TIO32 DLL FOR COMMUNICATION WITH THE USB DAQ BOARD

Project: PGA Software
Author(s): L-PSE: Uwe Leinweber, Heiko Dobslaw

Contents

1 General

For the use of the Tio32 DLL it is necessary to know three basic functions:

1. an open function **tio32_Open**, which which creates an instance for communication with USB DAQ boards and opens the communication to the boards connected. It also monitors if boards are connected or disconnected to or from the PC automatically.
2. A run mnemomic command function **tio32_ RunMnemoDev** or **tio32_ RunMnemoAdr** for the interpretation and execution of mnemonic commands on the USB DAQ board and
3. a close function **tio32_Close** to end the work with the USB DAQ board.

All other functions in the DLL are additionally high level functions or functions for test and service purposes only.

The DLL interface is defined in the header file `tio32_Api.h`.
Basic defines are located in the header file `Tio32_defs.h.`
Examples for using these DLL functions can be found in the Test_Tio32Dll project.

2 General functions

2.1 *tio32_Open*

The **tio32_Open** function creates an object for communication with the USB DAQ boards. It tries to find USB DAQ boards connected to the PC. This is the first function to be processed.

```
int WINAPI tio32_Open(void);
```

Parameters

-

Return Values

0	If function succeeds
-100	`ERR_ALREADY_OPENED`: If function was already called, i.e. the communication is already open.

Remarks

Call this function before all other action with the USB DAQ boards.
An application can call this function only once, i.e. it is not possible to close the communication and later on to re-open the communication.

Example Code

```
   int ival = tio32_Open();
   printf("tio32_Open: = %d",ival);
```

2.2 *tio32_ RunMnemo*

The **tio32_ RunMnemo** function performs the conversion of an mnemonic command string to an Tx string. This string is supplied to the DAQ board addressed by its device address. From the Rx string received by that DAQ board the read and status values are extracted and returned.

```
int WINAPI tio32_RunMnemo(int adr,
                          unsigned char *pCmd,
                          int *dlen,
                          unsigned char *pdata,
                          unsigned char *pTx,
                          unsigned char *pRx,
                          int *reslen, unsigned char *pres,
                          int *acklen, unsigned char *pack );
```

Parameters

adr

[in] the USB device address of the DAQ board to be addressed.

pCmd

[in] pointer to the mnemonic command string to be performed.

dlen

[in, out] pointer to length of the pdata array.

On entry it should contain the maximal length of the pdata array. 256 is OK.

On return it contains the length of the returned data array. This is the number of returned characters, not the number of returned results.

pdata

[in, out] pointer to result string.

On entry this pointer should address an array of sufficient size to receive the results. 256 is the maximal possible length of the result.

On return this array contains the results of all read commands, measure commands and status requests (e.g. ACKS) in the order of occurence determined by the performed command that delivered the Rx string and the *details* structure.

These values are coded as plain decimal ASCII and separated by a space character.

pTx

[in, out] pointer to the Tx string resulting from the mnemonic command string.

On entry it should point to an char array of 32 bytes length.

On return it contains the converted command string as sent to the DAQ board.

pRx

[in] pointer to the Rx string.

On entry it should point to an char array of 32 bytes length.

On return it contains the received string as sent by the DAQ board.

reslen

[in, out] pointer to length of result array.

On entry it should contain the maximal length of the pres array, that is 32.

pres

[in, out] pointer to unsigned char result array.

On entry this pointer should address an array of sufficient size to receive the results. 32 is the maximal possible length of the data result.

On return this array contains the results of all read and measure commands in the order of occurrence, determined by the performed command, that delivered the Rx string .

acklen

[in, out] pointer to length of ack array.

On entry it should contain the maximal length of the ack array, that is 32.

pack

[in, out] pointer to unsigned char statusS array.
On entry this pointer should address an array of sufficient size to receive the statusS results. 32 is the maximal possible length of the data result.
On return this array contains the status values in the order of occurrence, determined by the performed command, that delivered the Rx string .

Return Values

0	If function converted the command string without error and delivered the results.
-1	General Error: unspecific or wrong address
-6	`ERR_TOO_LESS_ARGS`: Argument list of a command too short
-7	`ERR_TOO_MANY_ARGS`: Argument list of a command too long
-8	`ERR_INVAL_ARG`: Invalid argument, syntax error
-11	`ERR_NO_WRITE_HANDLE`: sending failed as the write handle was invalid
-12	`ERR_NO_READ_HANDLE`: receiving failed as the read handle was invalid
-13	`ERR_READ_TIMEOUT`: receiving failed with timeout error, i.e. nothing was received
-15	`ERR_READ_UNDEF`: receiving failed
-20	`ERR_WRITE_FAIL`: sending failed
-55	`ERR_GETRES`: error while analyzing the Rx array
-56	`ERR_DATLEN`: data string too short

Remarks

Parameters pTx and pRx are intended for debugging purposes.

Example Code

```
int adr = (int) mDevAdr;
CString cmdStr = "";
unsigned char Cmd[MAX_MNEMO_CHARS];
unsigned char Tx[TIO32_TXRX_BUF_LEN+1];
unsigned char Rx[TIO32_TXRX_BUF_LEN+1];
unsigned char resultStr[OUTBUFLEN];

mnemoCombox.GetWindowText(cmdStr);
int cnt;
for (cnt = 0; cnt < (MAX_MNEMO_CHARS); cnt++)
{
  if ((Cmd[cnt] = cmdStr.GetAt(cnt)) == '\0')
     break;
}
if (cnt == MAX_MNEMO_CHARS)
{
  Cmd[MAX_MNEMO_CHARS-1] = '\0';
}

// convert the command and communicate
int outlen = OUTBUFLEN;
int reslen = 32; int acklen = 32;
unsigned char res[32];
unsigned char ack[32];
int rval = tio32_RunMnemo(adr, &Cmd[0], &outlen, &resultStr[0],
                             &Tx[0], &Rx[0], &reslen, res, &acklen, ack);
printf("tio32_RunMnemoAdr=%d, adr:0x%x ...", rval, adr);
printf("... cmd:[%s]", Cmd);
if (OK_VAL == rval)
{
  printf(printBuf("... Tx :", Tx, TIO32_TXRX_BUF_LEN).GetBuffer() );
  printf(printBuf("... Rx :", Rx, TIO32_TXRX_BUF_LEN).GetBuffer() );
  printf("... res:[%s] reslen=%d", resultStr, outlen );
}
```

2.3 *tio32_Close*

The **tio32_Close** function closes the communication to all USB DAQ boards.

```
int WINAPI tio32_Close(void);
```

Parameters

-

Return Values

0	If function succeeds
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

Call this function only once in the lifetime of your application.
After calling this function it is not possible to re-open the communication anymore.

Example Code

```
     int i = tio32_Close();
    printf(1,"tio32_Close: =%d",i);
```

3 High level DLL functions

3.1 tio32_ReadDoubleFromEEPROM

The **tio32_ReadDoubleFromEEPROM** function reads a double value from the EEPROM.

```
int WINAPI tio32_ReadDoubleFromEEPROM(int adr,
                                      double *pVal,
                                      int memadr);
```

Parameters

adr
    [in] int, the USB board address.
pVal
    [out] ptr to double, destination pointer for the value red from EEPROM.
    Value range is (-9.99999; 9.99999).
memadr
    [in] int, the EEPROM address where to read the value.

Return Values

0	If function succeeds
-1	General error, unspecific
-8	ERR_INVAL_ARG: invalid argument e.g. address
-105	ERR_NOT_OPENED: communication was not opened before

Remarks

The double value is contained in three consecutive bytes in the EEPROM. The logical addres memadr times three is the physical starting address for that value in the EEPROM.

Example Code

```
int  boardAdr = 0x1234;
int  memAddr  = 10;
double val;
int i = tio32_ReadDoubleFromEEPROM(boardAddr, &val, memAddr);
printf("tio32_ReadDoubleFromEEPROM(): val=%f", val);
```

3.2 tio32_WriteDoubleToEEPROM

The **tio32_WriteDoubleToEEPROM** function writes a double value to the EEPROM.

```
int WINAPI tio32_WriteDoubleToEEPROM(int adr,
                                     double val,
                                     int memadr);
```

Parameters

adr
    [in] int, the USB board address.
val
    [in] double, value to write to EEPROM. Value range is (-9.99999; 9.99999).
memadr
    [in] int, the EEPROM address where to write the value.

Return Values

0	If function succeeds
-1	General error, unspecific
-8	`ERR_INVAL_ARG`: invalid argument e.g. address or value range
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

The double value is contained in three consecutive bytes in the EEPROM. The logical addres memadr times three is the physical starting address for that value in the EEPROM.

Example Code

```
int  adr = 0x1234;
int  memAddr  = 10;
double val = 1.01;
int i = tio32_WriteDoubleToEEPROM(adr, val, memAddr);
val = 0.0;
i = tio32_ReadDoubleFromEEPROM(adr, &val, memAddr);
printf("tio32_WriteDoubleToEEPROM(): val=%f written to %d", val, memAddr);
```

3.3 tio32_LoadCalibrationConstants

The **tio32_LoadCalibrationConstants** function loads all calibration constants from the EEPROM of the addressed USB board.

```
int WINAPI tio32_LoadCalibrationConstants(int adr);
```

Parameters

adr
[in] int, the USB board address.

Return Values

0	If function succeeds
-1	General error, unspecific
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

The calibration constants are held inside the DLL after read out. They are needed for the Read_Ads and SetDac functions.
Normally this function is called implicite by the FindTheHID function when a connected board was found.
So it need not to be called except that these calibration values in the boards EEPROM have changed e.g. by a calibration of that board itself.

Example Code

```
int  adr = 0x1234;
int i = tio32_LoadCalibrationConstants(adr);
```

3.4 *tio32_Read_ADS1100*

The **tio32_Read_ADS1100** function reads a ADC value from one of two ADS1100 and delivers a value in different forms depending on the used mode.

```
int WINAPI tio32_Read_ADS1100(int adr, int adsNo, int mode,
                              int *pCode, double *pVal,
                              double refUser);
```

Parameters

adr

[in] int, the USB board address.

adsNo

[in] int, number of ADS device, 1 or 2.

mode

[in] int, operating mode and reference selection

1: slow ADC mode, 5V reference
2: slow ADC mode, 3V reference
3: slow ADC mode, user reference provided by parameter refUser
4: slow ADC mode, code output
5: fast ADC mode, 5V reference
6: fast ADC mode, 3V reference

pVal

[out] double, the value measured by the ADS1100.
Output value is volts except mode 4, that delivers a code number.

refUser

[in] double, user reference in mode 3

Return Values

0	If function succeeds
-1	General error, unspecific
-8	`ERR_INVAL_ARG`: invalid argument
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

Calibration constants are used to correct the result in mode 1 and 2.
The user has to make sure to use the right mode according to the applied reference voltage to get a correct result.

Example Code

```
int  adr = 0x1234;
int adsNo = 1;
int mode = 1;
int code;
double value;
double uref = 0.0;
int i = tio32_Read_ADS1100(adr, adsNo, mode, &code, &value, uref);
printf("tio32_Read_ADS1100(): ADS%d delivers %dV in mode %d", adsNo, value, mode);
```

3.5 tio32_WriteDAC

The **tio32_WriteDAC** function writes a DAC value to one of the DAC channels. The value written depends on the used mode.

```
int WINAPI tio32_WriteDAC(int adr, int dacNo, int dacMode,
                          double val, double refUser);
```

Parameters

adr
[in] int, the USB board address.
dacNo
[in] int, number of ADS device, 1 or 2.
dacMode
[in] int, operating mode and reference selection
1: slow DAC mode, 5V reference, correction by ADS readback
2: slow DAC mode, 3V reference, correction by ADS readback
3: slow DAC mode, user reference provided by parameter refUser, correction by ADS readback
4: slow DAC mode, code output
5: fast DAC mode, 5V reference
6: fast DAC mode, 3Vreference
val
[in] double, value to be written to the DAC. Value is in volts except in mode 4 where it is a code number
refUser
[in] double, user reference in mode 3

Return Values

0	If function succeeds
-1	General error, unspecific
-8	`ERR_INVAL_ARG:` invalid argument
-62	`ERR_DAC_DEVIATION:` dac output did not settle to the required value with the required accuracy of +/-0.0002V.
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

The inherent nonlinearity of the DAC is significantly improved in the modes 1 to 3. This is accomplished by reading back the DAC output voltage with an ADS, calculating the deviation from the required value and correcting the DAC value by this deviation. This correction is done in an so called auto cal loop in up to 4 steps. But this slows down the DAC output. The remaining modes only put out a single DAC value.

At values at the output range limits (0V, Vref) the DAC may not be able to settle at the required value. This results in error -62 when working in mode 1 to 3.

To prevent a runaway of the autocal loop in case the DAC output is errornously connected to an external voltage source, so that the output value is overwritten, the maximal allowed deviation value is limited to 0.2V. If deviation exceeds this value the autocal loop terminates and error -62 is returned.

The user has to make sure to use the right mode according to the applied reference voltage to get a correct result.

Example Code

```
int  adr= 0x1234;
int dacNo = 1;
int mode = 1;
int code;
double value = 1.23;
double uref = 0.0;
int i = tio32_RWriteDAC(adr, dacNo, mode, value, uref);
printf("tio32_Read_ADS1100(): DAC%d should deliver %dV in mode %d", dacNo, value, mode);
```

3.6 *tio32_SetCurrentVoltageMode*

The **tio32_SetCurrentVoltageMode** function connects the INA output to the ADS input for loop current measurement or connects the voltage inputs of the board to the ADS inputs depending on selected mode.

```
int WINAPI tio32_SetCurrentVoltageMode(int adr, int mode);
```

Parameters

adr

[in] int, the USB board address.

mode

[in] int, operating mode

1: current mode, INA connected

2: voltage mode, voltage inputs connected

Return Values

0	If function succeeds
-1	General error, unspecific
-8	`ERR_INVAL_ARG`: invalid argument
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

The correct mode need to be set before measuring any loop current or voltage.

Example Code

```
int  adr= 0x1234;
int mode = 1;
int i = tio32_SetCurrentVoltageMode(adr, mode);
printf("tio32_SetCurrentVoltageMode(): Current mode set");
```

3.7 *tio32_ReadILoop*

The **tio32_ReadILoop** function reads the loop current of the current loop.

```
int WINAPI tio32_ReadILoop(int adr, int *code, double *pVal);
```

Parameters

adr
    [in] int, the USB board address.
code
    [out] ptr to int, destination pointer for saving the ADS code value.
pVal
    [out] ptr to double, destination pointer for saving the result value. Result is in milliamperes.

Return Values

0	If function succeeds
-1	General error, unspecific
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

The current mode need to be set before measuring any loop current.
The A/D-D/A reference need to be set to 5V for this function to work properly.

Example Code

```
int  adr = 0x1234;
int mode = 1;
int code;
double Iloop;
int i = tio32_SetCurrentVoltageMode(adr, mode);  // set current mode
I = tio32_ReadILoop(adr, &code, &Iloop);         // read the loop current
printf("tio32_ReadILoop():Loop current is %fmA", Iloop);
```

3.8 tio32_AdsMux

The **tio32_AdsMux** function sets the connection of the ADS inputs.

```
int WINAPI tio32_AdsMux(int adr, int muxNo, int in_p, int in_n);
```

Parameters

adr
    [in] int, the USB board address.
muxNo
    [in] int, mux number, 1 or 2, dedicated to ADS1 or ADS2.

in_p
    [in] int, connection mode of the positive ADS input
        1: open
        2: GND
        3: DAC
        4: no change
pVal
    [in] int, connection mode of the negative ADS input
        1: open
        2: GND
        3: DAC
        4: no change

Return Values

0	If function succeeds
-1	General error, unspecific
-8	`ERR_INVAL_ARG`: invalid argument
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

What DAC channel will be connected on connection mode 3
depends on the MUX number and the selected input:

muxNo = 1: in_p = 3: DAC_B to positive ADS1 input
muxNo = 1: in_n = 3: DAC_A to negative ADS1 input
muxNo = 2: in_p = 3: DAC_D to positive ADS2 input
muxNo = 2: in_n = 3: DAC_C to negative ADS2 input

Example Code

```
int  adr = 0x1234;
mux  = 1;     // MUX1
con_p = 3;    // DAC
con_n = 2;    // GND
int ads = 1; int mode = 1; int code;
double volts;
int i = tio32_AdsMux(adr, mux, con_p, con_n);          // connect DAC to ADS1
I = tio32_ReadADS1100(adr, ads, mode, &code, &volts);  // read the DAC output
printf("tio32_AdsMux():DAC_B connected to ADS1 delivers %fV", volts);
```

4 Further helper and debug functions

These following functions are auxiliary functions for debugging and test purposes.

4.1 tio32_TestBoardAvail

The **tio32_ TestBoardAvail** function tests if a USB DAQ board is connected to the PC under a specific device address and returns its device number.

```
int WINAPI tio32_TestBoardAvail(int adr,
                                int *pDevNo);
```

Parameters

adr

[in] address of the USB DAQ board to communicate with

**pDevNo*

[out] device number of the USB DAQ board found at address adr

Return Values

0	If function succeeds and there is a USB DAQ board connected to the PC with this address
-1	Error: No USB DAQ board found with this address
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

Example Code

```
int devNo = 0;
int adr = 0x92F0;
int i = tio32_TestBoardAvail(adr, &devNo);
if (i == 0)
{
   printf("tio32_TestBoardAvail: Board at address 0x%X is #%d", adr, devNo);
}
else
{
   printf("tio32_ TestBoardAvail: no board found at address 0x%X", adr);
}
```

4.2 tio32_SendReceive32

The **tio32_SendReceive32** function transmitts a 32 bytes long character (8 bit number) array to the USB DAQ board. Then it waits until it receives a 32 bytes long answer or a time of 6 seconds has elapsed.
It also checks if the last byte in the Rx-array is the incremented value of the last byte in the Tx-array. This incrementation is done by the DAQ board to signal that the command was performed successfully. If the last byte is not incremented this is considered an error.

```
int WINAPI tio32_SendReceive32(int adr,
                               unsigned char *pTx,
                               unsigned char *pRx);
```

Parameters

adr
[in] address of the USB DAQ board to communicate with
pTx
[in] pointer to the bytes to be transmitted
pRx
[in] pointer to the array for the bytes to be received

Return Values

0	If function succeeds
-105	`ERR_NOT_OPENED`: communication was not opened before
-11	`ERR_NO_WRITE_HANDLE`: sending failed as the write handle was invalid
-12	`ERR_NO_READ_HANDLE`: receiving failed as the read handle was invalid
-13	`ERR_READ_TIMEOUT`: receiving failed with timeout error, i.e. nothing was received
-15	`ERR_READ_UNDEF`: receiving failed
-20	`ERR_WRITE_FAIL`: sending failed
-1	Error: all other errors

Remarks

The last byte is incemented by the USB DAQ board, it can be considered as a counter.

Example Code

```
   int adr = (int) mDevAdr;
   ZeroMemory(mRx,sizeof(mRx));
   CString csAction = "";
   // toggle the lamp
   if (mTx[1] == 1)
   {
      mTx[1] = 2;
      csAction = " lamp off ";
   }
   else
   {
      mTx[1] = 1;
      csAction = " lamp on ";
   }

   int i = tio32_SendReceive32(adr, _T(mTx), _T(mRx));

   if (i == OK_VAL)
   {
      printf(printBuf(" tx",mTx,TIO32_TXRX_BUF_LEN).GetBuffer ());
      printf(printBuf(" rx",mRx,TIO32_TXRX_BUF_LEN).GetBuffer ());
   }
   printf("  board %d %s =%d",dev, csAction.GetBuffer (), i);
   // take the counted value into the next tx msg
   mTx[31] = mRx[31];

CString CTest_Tio32DllDlg::printBuf(char *pName, unsigned char *pBuf, int len)
{
   CString res = pName;
   res += "[";
   for (int i=0; i<len; i++)
   {
      res.AppendFormat("%02x ",pBuf[i]);
   }
   res.TrimRight();
   res += "]";
   return res;
}
```

4.3 *tio32_GetBoardCount*

The **tio32_GetBoardCount** function provides the number of USB DAQ boards actually connected to the PC.

```
int WINAPI tio32_GetBoardCount(int *pNumberOfBoards, int *pAddr);
```

Parameters

pNumberOfBoards
 [out] destination pointer to number of USB DAQ boards actually connected to the PC

pAddr
 [out] destination pointer to list of USB addresses of the USB DAQ boards actually connected to the PC

Return Values

0	If function succeeds
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

None

Example Code

```
    int boards;
    int adr[4];
   int i = tio32_GetBoardCount(&boards, &adr[0]);
   printf("tio32_GetBoardCount: =%d; %d board(s) connected at addresses: ",i,boards);
   int cnt;
   for{cnt = 0; cnt < boards; cnt++)
   {
      printf("0x%X ", adr[cnt]);
   }
   printf("\n");
```

4.4 *tio32_GetCtrlPort*

The **tio32_GetCtrlPort** function provides the contents of the USB DAQ board control port. This values is not readable from the board, so its value is stored by the DLL in a shadow variable.

```
int WINAPI tio32_GetCrtlPort(int adr, unsigned char *ctrl);
```

Parameters

adr
 [in] the USB DAQ board address

ctrl
 [out] destination pointer receive the control port value

Return Values

0	If function succeeds
-1	if the the board is not connected
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

None

Example Code

```
 int board = 0x2F90;
unsigned char crtl;
int i = tio32_GetCrtlPort(board, &crtl);
printf("tio32_GetCrtlPort: =%d; control port of board %x = %x ",i,board,ctrl);
```

4.5 tio32_SetLogLevel

The **tio32_SetLogLevel** function sets the log level of the communication. This log level determines the amount of data being written into the log file.

```
int WINAPI tio32_SetLogLevel(int lvl);
```

Parameters

lvl

[in] New log level value

Return Values

0	If function succeeds
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

The log file name is `Tio32.log`. This file is stored in the application directory of the software calling the DLL. The maximum log file size is 1MB. If the log file exceeds this size, the file is copied into `Tio32.bak` and a new log file is created.
A smaller number always means less log messages and inceases execution speed. A greater log level contains all log messages of smaller log levels.
The following log levels are defined yet:

`0`	`LOGLEVL_ONLYSTARTSTOP`	Only start and stop of the DLL are logged
`1`	`LOGLEVL_NORMAL`	All messages of log level 0 plus all function call and results are logged.
`2`	`LOGLEVL_FINHIDS`	All messages of log level 1 plus details of the findHID procedure are logged.
`3`	`LOGLEVL_TESTTXRX`	All messages of log level 2 plus the input and output of the tio_SendReceive32 function are logged.
`10`	`LOGLEVL_MAX`	Maximum number, all log messges are written into the log file.

Example Code

```
    int i = tio32_SetLogLevel((int) mLogLvl);
   printf("tio32_SetLogLevel(%d): =%d",mLogLvl,i);
```

4.6 tio32_ShowWnd

The **tio32_ShowWnd** function shows the DLL window.

```
int WINAPI tio32_ShowWnd(void);
```

Parameters

-

Return Values

0	If function succeeds
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

The DLL creates a window which is normally hidden. It is not necessary for the communication to show the window. The DLL window was intended only for debugging purposes during development of the DLL. It may not be necessary to give external users access to this function.

Example Code

```
 int i = tio32_ShowWnd();
printf("tio32_ShowWnd: =%d",i);
```

4.7 tio32_HideWnd

The **tio32_HideWnd** function hides the DLL window.

```
int WINAPI tio32_HideWnd(void);
```

Parameters

-

Return Values

0	If function succeeds
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

See **tio32_ShowWnd** function.

Example Code

```
    int i = tio32_HideWnd();
   theApp.logPrintf("tio32_HideWnd: =%d",i);
```

4.8 *tio32_ShowDbgWnd*

The **tio32_ShowDbgWnd** function shows the DLL debug window.

```
int WINAPI tio32_ShowDbgWnd(void);
```

Parameters

-

Return Values

0	If function succeeds
-105	`ERR_NOT_OPENED`: communication was not opened before

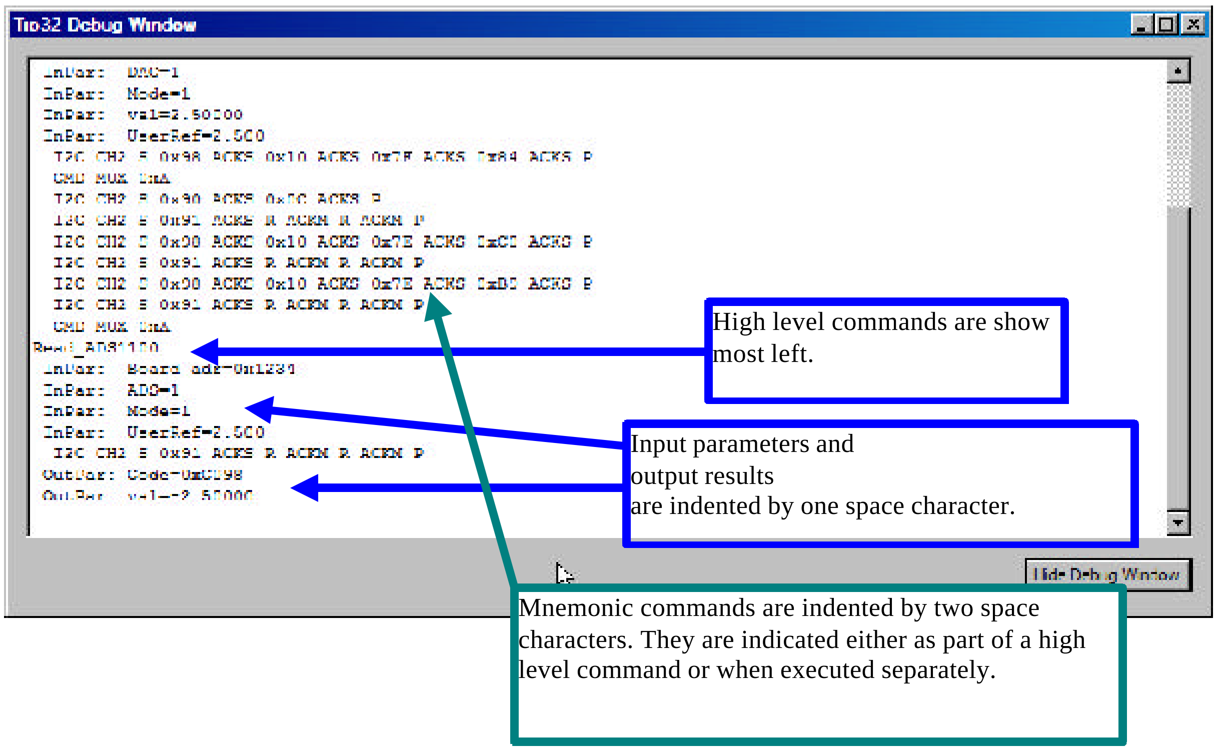
Remarks

This debug window shows the entry values and results of the higher level functions. It also shows the data transmitted to the DAQ board in the 32 Byte Tx-string.

Example Code

```
 int i = tio32_ShowDbgWnd();
printf("tio32_ShowDbgWnd: =%d",i);
```

The debug window shows high level commands with their input parameters and results as well as mnomonic commands:



```
 InPar:  DAC=1
 InPar:  Mode=1
 InPar:  val=2.50000
 InPar:  UserRef=2.500
  I2C CH2 S 0x98 ACKS 0x10 ACKS 0x7F ACKS 0x84 ACKS P
  CMD MUX 0xA
  I2C CH2 S 0x90 ACKS 0x0C ACKS P
  I2C CH2 S 0x91 ACKS R ACKM R ACKM P
  I2C CH2 S 0x90 ACKS 0x10 ACKS 0x7E ACKS 0xC0 ACKS P
  I2C CH2 S 0x91 ACKS R ACKM R ACKM P
  I2C CH2 S 0x90 ACKS 0x10 ACKS 0x7E ACKS 0xD0 ACKS P
  I2C CH2 S 0x91 ACKS R ACKM R ACKM P
  CMD MUX 0xA
Read_ADS1110
 InPar:  Board adr=0x1231
 InPar:  ADS=1
 InPar:  Mode=1
 InPar:  UserRef=2.500
  I2C CH2 S 0x91 ACKS R ACKM R ACKM P
 OutPar: Code=0xC098
 OutPar: val=-2.50000
```

Tio32 Debug Window
Hide Debug Window
High level commands are show most left.
Input parameters and output results are indented by one space character.
Mnemonic commands are indented by two space characters. They are indicated either as part of a high level command or when executed separately.

It is possible to select and mark pices of text and copy it into other applications:

4.9 tio32_HideDbgWnd

The **tio32_HideDbgWnd** function hides the DLL debug window.

```
int WINAPI tio32_HideDbgWnd(void);
```

Parameters

-

Return Values

0	If function succeeds
-105	`ERR_NOT_OPENED`: communication was not opened before

Remarks

See **tio32_ShowDbgWnd** function.

Example Code

```
    int i = tio32_HideDbgWnd();
   theApp.logPrintf("tio32_HideDbgWnd: =%d",i);
```

4.10 tio32_GetTimeStamp

The **tio32_GetTimeStamp** function provides a NULL-terminated string containing information about the time the DLL was compiled.

```
int WINAPI tio32_GetTimeStamp(char *pc, int len);
```

Parameters

pc
: [in,out] pointer to a charater array for the string the function shall deliver. After return from the function the array contains the string.

len
: [in] length of the array .

Return Values

0	always

Remarks

The timestamp response looks like this:
`Mon Jul 23 14:27:53 2007`
The length of the character buffer should be 50.
If the buffer is smaller than the string length the string will be truncated.

Example Code

```
#define CF_LEN  50

void CTest_Tio32DllDlg::OnBnClickedButtonTimestamp()
{
   char cf[CF_LEN];
   int i = tio32_GetTimeStamp(&cf[0], CF_LEN);
   theApp.logPrintf(1,"tio32_GetTimeStamp: DLL timestamp: [%s] =%d",cf,i);
}
```

4.11 tio32_ ConvertCmd

The **tio32_ ConvertCmd** function converts a mnemonic command string to a Tx string.

```
int WINAPI tio32_ConvertCmd(unsigned char *org,
                            unsigned char *result,
                            TCmdResultStruct *details);
```

Parameters

org

[in] the zero terminated mnemonic command string

result

[in,out] pointer to a charater array for the string the function shall deliver. After return from the function the array contains the converted command string. This is the 32 bytes Tx string for the DAQ board.

details

[in, out] pointer to a `TCmdResultStruct` structure, the function shall deliver. After return from the function the structure contains properties of the converted command.

```
// command string properties struct
// a summary of the properties of the mnemonic command string
typedef struct
{
  char cmdProps[PROPS_LEN];    // the token string list of the mnemonic command string
  int numTokens;               // number of mnemonic tokens in the command string
  int numCmds;                 // number of commands in the command string
  int numReads;                // number of read commands, i.e. no. of read vars needed
  int numStatus;               // number of status requests, i.e. no. of status vars needed
 int errorPos;                 // error position, token no. that caused the error
} TCmdResultStruct;
```

The `cmdProps` string consists of semicolon separated entries for each mnemonic in the command. Each entry consists of space separated property items coded as plain decimal ASCII taken from the following structure:

```
typedef struct
{
  char cmdToken[MAX_TOKEN_LEN]; // the command token string
  TECmdTokenType tokenType;     // the token type, see TECmdTokenType
  int pos;                      // position in the command word list
  int paramCnt;                 // if a command, its parameter count else 0
  int cmdPos;                   // if a parameter the cmd it belongs to else == pos
  int paramNo;                  // if a parameter the parameter number else 0
  int resultPos;                // position of a read or status cmd read in the Rx-array;
                                // -1 if none, >= 0 is position
} TCmdPropStruct;
```

```
// bit flags for command string token type classification
// used for command interpretation and syntax highligting
typedef enum
{
       NONE  = 0,
       TRANS = 0x0001,          // is transaction type
       CMD   = 0x0002,          // is command
       PARAM = 0x0004,          // is parameter
       NUM   = 0x0008,          // is numeric
       READ  = 0x0010,          // is read request (requires a variable)
       STAT  = 0x0020           // is status request (requires a variable)
}TECmdTokenType;
```

Example mnemonic command:
SPI CH1 BREH L R R 0x22 0x56 H
The result string is:
01 01 01 06 E4 00 00 01 FF FF 22 56 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
The resulting `cmdProps` string is:
SPI 1 0 2 0 0 -1;CH1 4 1 0 0 0 -1;BREH 4 2 0 0 1 -1;L 2 3 0 3 0 -1;R 18 4 0 4 0 1;R 18 5 0 5 0 2;0X22 10 6 0 6 0 -1;0X56 10 7 0 7 0 -1;H 2 8 0 8 0 -1;
(The resulting Rx string could be (not handled by this function):
00 0D 37 91 2B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01)

Return Values

0	If function converted the command string without error and delivered the results.
-1	General Error: unspecific
-6	`ERR_TOO_LESS_ARGS`: Argument list of a command too short
-7	`ERR_TOO_MANY_ARGS`: Argument list of a command too long
-8	`ERR_INVAL_ARG`: Invalid argument, syntax error

Remarks

For the LabVIEW Environment use function `tio32_ConvertCmdLv()`instead because the DLL-interface of LabVIEW cannot handle structures as parameters.

Example Code

```
// convert the command
TCmdResultStruct details;
int rval = tio32_ConvertCmd(&Cmd[0], &Tx[0], &details);
printf("tio32_ConvertCmd=%d ...", rval);
printf("... cmd:[%s]", Cmd);
if (rval == OK_VAL)
{
    printf(printBuf("... Tx:", Tx, TIO32_TXRX_BUF_LEN).GetBuffer() );
}
printf("... #tokens:%d, #cmds:%d, #reads:%d,
        #status:%d, errPos:%d", details.numTokens, details.numCmds, details.numReads,
        details.numStatus, details.errorPos);
```

4.12 tio32_ ConvertCmdLv

The **tio32_ ConvertCmdLv** function converts a mnemonic command string to a Tx string.

```
int WINAPI tio32_ConvertCmdLv(unsigned char *org,
                                            unsigned char *result,
                                            int lenDetails,
                                            int *details,
                                            int lencmdProps,
                                            unsigned char *cmdProps);
```

Parameters

org

[in] the zero terminated mnemonic command string

result

[in,out] pointer to a charater array for the string the function shall deliver. After return from the function the array contains the converted command string. This is the 32 bytes Tx string for the DAQ board.

lenDetails

[in, out] pointer to length of the details int array.

details

[in, out] pointer to an int array, the function shall deliver. After return from the function the array contains properties of the converted command in the following order:

details[0]: Number of command string tokens

details[1]: Number of commands

details[2]: Number of reads

details[3]: Number of status requests

details[4]: Error position (-1: no error; >= 0 : token number that caused the error)

lencmdProps

[in, out] pointer to length of the cmdProps string.

The `cmdProps` string consists of semicolon separated entries for each mnemonic in the command.
Each entry consists of space separated property items coded as plain decimal ASCII taken from the following structure:

```
typedef struct
{
char cmdToken[MAX_TOKEN_LEN]; // the command token string
TECmdTokenType tokenType;     // the token type, see TECmdTokenType
int pos;                      // position in the command word list
int paramCnt;                 // if a command, its parameter count else 0
int cmdPos;                   // if a parameter the cmd it belongs to else == pos
int paramNo;                  // if a parameter the parameter number else 0
int resultPos;                // position of a read or status cmd read in the Rx-array;
                              // -1 if none, >= 0 is position
} TCmdPropStruct;


// bit flags for command string token type classification
// used for command interpretation
typedef enum
{
       NONE  = 0,
       TRANS = 0x0001,         // is transaction type
       CMD   = 0x0002,         // is command
       PARAM = 0x0004,         // is parameter
       NUM   = 0x0008,         // is numeric
       READ  = 0x0010,         // is read request (requires a variable)
       STAT  = 0x0020          // is status request (requires a variable)
}TECmdTokenType;
```

Example mnemonic command:

```
SPI CH1 BREH L R R 0x22 0x56 H
```

The result string is:

```
 01 01 01 06 E4 00 00 01 FF FF 22 56 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The resulting `cmdProps` string is:

```
SPI 1  0 2 0 0 -1;CH1 4  1 0 0 0 -1;BREH 4  2 0 0 1 -1;L 2  3 0 3 0 -1;R 18  4 0 4 0 1;R 18  5 0 5 0 2;0X22 10  6 0 6 0 -1;0X56 10  7 0 7 0 -1;H
2  8 0 8 0 -1;
```

(The resulting Rx string could be (not handled by this function):

```
   00 0D 37 91 2B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01)
```

Return Values

0	If function converted the command string without error and delivered the results.
-1	General Error: unspecific
-6	`ERR_TOO_LESS_ARGS`: Argument list of a command too short
-7	`ERR_TOO_MANY_ARGS`: Argument list of a command too long
-8	`ERR_INVAL_ARG`: Invalid argument, syntax error

Remarks

This function is the LabVIEW adapted version of function `tio32_ConvertCmd()` with different parameter handling (no structures).

Example Code

See LabVIEW virtual instrument **tio32_ConvertCmdLv.vi** for usage.

4.13 tio32_ GetResults

The **tio32_ GetResult** function extracts read results and status results from an Rx string resulting from an previous performed command.

```
int WINAPI tio32_GetResults(unsigned char *prx,
                             TCmdResultStruct *details,
                             int *dlen,
                             unsigned char *pdata
                              int *reslen, unsigned char *pres,
                              int *acklen, unsigned char *pack);
```

Parameters

prx
[in] pointer to the Rx string

details
[in, out] pointer to a `TCmdResultStruct` structure as delivered by function `tio32_ConvertCmd()`.
This structure is used to identify and locate the results in the Rx string.

dlen
[in, out] pointer to length of the pdata array.
On entry it should contain the maximal length of the pdata array. 256 is OK.
On return it contains the length of the returned data array. This is the number of returned characters, not the number of returned results.

pdata
[in, out] pointer to result string.
On entry this pointer should address an array of sufficient size to receive the results. 256 is the maximal possible length of the data result.
On return this array contains the results of all read commands, measure commands and status requests (e.g. ACKS) in the order of occurence determined by the performed command that delivered the Rx string and the *details* structure.
These values are coded as plain decimal ASCII and separated by a space character.

reslen
[in, out] pointer to length of result array.
On entry it should contain the maximal length of the pres array, that is 32.

pres
[in, out] pointer to unsigned char result array.
On entry this pointer should address an array of sufficient size to receive the results. 32 is the maximal possible length of the data result.
On return this array contains the results of all read and measure commands in the order of occurrence, determined by the performed command, that delivered the Rx string .

acklen
[in, out] pointer to length of ack array.
On entry it should contain the maximal length of the pres array, that is 32.

pack
[in, out] pointer to unsigned char ACKS array.
On entry this pointer should address an array of sufficient size to receive the ACKS results. 32 is the maximal possible length of the data result.
On return this array contains the ACKS status values in the order of occurrence, determined by the performed command, that delivered the Rx string .

Return Values

0	If function converted the command string without error and delivered the results.
-1	General Error: unspecific
-55	`ERR_GETRES`: error while analyzing the Rx array
-56	`ERR_DATLEN`: data string too short

Remarks

This function is intended for use by the command list editor to extract and display the command results.

Example Code

```
unsigned char prx[TIO32_TXRX_BUF_LEN];
unsigned char Cmd[] = "I2C CH1 S 0x80 ACKS 0x03 ACKS 0x81 ACKS R ACKM R ACKM P";
unsigned char Tx[TIO32_TXRX_BUF_LEN];

// first: convert the command
  TCmdResultStruct details;
  int reslen = 32; int acklen = 32;
  unsigned char res[32];
  unsigned char ack[32];
  int rval = tio32_ConvertCmd(&Cmd[0], &Tx[0], &details, &reslen, res, &acklen, ack);
  // logging
  theApp.logPrintf(LOGLEVL_ALL, "tio32_ConvertCmd=%d ...", rval);
  theApp.logPrintf(LOGLEVL_ALL, "... cmd:[%s]", Cmd);
  if (rval == OK_VAL)
  {
    printf(printBuf("... Tx:", Tx, TIO32_TXRX_BUF_LEN).GetBuffer() );
    printf("... #tokens:%d, #cmds:%d, #reads:%d, #status:%d, errPos:%d",
           details.numTokens, details.numCmds, details.numReads,
           details.numStatus, details.errorPos);

    int dlen = SLEN;
    unsigned char pdata[SLEN];
    acklen = reslen = 32;
    rval = tio32_GetResults(prx,&details,&dlen,pdata, &reslen, res, &acklen, ack);
    printf("tio32_GetResults=%d ...", rval);
    printf("... pdata:[%s]", pdata);
  }
```

5 DLL functions implemented in LabVIEW

Most of the listed DLL functions where implemented in LabVIEW as Virtual Instruments (VI). So these DLL functions can be easily accessed in the LabVIEW environment.
Any such VI contains a single DLL function and has the same name as its contained DLL function.

The used LabVIEW version is V8.5.

The following list shows all DLL functions implemented as VI.

tio32_AdsMux.vi
tio32_Close.vi
tio32_ConvertCmdLv.vi
tio32_FindTheHID.vi
tio32_GetBoardCount.vi
tio32_GetTimeStamp.vi
tio32_HideDbgWnd.vi
tio32_HideWnd.vi
tio32_LoadCalibrationConstants.vi
tio32_Open.vi
tio32_ReadDoubleFromEEPROM.vi
tio32_ReadILoop.vi
tio32_Read_ADS1100.vi
tio32_RunMnemo.vi
tio32_SendReceive32.vi
tio32_SetCurrentVoltageMode.vi
tio32_SetLogLevel.vi
tio32_ShowDbgWnd.vi
tio32_ShowWnd.vi
tio32_TestBoardAvail.vi
tio32_WriteDac.vi
tio32_WriteDoubleToEEPROM.vi

The `ErrorCombiner.vi` is an additional VI used in any DLL-VI to combine the DLL-functions return value with the LabVIEW error-cluster for what any VI has an in- and output. So function error return values can be handled easily by LabVIEW.

The `Test_tio32_DLL.vi` is a test instrument for testing all DLL-VIs. It is described in the file Tio32Dll_Softwaretest.doc.