TurbulentFlow Instrumentation

Reference
TFI Device Control v3.7
Software Interface

Last modified 2 September 2016

Overview

The *TFI Device Control* Software Interface provides access to functions required to process DPMS, Probe and other device data. This enables end users to incorporate the functionality of the *Device Control* software into their own software. The software interface also provides a method of processing device data acquired using unsupported data acquisition systems.

Contents

Using the Interface

The interface is supplied as a Windows dynamic link library (DLL) that can be called from any software capable of calling DLLs, including C++ and most other programming languages, LabVIEW® and MATLAB®. To simplify implementation, the interface replies on the *Device Control* software for many functions. Although this requires the *Device Control* software to be installed, the end user is not required to perform a complex configuration procedure – only a few function calls are required to process data. Linking the interface to the *Device Control* software also allows end users to features such as the real-time display form – only one function call is required to view DPMS and Probe data in real time if a supported A/D card is available!

Linking

Information about linking to the interface from various other applications is provided below. Please contact TFI for assistance and for further information about linking from other applications.

C and C++

For end-user programs written with C or C++, the supplied DLL run-time dynamic loading code ('pmDevCtrlInterfaceLink') can be utilised – just include the source and header files and compile with your application. After a successful call to the `dcilLoadInterface` function, all functions in the interface will be available for use.

Other Languages

The interface can be used from any other programming language that supports Windows DLLs. Refer to the relevant documentation for the language for further information and use the function prototypes provided in the reference as a guide. Methods of linking include creating an import library from the DLL to be used for static linking and writing a custom interface link for performing run-time dynamic linking (similar to the interface link provided for C++).

LabVIEW®

For LabVIEW users, a sample VI that demonstrates calling of the interface functions is available. Other functions from the interface can be called using the same procedures used in the example. Further details are available in the documentation supplied with LabVIEW (search for 'shared libraries').

MATLAB®

The interface can be directly accessed when using MATLAB version 6.5.1 and newer (version 6.5.0 users can obtain a free download from the MathWorks website to provide access to the interface). See the 'Language Notes' section for important information about using MATLAB.

- Use MATLAB's `LoadLibrary` function to load the interface
  - `LoadLibrary('pmDevCtrlInterface','pmDevCtrlInterfaceMain.h');`

- Alternatively, use the extended form of `LoadLibrary` to provide an alias to the interface ('dci' in this example), which then allows a shorter name to be used with MATLAB's library functions
  - `LoadLibrary('pmDevCtrlInterface','pmDevCtrlInterfaceMain.h','alias','dci');`
- Use MATLAB's `CallLib` function to call the interface functions (such as setting the temperature to 22.5°C in the following example). Note that this example uses the alias 'dci' described above
  - `result = CallLib('dci','dciSetTemperature',22.5);`
- Interface functions taking array pointers as arguments will return the result on the left-hand side of the MATLAB function call. For example, use the following code to retrieve the current output file name (note that memory for the character array must be allocated before calling the `dciGetOutputFileName` interface function)
  - `name = setstr( zeros(1,256) );`
  - `[result,name] = CallLib('dci','dciGetOutputFileName,name,length(name));`

Block Processing

All data used by the *Device Control* software are handled in 'blocks'. This enables high-speed data sampling and near real-time processing and data display. Blocks are typically between 0.3s and 1.0s in length. Importantly, there are no gaps between blocks, i.e. the time record is continuous. To perform post-processing, it is important to understand how data are processed in blocks.

Block Length

The length of blocks used for data acquisition and processing is mostly dependent upon the desired display update rate when monitoring data. For post-processing, the block length chosen is almost irrelevant, although the block length (measured in time) should be shorter than any events of interest. For example, if a Cobra Probe is being used to measure turbulence, the block length should be less than the time between significant changes of the turbulence characteristics, such as moving from a low- to a high-turbulence region. Block lengths in the range 0.3s to 1.0s are normally a good choice.

Passing and Reading Block Data

After configuring the interface for processing, raw voltage data need to be passed for processing. If required, the processed data can be read from the interface, or the interface can output the results to screen or disk. The diagram below shows a timeline of how raw data are passed and how processed data are read. The most notable point is that processed data lag the raw data by one block, i.e. the first block of processed data is read after the second block of raw data has been passed, the second block of processed data is read after the third block of raw data has been passed and so on.

Table 1 - Block processing timeline for three data blocks

Function	Time > > > > > >									
Pass raw data	Block 1		Block 2			Block 3				
Process data		Block 1		Block 2			Block 3			
Read processed data					Block 1			Block 2		Block 3
Complete processing									Done	

The processed data lag the raw data by one block due to the algorithm used to minimise spectral 'windowing' effects introduced during processing; two blocks of data are required to almost completely eliminate the windowing effects. Prior to version 3.2 of the *Device Control* software and interface, this 'block lag' meant that one extra block of raw data had to be passed. For example, if 3072 raw data samples were passed for processing in blocks of 1024, only 2048 processed data samples would be returned. If 3072 processed samples were required, an extra block of 1024 samples needed to be passed, i.e. a total of 4096 raw data samples. As of version 3.2, this has changed – the number of returned processed samples is now the same as the number of passed raw samples, although there is still a lag of one block between raw and processed data blocks (the final block of processed data is returned after calling the `dciDoneProcessing` function).

Post-Processing Procedure

The following is an overview of the typical procedure for post processing data using the interface. (Note: Many of the steps listed may not be required if they have already been configured using the *Device Control* software or if previously set via the interface; functions only need be called to changed required values, otherwise the existing value will be used.)

- Use the *Device Control* software to configure the devices that were used during data acquisition
    - Other settings, such as acquisition rate and output file selection, can also be performed in the *Device Control* software but are not necessary. The devices can also be 'zeroed' using the *Device Control* software.
- Load the interface DLL and initialise, specifying if the data acquisition driver should be loaded
    - Call the `dciInit` to initialise the interface
- Ensure the interface DLL is correctly loaded and initialised
    - Call the `dciInitialised` function to check
- Configure the active devices list
    - Call the `dciRemoveAllDevices` function to clear the active device list
    - Call the `dciAddDevice` function for each device in use to add it to the list
- Set the device channels to match the order of data that will be passed during processing

  - Call the `dciSetDeviceChannels` function, for each device in use, to set

- Set the device voltage offsets if required (the offsets can be obtained using the *Device Control* software or set using the interface if another data acquisition system has been used)
  - Call the `dciSetDeviceVoltageOffsets` function, for each device in use, to set
- Set the data acquisition scan rate, which is the rate at which data where obtained (the sampling frequency per channel)
  - Call the `dciSetAcquisitionScanRate` function to set
- Set the acquisition block size, which is the length of data blocks to be passed for processing
  - Call the `dciSetAcquisitionBlockSize` function to set
- Set the over-sampling ratio, which determines the size of data output blocks
  - Output block size = acquisition block size / over-sampling ratio
  - Call the `dciSetOverSamplingRatio` function to set
- Set the base name for output files
  - Call the `dciSetOutputFileName` function to set
- Set whether the processing software should output time-history and/or summary data files
  - Call the `dciSetOutputTimeHistoryFile` and `dciSetOutputSummaryFile` functions to set
- Set the absolute (barometric) pressure and temperature that existed when the data were obtained
  - Call the `dciSetAbsPressure` and `dciSetTemperature` functions to set
- Prepare the system for post processing (settings will be checked and memory allocated)
  - Call the `dciPrepareToPostProcess` function to prepare for post processing
- Pass data to the processor in blocks (the length of which is defined by the acquisition block size)
  - Call the `dciPostProcess` (C or C++) or `dciPostProcessLV` (LabVIEW) or `dciPostProcessML` (MATLAB) function to process a block of data
- Retrieve the resulting processed data if required for custom output, plotting or further processing (Note: No data will be available after the first block, only after subsequent blocks. The data returned after passing the second block for processing actually corresponds, in time, to the first block that was passed, i.e. the returned data is always one block behind the passed data. See the section titled ‘Block Processing’ for details.)

- Call the `dciGetData_FilteredPressure`, `dciGetData_UVW` and `dciGetData_VelPitchYaw` functions to get the most recent block of processed data

- Finish processing after all blocks of data have been processed (data will be saved to file and memory freed as required)
  - Call the `dciDonePostProcessing` function after all data blocks have been processed

- New to version 3.2 and above…

  (Note: This step is not necessary if the user does not require the final block of processed data.) Retrieve the final block of processed data, i.e. the data that corresponds, in time, to the last block of raw data passed for processing.
  - Call the `dciGetData_FilteredPressure`, `dciGetData_UVW` and `dciGetData_VelPitchYaw` functions to get the final block of processed data

Post-Processing Examples

The following examples provide simplified code and techniques for post-processing using various languages. For all examples, it is assumed that the appropriate device ('Cobra 100', a four-hole Cobra Probe for this example) has already been configured using the *Device Control* software. It is also assumed that the input voltage data have been 'acquired' at 2 kHz and 4096 samples will be processed in blocks of 1024. An over-sampling ratio of 2 will be used to provide a data output rate of 1 kHz with data in blocks of 512, giving a total of 2048 output samples. The processed data are saved to disk as a time-history file and a summary data file using the *Device Control* software. A temperature of 25°C and a barometric pressure of 1016 hPa are used as the conditions that existed when the raw data was acquired.

Example using C++

The following provides a simplified example of post-processing using C++. The data parameters and required output are defined at the start of the parent section above. This example assumes that 'pmDevCtrlInterfaceLink.cpp' and its associated header files are included in your project. The interface DLL will be loaded using run-time dynamic linking.

Note: All interface function calls should return a result of one (1) if successful, and this result should be checked after each call.

- Use the interface link to load the interface (all functions will be available after a successful call)
  - `result = dcilLoadInterface();`

- Initialise (without data acquisition support) and store device number to use
  - `result = dciInit(false);`
  - `int devNum = 0;`

- Configure the active device list (remove all existing devices and add the required 'Cobra 100')

- `result = dciRemoveAllDevices();`
- `result = dciAddDevice("Cobra 100");`

- Set the input data block size (1024), data rate (2000 Hz) and over-sampling ratio (2)
  - `result = dciSetAcquisitionBlockSize(1024);`
  - `result = dciSetAcquisitionScanRate(2000.0);`
  - `result = dciSetOverSamplingRatio(2);`
- Set the temperature (25°C) and barometric (absolute) pressure (101600 Pa)
  - `result = dciSetTemperature(25.0);`
  - `result = dciSetAbsPressure(101600.0);`
- Set the output file name and configure to output both time-history and summary files
  - `result = dciSetOutputFileName("c:\\interface_test");`
  - `result = dciSetOutputTimeHistoryFile(true);`
  - `result = dciSetOutputSummaryFile(true);`
- Set the device channel list (0 to 3)
  - `int channels[4] = {0,1,2,3};`
  - `result = dciSetDeviceChannels(devNum,channels);`
- Create some sample voltage offsets and set (the actual offsets need to be determined at the time of data acquisition)
  - `float offsets[4] = {0.053,0.021,-0.082,0.017};`
  - `result = dciSetDeviceVoltageOffsets(devNum,offsets);`
- Allocate memory for sample data (1024 samples per channel block in row-major format)

```
float** voltageData = new float[4];
for (int i = 0; i < 4; i++)
  voltageData[i] = new float[1024];
```

- Allocate memory for returned velocity data (512 block size at 1000 Hz)

```
float* vel = new float[512];
float* pitch = new float[512];
float* yaw = new float[512];
float* ps = new float[512];
```

- Prepare interface for post processing
  - `result = dciPrepareToPostProcess();`

- Create (normally read from memory or file) the first block of data and pass for processing

```
// simulate pressures for the outer ports
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 1024; j++)
    voltageData[i][j] = 3.0 + (float)rand()/(float)RAND_MAX;
// simulate pressures for the centre port
for (int j = 0; j < 1024; j++)
  voltageData[3][j] = 4.0 + (float)rand()/(float)RAND_MAX;
```

```
result = dciPostProcess(voltageData);
```

- Pass subsequent blocks of data and retrieve processed velocity data after each block

```
for (int block = 1; block < 4; block++)
{
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 1024; j++)
      voltageData[i][j] = 3.0 + (float)rand()/(float)RAND_MAX;
  for (int j = 0; j < 1024; j++)
    voltageData[3][j] = 4.0 + (float)rand()/(float)RAND_MAX;
  result = dciPostProcess(voltageData);
  result = dciGetData_VelPitchYaw(devNum,vel,pitch,yaw,ps);
  % Use or store processed data as required
}
```

- Complete post processing

```
result = dciDonePostProcessing();
```

- Retrieved the final block of processed velocity data

```
result = dciGetData_VelPitchYaw(devNum,vel,pitch,yaw,ps);
% Use or store processed data as required
```

- Free memory

```
for (int i = 0; i < 4; i++)
  delete[] voltageData[i];
delete[] voltageData;
delete[] vel;
delete[] pitch;
delete[] yaw;
delete[] ps;
```

- Unload (release) the interface

```
result = dciFree();
```

Example using MATLAB®

The following provides a simplified example of post-processing using MATLAB. The data parameters and required output are defined at the start of the parent section above.

Note: All interface function calls should return a result of one (1) if successful, and this result should be checked after each function call.

- Load the interface
  - `loadlibrary('pmDevCtrlInterface','pmDevCtrlInterfaceMain.h','alias','dci');`
- Initialise (without data acquisition support) and store device number to use
  - `calllib('dci','dciInit',false);`
  - `devNum = 0;`
- Configure the active device list (remove all existing devices and add the required 'Cobra 100')
  - `calllib('dci','dciRemoveAllDevices');`
  - `calllib('dci','dciAddDevice','Cobra 100');`
- Set the input data block size (1024), data rate (2000 Hz) and over-sampling ratio (2)
  - `calllib('dci','dciSetAcquisitionBlockSize',1024);`
  - `calllib('dci','dciSetAcquisitionScanRate',2000.0);`
  - `calllib('dci','dciSetOverSamplingRatio',2);`
- Set the temperature (25°C) and barometric (absolute) pressure (101600 Pa)
  - `calllib('dci','dciSetTemperature',25);`
  - `calllib('dci','dciSetAbsPressure',101600);`
- Set the output file name and configure to output both time-history and summary files
  - `calllib('dci','dciSetOutputFileName','c:\interface_test');`
  - `calllib('dci','dciSetOutputTimeHistoryFile',true);`
  - `calllib('dci','dciSetOutputSummaryFile',true);`
- Set the device channel list (0 to 3)
  - `calllib('dci','dciSetDeviceChannels',devNum,[0,1,2,3]);`
- Create some sample voltage offsets and set (the actual offsets need to be determined at the time of data acquisition)
  - `offsets = [0.053,0.021,-0.082,0.017];`

- `calllib('dci','dciSetDeviceVoltageOffsets',devNum,offsets);`

- Create some sample data (4096 samples per channel)
  - `voltageData = [3.0+rand(4096,3), 4.0+rand(4096,1)];`
- Pre-allocate memory for one block of returned velocity data (512 block size at 1000 Hz)
  - `vel=zeros(512,1); pitch=zeros(512,1); yaw=zeros(512,1); ps=zeros(512,1);`
- Prepare interface for post processing
  - `calllib('dci','dciPrepareToPostProcess');`
- Pass first block of data (note that the data are transposed!)
  - `calllib('dci','dciPostProcessML',voltageData(1:1024,:));`
- Pass subsequent blocks of data and retrieve processed velocity data after each block
  - ```
    for block = 2:4,
    start = (block-1)*1024 + 1;
    calllib('dci','dciPostProcessML',voltageData(start:start+1023,:));
    [result,vel,pitch,yaw,ps]=calllib('dci','dciGetData_VelPitchYaw',devNum,v
    el,pitch,yaw,ps);
    % Use or store returned data here as required
    end
    ```
- Complete post processing
  - `calllib('dci','dciDonePostProcessing');`
- Retrieve the final block of processed velocity data
  - ```
    [result,vel,pitch,yaw,ps]=calllib('dci','dciGetData_VelPitchYaw',devNum,v
    el,pitch,yaw,ps);
    % Use or store returned data here as required
    ```
- Unload (release) the interface
  - `unloadlibrary('dci');`

Language Notes

The following sections provide general and specific notes about using the interface with various languages.

Data Types

Data types used by this reference and their equivalents for several languages are listed below. In general, the naming format consists of a combination of the type (signed integer, unsigned integer or floating point) and the size (number of bits).

Type	Description	C/C++	Visual Basic	Visual Basic .NET	MATLAB
int8	8-bit signed integer	char	byte	byte	int8
int32	32-bit signed integer	int	long	integer	int32
float32	32-bit floating point	float	single	single	single
float64	64-bit floating point	double	double	double	double

MATLAB®

Two important considerations when using MATLAB are array data order and memory allocation. MATLAB multi-dimensional arrays are transposed in memory relative to C arrays; MATLAB uses column-major array layout while C uses row-major arrays. Therefore, multi-dimensional input arrays from MATLAB should be transposed before passing to the interface and output arrays should be transposed after returning from the interface (this does not affect vector arrays, only 2-dimensial and higher-order arrays). Failure to transpose arrays will produce incorrect results and possible program memory corruption.

It is also required to pre-allocate memory for interface function return data arrays, such as when passing an array to receive processed velocity data. The MATLAB examples provided in this guide demonstrate how to pre-allocate array memory before passing the arrays to the interface. Failure to pre-allocate memory will cause memory corruption, program instability and other problems.

Function Reference

dciInit

```
int32 __stdcall dciInit(int32 LoadDaq);
```

Use	Initialises the interface DLL.		
Arguments	int32	LoadDaq	0: Do not load data acquisition driver 1: Load data acquisition driver
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function after loading the DLL to initialise the interface. Specify whether the data acquisition driver should be loaded (only required if you intend to perform data acquisition functions via the interface, i.e. the driver is not required for post processing).		

dciInitialised

```
int32 __stdcall dciInitialised(void);
```

Use	Checks if the interface DLL has been initialised.		
Arguments	None		
Return value	int32	Initialised	0: Not initialised 1: Initialised
Remarks	Call this function to check that the interface DLL has properly initialised.		

dciFreeMemory

```
void __stdcall dciFreeMemory(void);
```

Use	Frees all memory and unloads the interface DLL.		
Arguments	void		
Return value	void		
Remarks	You do not need to call this function. It is called automatically when the DCI is released.		

dciLoadDaq

```
int32 __stdcall dciLoadDaqLoaded(void);
```

Use	Attempts to load the data acquisition driver for use by the interface DLL.		
Arguments	None		
Return value	int32	Initialised	0: Not loaded 1: Loaded
Remarks	Call this function to attempt to load the data acquisition driver. Calling this function is not required if the driver has already been loaded by the `dciInit` function.		

dciDaqLoaded

```
int32 __stdcall dciDaqLoaded(void);
```

Use	Checks if the data acquisition driver has been loaded for use by the interface DLL.		
Arguments	None		
Return value	int32	Initialised	0: Not loaded 1: Loaded
Remarks	Call this function to check if the data acquisition driver has been loaded.		

dciNumDevices

```
int32 __stdcall dciNumDevices(void);
```

Use	Returns the number of devices in use.		
Arguments	None		
Return value	int32	NumDevices	Number of devices in use
Remarks	Function returns the number of devices loaded and ready for use.		

dciGetDeviceName

```
int32 __stdcall dciGetDeviceName(int32 DeviceNumber, int8* Name, int32 Length);
```

Use	Set the device voltage offsets.		
Arguments	int32 int8* int32	DeviceNumber Name Length	Number of device in list (from 0 to NumDevices-1) Character array for device name Length of device name character array
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to get the name of a device in the active devices list. The device name is the same as that displayed on the 'DP Device Settings' page of the *Device Control* software. The length of the array passed to the function must be specified in the `Length` argument.		

dciAddDevice

```
int32 __stdcall dciAddDevice(int8* Name);
```

Use	Add a device to the active devices list.		
Arguments	int8*	Name	Character array for device name
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to add a device to the active devices list. The device name is the same as that displayed on the 'DP Device Settings' page of the *Device Control* software.		

dciRemoveDevice

```
int32 __stdcall dciRemoveDevice(int8* Name);
```

Use	Remove a device from the active devices list.		
Arguments	int8*	Name	Character array for device name
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to remove a device from the active devices list. The device name is the same as that displayed on the 'DP Device Settings' page of the *Device Control* software.		

dciRemoveDeviceByNum

```
int32 __stdcall dciRemoveDeviceByNum(int32 DeviceNumber);
```

Use	Remove a device from the active devices list.		
Arguments	int32	DeviceNumber	Number of device in list (from 0 to NumDevices-1)
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to remove a device from the active devices list.		

dciRemoveAllDevices

```
int32 __stdcall dciRemoveAllDevices(void);
```

Use	Remove all devices from the active devices list.		
Arguments	n/a	n/a	n/a
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to remove all devices from the active devices list.		

dciDeviceNumChannels

```
int32 __stdcall dciDeviceNumChannels(int32 DeviceNumber);
```

Use	Return the number of channels the specified device has.		
Arguments	int32	DeviceNumber	Number of device in list (from 0 to NumDevices-1)
Return value	int32	NumChannels	Device number of channels
Remarks	Call this function to determine the number of channels a device has.		

dciTotalNumChannels

```
int32 __stdcall dciTotalNumChannels(void);
```

Use	Return the total number of channels of all devices.		
Arguments	None		
Return value	int32	NumChannels	Total number of channels
Remarks	Call this function to determine the total number of channels for all devices in the active device list.		

dciGetDeviceChannels

```
int32 __stdcall dciGetDeviceChannels(int32 DeviceNumber, int32* Channels);
```

Use	Get the device channel list.		
Arguments	int32 int32*	DeviceNumber Channels	Number of device in list (from 0 to NumDevices-1) Channel list array to be filled
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to get a device channel list.		

dciSetDeviceChannels

```
int32 __stdcall dciSetDeviceChannels(int32 DeviceNumber, int32* Channels);
```

Use	Set the device channel list.		
Arguments	int32 int32*	DeviceNumber Channels	Number of device in list (from 0 to NumDevices-1) Channel list array
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set a device channel list. This is the same as setting a device's channel list on the 'DP Device Settings' page of the *Device Control* software. If the *Device Control* software is controlling data acquisition, the channel list is the A/D card channels. For post processing, the channel list should specify the channel order of data passed for processing.		

dciGetDeviceVoltageScales

```
int32 __stdcall dciGetDeviceVoltageScales(int32 DeviceNumber, float32* Scales);
```

Use	Get the device voltage scales.		
Arguments	int32 float32*	DeviceNumber Scales	Number of device in list (from 0 to NumDevices-1) Voltage scales list array to be filled
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to get a device voltage scales. The voltage scales are used to scale raw voltage data to engineering units during processing.		

dciSetDeviceVoltageScales

```
int32 __stdcall dciSetDeviceVoltageScales(int32 DeviceNumber, float32* Scales);
```

Use	Set the device voltage scales.		
Arguments	int32 float32*	DeviceNumber Scales	Number of device in list (from 0 to NumDevices-1) Scales list array
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set a device voltage Scales. The voltage scales are used to scale raw voltage data to engineering units during processing. If post processing data that has already been scaled, set the device scales to one.		

dciGetDeviceVoltageOffsets

```
int32 __stdcall dciGetDeviceVoltageOffsets(int32 DeviceNumber, float32* Offsets);
```

Use	Get the device voltage offsets.		
Arguments	int32 float32*	DeviceNumber Offsets	Number of device in list (from 0 to NumDevices-1) Offsets list array to be filled
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to get a device voltage offsets. The voltage offsets are subtracted from sampled or passed voltage data during processing.		

dciSetDeviceVoltageOffsets

```
int32 __stdcall dciSetDeviceVoltageOffsets(int32 DeviceNumber, float32* Offsets);
```

Use	Set the device voltage offsets.		
Arguments	int32 float32*	DeviceNumber Offsets	Number of device in list (from 0 to NumDevices-1) Offsets list array
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set a device voltage offsets. The voltage offsets are subtracted from sampled or passed voltage data during processing. If post processing data that has already had the offsets removed, set the device offsets to zero.		

dciSetDeviceTFFileName

```
int32 __stdcall dciSetDeviceTFFileName(int32 DeviceNumber, int8* TFFileName);
```

Use	Set the device transfer function file name.		
Arguments	int32 int8*	DeviceNumber TFFileName	Number of device in list (from 0 to NumDevices-1) Character array for the transfer function file name
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set a device transfer function file name. This function is only usable for devices that require a transfer function file, including Dynamic Pressure Modules. The transfer function file contains the tube response calibration (this data is used to correct for amplitude and phase distortions in tubing, and is necessary whenever accurate measurements of fluctuating pressures are required).		

dciZeroAllDevices

```
int32 __stdcall dciZeroAllDevices(void);
```

Use	Zeros all devices in the active devices list.		
Arguments	None		
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to zero all devices in the active devices list. This is the same as clicking the 'Zero' button on the main form of the *Device Control* software. Note: This function requires use of the A/D card, so a card compatible with the *Device Control* software must be available.		

dciMonitorDevices

```
int32 __stdcall dciMonitorDevices(void);
```

Use	Starts the real-time device monitor to display real-time data.		
Arguments	None		
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to start the real-time data monitor. This is the same as clicking the 'Real-Time Monitor' button on the main form of the *Device Control* software. Note: This function requires use of the A/D card, so a card compatible with the *Device Control* software must be available.		

dciStopMonitoringDevices

```
int32 __stdcall dciStopMonitoringDevices(void);
```

Use	Stops the real-time data monitor.		
Arguments	None		
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to stop the real-time data monitor. This is the same as closing the real-time display form. Note: This function requires use of the A/D card, so a card compatible with the *Device Control* software must be available.		

dciAcquisitionScanRate

```
float32 __stdcall dciAcquisitionScanRate(void);
```

Use	Returns the current data acquisition scan rate.		
Arguments	None		
Return value	float32	ScanRate	Current scan rate in Hertz
Remarks	Call this function to return the currently set acquisition scan rate (the per channel sampling rate). The acquisition scan rate must be set even for post processing.		

dciSetAcquisitionScanRate

```
int32 __stdcall dciSetAcquisitionScanRate(float32 Rate);
```

Use	Set the data acquisition scan rate.		
Arguments	floatt32	ScanRate	Acquisition scan rate in Hertz
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set the data acquisition scan rate (the per channel sampling rate). This is the same as setting the acquisition scan rate on the 'Advanced Settings' page of the *Device Control* software. Note: The acquisition scan rate must be set even for post processing as it is used during processing.		

dciAcquisitionBlockSize

```
int32 __stdcall dciAcquisitionBlockSize(void);
```

Use	Returns the current data acquisition block size.		
Arguments	None		
Return value	int32	BlockSize	Current block size
Remarks	Call this function to get the currently set acquisition block size. Note: The acquisition block size must be set even for post processing.		

dciSetAcquisitionBlockSize

```
int32 __stdcall dciSetAcquisitionBlockSize(int32 BlockSize);
```

Use	Set the data acquisition block size.		
Arguments	int32	BlockSize	Acquisition block size
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set the data acquisition block size. This is the same as setting the acquisition block size on the 'Advanced Settings' page of the *Device Control* software. Note: The acquisition block size must be set even for post processing as it is used during processing.		

dciOverSamplingRatio

```
int32 __stdcall dciOverSamplingRatio(void);
```

Use	Returns the current data over-sampling ratio.		
Arguments	None		
Return value	int32	Ratio	Current over-sampling ratio
Remarks	Call this function to get the currently set over-sampling ratio. The over-sampling ratio defines the data output rate as … DataOutputRate = AcquisitionScanRate/OverSamplingRatio Therefore, a ratio of 1 gives a data output rate equal to the acquisition rate and a ratio of 2 gives an output rate half the acquisition rate.		

dciSetOverSamplingRatio

```
int32 __stdcall dciSetOverSamplingRatio(int32 Ratio);
```

Use	Set the data over-sampling ratio.		
Arguments	int32	Ratio	Over-sampling ratio
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set the over-sampling ratio. This is the same as setting the over-sampling ratio on the 'Advanced Settings' page of the *Device Control* software. The over-sampling ratio defines the data output rate as … DataOutputRate = AcquisitionScanRate/OverSamplingRatio Therefore, a ratio of 1 gives a data output rate equal to the acquisition rate and a ratio of 2 gives an output rate half the acquisition rate. Note: The over-sampling ratio must be a radix-2 value (1, 2, 4, 8, etc.).		

dciNumOutputSamples

```
int32 __stdcall dciNumOutputSamples(void);
```

Use	Returns the current number of samples to be output.		
Arguments	None		
Return value	int32	NumSamples	Current number of output samples
Remarks	Call this function to get the currently set number of samples to be output. This is the same as the number of output samples setting on the 'Sampling Settings' page of the *Device Control* software.		

dciSetNumOutputSamples

```
int32 __stdcall dciSetNumOutputSamples(int32 NumSamples);
```

Use	Set the required number of data output samples.		
Arguments	int32	NumSamples	Required number of output samples
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set the required number of data output samples. This is the same as setting the number of output samples on the 'Sampling Settings' page of the *Device Control* software. Note: The number of output samples will be incremented to be a multiple of the output block size.		

dciSamplingTime

```
float32 __stdcall dciSamplingTime(void);
```

Use	Returns the current sampling time.		
Arguments	None		
Return value	float32	SamplingTime	Current sampling time (in seconds)
Remarks	Call this function to get the currently set sampling time. The sampling time is the length, in seconds, of the output data. This is the same as the sampling time setting on the 'Sampling Settings' page of the *Device Control* software.		

dciSetSamplingTime

```
int32 __stdcall dciSetSamplingTime(float32 SamplingTime);
```

Use	Set the required sampling time.		
Arguments	float32	SamplingTime	Required sampling time (in seconds)
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set the required sampling time. The sampling time is the length, in seconds, of the output data. This is the same as setting the sampling time on the 'Sampling Settings' page of the *Device Control* software. Note: The sampling time will be incremented to be a multiple of the block time.		

dciDataOutputRate

```
float32 __stdcall dciDataOutputRate(void);
```

Use	Returns the current data output rate.		
Arguments	None		
Return value	float32	OutputRate	Current data output rate (in Hertz)
Remarks	Call this function to get the currently set data output rate. The data output rate is the number of data samples per second in the output stream. This is the same as the data output rate setting on the 'Sampling Settings' page of the *Device Control* software.		

dciSetDataOutputRate

```
int32 __stdcall dciSetDataOutputRate(float32 OutputRate);
```

Use	Set the required data output rate.		
Arguments	float32	OutputRate	Required output rate (in Hertz)
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set the required data output rate. The data output rate is the number of data samples per second in the output stream. This is the same as setting the data output rate on the 'Sampling Settings' page of the *Device Control* software. Note: The data output rate will be incremented to be a radix-2 divisor of the data acquisition rate.		

dciOutputBlockSize

```
int32 __stdcall dciOutputBlockSize(void);
```

Use	Returns the current output block size.		
Arguments	None		
Return value	int32	BlockSize	Current output block size
Remarks	Call this function to get the currently set output block size. This is the same as the output block size setting on the 'Sampling Settings' page of the *Device Control* software.		

dciSetOutputBlockSize

```
int32 __stdcall dciSetOutputBlockSize(int32 BlockSize);
```

Use	Set the required output block size.		
Arguments	int32	BlockSize	Required output block size
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set the required output block size. This is the same as setting the output block size on the 'Sampling Settings' page of the *Device Control* software. Note: The number of output block size will be incremented to be a radix-2 divisor of the acquisition block size.		

dciOutputFileNameLength

```
int32 __stdcall dciOutputFileNameLength(void);
```

Use	Returns the length of the output file name.		
Arguments	None		
Return value	int32	Length	Output file name length
Remarks	Call this function to get the length of the currently set output file name. This function can be used to determine the length of character array required to obtain the output file name from a call to the `dciGetOutputFileName` function. Note: The length does not include the null-terminator character, so a character array to obtain the output file name must be the file name length + 1.		

dciGetOutputFileName

```
int32 __stdcall dciGetOutputFileName(int8* FileName, int32 Length);
```

Use	Returns the current output file name.		
Arguments	int8* int32	FileName Length	Character array to store the file name Length of the FileName character array
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to obtain the currently set output file name. The length of the array passed to the function must be specified in the `Length` argument. Note: The returned name will be a null-terminated character array, so the `Length` argument must allow for this, i.e. `Length` must be at least the length of the actual file name + 1.		

dciSetOutputFileName

```
int32 __stdcall dciSetOutputFileName(int8* FileName);
```

Use	Set the output file name.		
Arguments	int8*	FileName	The file name to set
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set the output file name (including path). This is the same as setting a file name in the 'Output Options' box on the main form of the *Device Control* software. Note: A file extension is not required as this will be set as required. Note: The file name should be a null-terminated character array, i.e. the last character should be ASCII code 0.		

dciOutputTimeHistoryFile

```
int32 __stdcall dciOutputTimeHistoryFile(void);
```

Use	Returns whether time-history data files will be output.		
Arguments	None		
Return value	int32	Output	0: Time-history data files will not be output 1: Time-history data files will be output
Remarks	Call this function to determine if time-history data files (the '.th?" files) will be output.		

dciSetOutputTimeHistoryFile

```
int32 __stdcall dciSetOutputTimeHistoryFile(int32 Output);
```

Use	Sets whether to output time-history data files.		
Arguments	int32	Output	0: Do not output time-history data files 1: Output time-history data files
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set whether to output time-history data files (the '.th?' files). This is the same as using the 'Output time history data' check box on the main form of the *Device Control* software.		

dciOutputSummaryFile

```
int32 __stdcall dciOutputSummaryFile(void);
```

Use	Returns whether summary data files will be output.		
Arguments	None		
Return value	int32	Output	0: Summary data files will not be output 1: Summary data files will be output
Remarks	Call this function to determine if summary data files (the '.as?" files) will be output.		

dciSetOutputSummaryFile

```
int32 __stdcall dciSetOutputSummaryFile(int32 Output);
```

Use	Sets whether to output summary data files.		
Arguments	int32	Output	0: Do not output summary data files 1: Output summary data files
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set whether to output summary data files (the '.as?' files). This is the same as using the 'Output summary data' check box on the main form of the *Device Control* software.		

dciTemperature

```
float32 __stdcall dciTemperature(void);
```

Use	Returns the current temperature.		
Arguments	None		
Return value	float32	Temperature	Temperature in degrees Celsius
Remarks	Call this function to obtain the currently set temperature.		

dciSetTemperature

```
int32 __stdcall dciSetTemperature(float32 Temperature);
```

Use	Sets the current temperature.		
Arguments	float32	Temperature	Temperature in degrees Celsius
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set the current temperature. This is the same as settings the temperature on the 'Fluid Properties' page of the *Device Control* software.		

dciAbsPressure

```
float32 __stdcall dciAbsPressure(void);
```

Use	Returns the current absolute (barometric) fluid pressure.		
Arguments	None		
Return value	float32	AbsPressure	Absolute (barometric) fluid pressure in Pascals
Remarks	Call this function to obtain the currently set absolute (barometric) fluid pressure.		

dciSetAbsPressure

```
int32 __stdcall dciSetAbsPressure(float32 AbsPressure);
```

Use	Sets the current absolute (barometric) fluid pressure.		
Arguments	float32	AbsPressure	Absolute (barometric) fluid pressure in Pascals
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to set the current absolute (barometric) fluid pressure. This is the same as settings the absolute pressure on the 'Fluid Properties' page of the *Device Control* software.		

dciSampleAndProcess

```
int32 __stdcall dciSampleAndProcess(void);
```

Use	Start sampling and processing.		
Arguments	None		
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to prepare for and start sampling and processing. The sampling and processing is entirely control by the *Device Control* software, using existing settings or those passed before calling this function. Note: This function requires use of the A/D card, so a card compatible with the *Device Control* software must be available. Note: This function returns immediately. If the function call is successful, use the Processing function to monitor for when sampling and processing completes. Use the `dciStopProcessing` function to prematurely stop sampling and processing.		

dciStopProcessing

```
int32 __stdcall dciStopProcessing(void);
```

Use	Stop sampling and processing.		
Arguments	None		
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to stop sampling and processing before the required number of samples have been obtained. Note: This function requires use of the A/D card, so a card compatible with the *Device Control* software must be available. Note: This function returns immediately. If the function call is successful, use the Processing function to monitor for when sampling and processing actually stops.		

dciProcessing

```
int32 __stdcall dciProcessing(void);
```

Use	Checks if sampling and processing is in progress.		
Arguments	None		
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to check if sampling and processing is in progress.		

dciPrepareToPostProcess

```
int32 __stdcall dciPrepareToPostProcess(void);
```

Use	Prepare for post processing.		
Arguments	None		
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to prepare for post processing. This function checks all settings and allocates memory for processing. Note: This function must be called before calling the `dciPostProcess` function.		

dciPostProcess

```
int32 __stdcall dciPostProcess(float32** VoltageData);
```

Use	Post processes a block of data.		
Arguments	float32**	VoltageData	Voltage data
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to post process a block of data. The function expects one row of data for each channel in use, i.e. one row for each channel of each device in use. The block length must be equal to `dciAcquisitionBlockSize`. Note: The first array subscript refers to the channel number and the second subscript refers to the block sample number, i.e. the channel is the major index (row) and the sample number the minor index (column). For example, data[0][7] refers to the 7th sample of channel 0.		

dciPostProcess1D

```
int32 __stdcall dciPostProcess1D(float32* VoltageData);
```

Use	Post processes a block of data in 1-D array format.		
Arguments	float32*	VoltageData	Voltage data
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to post process a block of data when it is not possible to use a 2-D array. The function expects the data to be in an array consisting of channel blocks, i.e. a block of channel 0, followed by a block of channel 1, etc. There must not be any gaps between channel blocks. The block length must be equal to `dciAcquisitionBlockSize`.		

dciPostProcessML

```
int32 __stdcall dciPostProcessML(float64* VoltageData);
```

Use	Post processes a block of data from MATLAB.		
Arguments	float64*	VoltageData	Voltage data
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to post process a block of data when using the interface via MATLAB. The function expects one column of data for each channel in use, i.e. one column for each channel of each device in use. The block length must be equal to `dciAcquisitionBlockSize`. Note: The first array subscript (row) refers to the block sample number and the second subscript (column) refers to the channel number, i.e. the sample number is the major index (row) and the channel number the minor index (column). For example, in MATLAB, data(33,2) refers to the 33rd sample of the second channel.		

dciPostProcessLV

```
int32 __stdcall dciPostProcessLV(HLabVIEW2DArray VoltageData);
```

Use	Post processes a block of data using a LabVIEW style array.		
Arguments	HLabVIEW2D	VoltageData	Voltage data passed as a LabVIEW array handle
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to post process a block of data using the LabVIEW array format (data are passed via a LabVIEW array handle). The function expects one row of data for each channel in use, i.e. one row for each channel of each device in use. Note: The first array subscript refers to the channel number and the second subscript refers to the block sample number, i.e. the channel is the major index (row) and the sample number the minor index (column). For example, data[0][7] refers to the 7th sample of channel 0.		

dciDonePostProcessing

```
int32 __stdcall dciDonePostProcessing(void);
```

Use	Complete post processing.		
Arguments	None		
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to complete post processing, i.e. after all data has been processed This function completes processing, writes data to file as required and frees memory. Note: This function must be called after all data have been processed.		

dciPostProcessTHFile

```
int32 __stdcall dciPostProcessTHFile(int8* THFileName);
```

Use	Post processes a time-history file containing raw voltage data.		
Arguments	int8*	THFileName	Character array containing the name of the TH file.
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to post process a complete set of raw voltage data stored in a time-history file. The function will configure many required settings, including the device, the acquisition sampling rate and block size and the output file name. Before calling this function, the over-sampling ratio should be set such that the required data output rate will be obtained given the data rate of the raw data. Outputting of time-history and summary files should also be configured. Note: If the TH file contains raw voltage data without offsets removed, `dciSetDeviceOffsets` must be used to set the appropriate offset values.		

dciGetData_FilteredPressure

```
int32 __stdcall dciGetData_FilteredPressure(int32 DeviceNumber, float32** Data);
```

Use	Retrieve a block of filtered pressure data during processing.		
Arguments	int32 float32**	DeviceNumber Data	Number of device in list (from 0 to NumDevices-1) Filtered pressure data in Pascals
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to obtain filtered pressure data from the last block of data processed. The function returns one row of data for each channel of pressure data. The length of each row must be equal to the OutputBlockSize, which is equal to AcquisitionBlockSize/OverSamplingRatio. Note: Memory must be allocated for the return data before calling this function. Note: This function should only be called after a successful call to the `dciPostProcess` or `dciPostProcessLV` function.		

dciGetData_FilteredPressureML

```
int32 __stdcall dciGetData_FilteredPressureML(int32 DeviceNumber, float64* Data);
```

Use	Retrieve a block of filtered pressure data during processing.		
Arguments	int32 float64*	DeviceNumber Data	Number of device in list (from 0 to NumDevices-1) Filtered pressure data in Pascals
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to obtain filtered pressure data from the last block of data processed. The function returns one column of data for each channel of pressure data. The length of each column must be equal to the OutputBlockSize, which is equal to AcquisitionBlockSize/OverSamplingRatio. Note: Memory must be allocated for the return data before calling this function. This means an array of the appropriate size must be defined, e.g. using 'zeros(outputBlockSize, numChannels)'. Note: This function should only be called after a successful call to the `dciPostProcessML` function.		

dciGetData_UVW

```
int32 __stdcall dciGetData_UVW(int32 DeviceNumber, float32* U, float32* V, float32* W,
float32* Ps);
```

Use	Retrieve a block of velocity and static pressure data during processing.		
Arguments	int32 float32* float32* float32* float32*	DeviceNumber U V W Ps	Number of device in list (from 0 to NumDevices-1) U-component of velocity data in m/s V-component of velocity data in m/s W-component of velocity data in m/s Static pressure data in Pascals
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to obtain U, V, W and static pressure data from the last block of data processed. The length of each array is equal to the OutputBlockSize, which is equal to AcquisitionBlockSize/OverSamplingRatio. Note: This function should only be called after a successful call to the `dciPostProcess` or `dciPostProcessLV` function.		

dciGetData_VelPitchYaw

```
int32 __stdcall dciGetData_VelPitchYaw(int32 DeviceNumber, float32* Velocity, float32*
Pitch, float32* Yaw, float32* Ps);
```

Use	Retrieve a block of velocity and static pressure data during processing.		
Arguments	int32 float32* float32* float32* float32*	DeviceNumber Velocity Pitch Yaw Ps	Number of device in list (from 0 to NumDevices-1) Velocity data in m/s Pitch data in degrees Yaw data in degrees Static pressure data in Pascals
Return value	int32	Success	0: Error 1: Ok
Remarks	Call this function to obtain velocity, pitch, yaw and static pressure data from the last block of data processed. The length of each array is equal to the OutputBlockSize, which is equal to AcquisitionBlockSize/OverSamplingRatio. Note: Memory for must be allocated for the return data before calling this function. Note: This function should only be called after a successful call to the `dciPostProcess` or `dciPostProcessLV` function.		

Change Log

This section provides a summary of changes made to the interface and the interface reference guide.

Version 3.7.3

23 May 14 Added `dciFreeMemory` function.

8 Nov 12 Added `dciGetDeviceVoltageScales` function.

Added `dciSetDeviceVoltageScales` function.

Changed `dciGetDeviceOffsets` function name to `dciGetDeviceVoltageOffsets`.

Changed `dciSetDeviceOffsets` function name to `dciSetDeviceVoltageOffsets`.

23 Jun 11 Changed the device name to be constant in `dciSetDeviceTFFileName` function.

Version 3.6.4

11 Dec 09 Fixed description of parameter types for the `dciAcquisitionScanRate`, `dciPostProcessML` and `dciGetData_FilteredPressureML` functions.

9 Sep 09 Fixed description of `dciOverSamplingRatio` and `dciSetOverSamplingRatio` functions

Version 3.6.3

24 Sep 08 Changed descriptions of MATLAB related functions to use (sample,channel) indexing

Version 3.3.6

5 Oct 07 Renamed `dciPostProcessMATLAB` function to `dciPostProcessML`

Added `dciGetData_FilteredPressureML` function

Added `dciDeviceNumChannels` function

Added `dciTotalNumChannels` function

Added `dciSetDeviceTFFileName` function

Version 3.2.3

24 Jan 07 Added `dciAddDevice` function

Added `dciRemoveDevice` function

Added `dciRemoveDeviceByNum` function

Added `dciRemoveAllDevices` function