_ACTION_RESET                 0x0080  // Reset {reserved for future}
#define PRM_ACTION_GPS_SYNC              0x0081  // GPS Time Sync
#define PRM_BACK_ERASE                   0x0082   // Back Erase Session Log
#define PRM_ACTION_MARKER                0x0083  // marker activated
#define PRM_ACTION_CALCHG                0x0084  // Calibration Change Performed
#define PRM_ACTION_PREAMPOFF             0x0085   // Preamp Removed
#define PRM_ACTION_PREAMPON              0x0086   // Preamp Connected
```

Session Log cause flags indicate the cause of the session log entry, and the meaning of the flag will be based on the type, or action, flag. Currently, cause flags can be one of the following values:

```
#define PRM_CAUSE_MASK                   0xFF00  // mask for CAUSE flags
#define PRM_CAUSE_KEY                    0x0100  // action caused by keyboard command
#define PRM_CAUSE_IO                     0x0200  // action caused by I/O command
#define PRM_CAUSE_TIMER                  0x0400  // action caused by the run or stable timer
#define PRM_CAUSE_POWER                  0x0800  // stop due to power failure
#define PRM_CAUSE_MEMORY                 0x1000  // stop due to out-of-memory
#define PRM_CAUSE_PREAMP_CONNECT         0x2000  // stop due to preamp connect
#define PRM_CAUSE_PREAMP_DISCONNECT      0x4000  // stop due to preamp disconnect
#define PRM_CAUSE_STABLE                 0x8000   // stop on STABLE
#define PRM_CAUSE_MARKER1                0x0100   // Marker 1 activated
#define PRM_CAUSE_MARKER2                0x0200   // Marker 2 activated
#define PRM_CAUSE_MARKER3                0x0300   // Marker 3 activated
#define PRM_CAUSE_MARKER4                0x0400   // Marker 4 activated
#define PRM_CAUSE_MARKER5                0x0500   // Marker 5 activated
#define PRM_CAUSE_MARKER6                0x0600   // Marker 6 activated
#define PRM_CAUSE_MARKER7                0x0700   // Marker 7 activated
#define PRM_CAUSE_MARKER8                0x0800   // Marker 8 activated
#define PRM_CAUSE_MARKER9                0x0900   // Marker 9 activated
#define PRM_CAUSE_MARKER10               0x0a00   // Marker 10 activated
#define PRM_CAUSE_GPS_POSITIVE           0x0100   // gps positive time correction
#define PRM_CAUSE_GPS_NEGATIVE           0x0200  // gps negative time correction
```

The interpretation of these flags is as follows:

If type (or action flag) is:	Then the cause flag will be on of the following:
PRM_ACTION_AUDIO	PRM_CAUSE_IO PRM_CAUSE_KEY PRM_CAUSE_TIMER
PRM_ACTION_MARKER	PRM_CAUSE_MARKER1 PRM_CAUSE_MARKER2 PRM_CAUSE_MARKER3 PRM_CAUSE_MARKER4 PRM_CAUSE_MARKER5 PRM_CAUSE_MARKER6 PRM_CAUSE_MARKER7 PRM_CAUSE_MARKER8 PRM_CAUSE_MARKER9 PRM_CAUSE_MARKER10

PRM_ACTION_GPS_SYNC	PRM_CAUSE_GPS_POSITIVE PRM_CAUSE_GPS_NEGATIVE

No further interpretation is needed for the remaining flags.

Notes

1. If you are using the SlmServer function GetData to download binary data from the instrument, instead of the newer DownloadBinaryData function, and wish to save that data in a format recognized by the SlmTranslator, please be aware of the following items:

   1. When writing the .bin file, please make sure the file is opened in binary mode to avoid automatic translation to CRLF pairs.
   2. When writing the .bin file, the first 4 bytes (int) must be the number of records in the file. Please set this to one. The next 4 bytes (int) must be the size of the file.

2. When translating Time History records, the m_nFlagField field will be a combination of the following values. These flags indicate if the sample has a marker (1-10) associated with it, if it was overloaded, or if the record is actually a keypress (session log) event (not data).

```
#define TIMEHIST_MARKER1      0x00000001
#define TIMEHIST_MARKER2      0x00000002
#define TIMEHIST_MARKER3      0x00000004
#define TIMEHIST_MARKER4      0x00000008
#define TIMEHIST_MARKER5      0x00000010
#define TIMEHIST_MARKER6      0x00000020
#define TIMEHIST_MARKER7      0x00000040
#define TIMEHIST_MARKER8      0x00000080
#define TIMEHIST_MARKER9      0x00000100
#define TIMEHIST_MARKER10     0x00000200
#define TIMEHIST_OVERLOAD     0x00000400
#define TIMEHIST_OBA_OVLD     0x00000800
#define TIMEHIST_EXCDED       0x00001000
#define TIMEHIST_PARTIAL      0x00002000
#define TIMEHIST_SR           0x00004000
#define TIMEHIST_BACKERASE    0x00008000
#define TIMEHIST_SESSION_LOG  0x80000000
```

3. When parsing Session Log records, the Count field refers to the current session log entry.