This user’s manual contains useful information about the precautions, functions, and API specifications of the DL850E/DL850EV series FreeRun Application Programming Interface (ScAPI.dll).

To ensure correct use, please read this manual thoroughly during operation. Keep this manual in a safe place for quick reference.

For information about the handling precautions, functions, and operating procedures of the DL850E/DL850EV series and the handling and operating procedures of Windows, see the relevant manuals.

**Notes**

- The contents of this manual are subject to change without prior notice as a result of continuing improvements to the instrument’s performance and functionality. The figures given in this manual may differ from those that actually appear on your screen.
- Every effort has been made in the preparation of this manual to ensure the accuracy of its contents. However, should you have any questions or find any errors, please contact your nearest YOKOGAWA dealer.

**Trademarks**

- Windows 7, Windows 8, Windows 8.1, and Windows 10 are registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.
- In this manual, the ® and TM symbols do not accompany their respective registered trademark or trademark names.
- Other company and product names are trademarks or registered trademarks of their respective companies.

**Revisions**

1st Edition: March 2016
Contents

Chapter 1  Software Overview
  1.1  Software Overview ................................................................. 1-1

Chapter 2  Notes on Using the Software
  2.1  Notes on Using the Software .................................................... 2-1

Chapter 3  FreeRun API Overview
  3.1  FreeRun API Overview .............................................................. 3-1
  3.2  API Overview ................................................................. 3-2
  3.3  Basic Flow of How to Use the API .................................................. 3-3

Chapter 4  API Functional Specifications
  4.1  Definition of Class ............................................................... 4-1
  4.2  Definition of Constants ............................................................. 4-2
  4.3  Detailed API Specifications .......................................................... 4-3
  4.4  DLL Linking Method ................................................................. 4-18
1.1 Software Overview

Overview

This software (ScAPI.dll) provides an API (Application Programming Interface) for acquiring data from the DL850E/DL850EV series in FreeRun mode.

Functions

This software can be used to perform the following functions. For details, see “Detailed API Specifications.”

- Initializing the API
- Connecting and disconnecting from the measurement instrument
- Setting parameters
- Getting waveform data

Software Structure

This software package contains the following items.

- FreeRun API Library User’s Manual (this manual)
- API files (see below)

<table>
<thead>
<tr>
<th>File Name</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScAPI.dll</td>
<td>FreeRun API Library</td>
</tr>
<tr>
<td>ScAPI64.dll</td>
<td>FreeRun API Library 64-bit Version</td>
</tr>
<tr>
<td>ScAPI.lib</td>
<td>FreeRun API Import Library for C++</td>
</tr>
<tr>
<td>ScAPI.h</td>
<td>Function Declaration Header File for C++</td>
</tr>
<tr>
<td>ScAPI.Net.dll</td>
<td>FreeRun API Library for .NET</td>
</tr>
<tr>
<td>tmctl.dll</td>
<td>Communication Library</td>
</tr>
<tr>
<td>tmctl64.dll</td>
<td>Communication Library 64-bit Version</td>
</tr>
<tr>
<td>YKMUSB.dll</td>
<td>USB Communication Library</td>
</tr>
<tr>
<td>YKMUSB64.dll</td>
<td>USB Communication Library 64-bit Version</td>
</tr>
</tbody>
</table>

System Requirements

- PC

  A PC that meets the following conditions is required.

  Operating System

  - Microsoft Windows 7 (SP1 or later), Windows 8, Windows 8.1, or Windows 10
  - CPU: Core2Duo 2 GHz or better
  - Memory: At least 1 GB (at least 2 GB recommended)

- Development Environment

  Visual Studio 2008 or later, .NET Framework 3.5 or later
2.1 Notes on Using the Software

Disclaimer

YOKOGAWA assumes no responsibility for any and all damages that may occur directly or indirectly through the use of this software.

Usage Precautions

• This software is a library designed exclusively for DL850E/DL850EV series FreeRun mode. It cannot be used with other products.

• Check the version of this software and the firmware version of the DL850E/DL850EV prior to use.
The API is provided as a dynamic link library (DLL). The API can be used by linking user applications with this DLL. As shown in the following figure, the API provides functions for acquiring waveform data from the DL850E/DL850EV running in FreeRun mode and setting measurement conditions.
3.2 API Overview

This section provides an overview of the API.

Initialization and Termination
The API functions for initialization and termination are as follows.

<table>
<thead>
<tr>
<th>API Name</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScInit</td>
<td>Initialize the API</td>
<td>4-3</td>
</tr>
<tr>
<td>ScExit</td>
<td>End the API</td>
<td>4-3</td>
</tr>
</tbody>
</table>

Connection and Disconnection
The API functions for connecting and disconnecting from the measurement instrument are as follows.

<table>
<thead>
<tr>
<th>API Name</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScOpenInstrument</td>
<td>Open an instrument and get the API handle</td>
<td>4-4</td>
</tr>
<tr>
<td>ScCloseInstrument</td>
<td>Close the instrument</td>
<td>4-4</td>
</tr>
</tbody>
</table>

Getting or Setting Measurement Conditions
The API functions for getting and setting measurement conditions are as follows.

<table>
<thead>
<tr>
<th>API Name</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScSetControl</td>
<td>Send a command to the instrument</td>
<td>4-5</td>
</tr>
<tr>
<td>ScGetControl</td>
<td>Receive a command response from the instrument</td>
<td>4-5</td>
</tr>
<tr>
<td>ScQueryMessage</td>
<td>Send a command and receive a response</td>
<td>4-7</td>
</tr>
<tr>
<td>ScGetBinaryData</td>
<td>Receive binary data</td>
<td>4-6</td>
</tr>
<tr>
<td>ScSetSamplingRate</td>
<td>Set the sampling rate</td>
<td>4-12</td>
</tr>
<tr>
<td>ScGetSamplingRate</td>
<td>Get the sampling rate</td>
<td>4-12</td>
</tr>
<tr>
<td>ScGetBaseSamplingRate</td>
<td>Get the base sampling rate</td>
<td>4-12</td>
</tr>
<tr>
<td>ScGetChannelSamplingRatio</td>
<td>Get the sampling ratio from the base sampling rate</td>
<td>4-13</td>
</tr>
<tr>
<td>ScStart</td>
<td>Start measurement</td>
<td>4-8</td>
</tr>
<tr>
<td>ScStop</td>
<td>Stop measurement</td>
<td>4-8</td>
</tr>
</tbody>
</table>

Getting FreeRun Information
The API functions for getting FreeRun information are as follows.

<table>
<thead>
<tr>
<th>API Name</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScGetLatchCount</td>
<td>Get the sample count from the LATCH position</td>
<td>4-9</td>
</tr>
<tr>
<td>ScGetLatchIntervalCount</td>
<td>Get the sample count from the previous LATCH position</td>
<td>4-9</td>
</tr>
<tr>
<td>ScGetChannelDelay</td>
<td>Get the phase difference of the channel</td>
<td>4-11</td>
</tr>
<tr>
<td>ScGetStartTime</td>
<td>Get the measurement start time and date</td>
<td>4-11</td>
</tr>
<tr>
<td>ScChannelBits</td>
<td>Get the data bit count of the channel</td>
<td>4-13</td>
</tr>
<tr>
<td>ScGetChannelGain</td>
<td>Get the gain value of the channel (used to convert waveform data into actual data)</td>
<td>4-14</td>
</tr>
<tr>
<td>ScGetChannelOffset</td>
<td>Get the offset value of the channel (used to convert waveform data into actual data)</td>
<td>4-14</td>
</tr>
<tr>
<td>ScGetDataReadyCount</td>
<td>Set the data count for the DataReady event</td>
<td>4-15</td>
</tr>
<tr>
<td>ScGetDataReadyCount</td>
<td>Get the data count for the DataReady event</td>
<td>4-15</td>
</tr>
<tr>
<td>ScAddEventListener</td>
<td>Add an event listener (C++ only)</td>
<td>4-16</td>
</tr>
<tr>
<td>ScRemoveEventListener</td>
<td>Delete the event listener (C++ only)</td>
<td>4-16</td>
</tr>
<tr>
<td>ScAddCallback</td>
<td>Add a call back method (C# only)</td>
<td>4-17</td>
</tr>
<tr>
<td>ScRemoveCallback</td>
<td>Delete the call back method (C# only)</td>
<td>4-17</td>
</tr>
</tbody>
</table>

Getting Waveform Data
The API functions for getting FreeRun waveform data are as follows.

<table>
<thead>
<tr>
<th>API Name</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScLatchData</td>
<td>Latch the measurement position</td>
<td>4-8</td>
</tr>
<tr>
<td>ScGetLatchAcqData</td>
<td>Get waveform data after latching</td>
<td>4-10</td>
</tr>
</tbody>
</table>
3.3 Basic Flow of How to Use the API

Each API function is used through a handle. First, a handle is created when an instrument is opened. Then, the target instrument is accessed by passing the handle as a parameter.

![Diagram of API flow](image.png)

1) In the case of .NET, use the callback method.
3.3 Basic Flow of How to Use the API

Unmanaged Application

The basic flow of how to use the API and a sample code for C++ (unmanaged application) are provided below. Error procedures are omitted.

1. Initialize the API (required).

```c
#include "ScAPI.h"

ScInit();
```

2. Open the instrument (DL850E/DL850EV) and create a handle (required).
After opening the instrument, use this handle to access the instrument.

```c
ScHandle handle;
ScOpenInstrument(SC_WIRE_USB,"91K225903", &handle);
```

3. Add an event listener.
To use data ready events, create a class that inherits the ScEventListener class, and register it to the API. Overwriting the handleEventScDataReady() method causes the same method to be called when a data ready event occurs. Creating and adding an event listener is not a requirement. (Waveform acquisition is possible also by periodically calling a waveform acquisition procedure.)

```c
class cYourClass : public ScEventListener {
public:
    virtual void handleEventScDataReady(ScHandle handle, __int64 dataCount);
};

cYourClass* yourClass = new cYourClass();
ScAddEventListener(handle, yourClass);
```

4. Start measuring

```c
ScStart(handle);
```

5. Latch (required to acquire waveforms).
This marks the acquisition position of the waveform data.

```c
ScLatchData(handle);
```

6. Get the waveform.

```c
char buff[100000];
ScGetLatchAcqData(handle, 1, 0, buff, sizeof(buff), &count, &dataSize);
```

7. Disconnect from the instrument (required).
The handle is invalidated when this API function is called.

```c
ScCloseInstrument(handle);
```

8. Close the API (required).

```c
ScExit();
```
Managed Application

The basic flow of how to use the API and a sample code for C# (managed application) are provided below. Error procedures are omitted.

1. Initialize the API (required).
   Add ScAPINet.dll to References of the Visual Studio Solution Explorer in advance. The name space is ScAPI.Net, and the API is defined as methods in the ScAPI class.
   ```csharp
   using ScAPINet;
   . . .
   ScAPI api = new ScAPINet.ScAPI();
   api.ScInit();
   ```

2. Open the instrument (DL850E/DL850EV) and create a handle (required).
   After opening the instrument, use this handle to access the instrument.
   ```csharp
   int handle;
   api.ScOpenInstrument(ScAPI.SC_WIRE_USB,"91K225903",out handle);
   ```

3. Add an event callback method.
   To use data ready events, add a callback method to the API. The same method will be called when data ready events occur. Creating and adding a callback method is not a requirement. (Waveform acquisition is possible also by periodically calling a waveform acquisition procedure.)
   ```csharp
   private void dataReadyCallback(int hndl, int type)
   {
       . . .
   }
   api.ScAddCallback(hndl, dataReadyCallback);
   ```

4. Start measuring
   ```csharp
   api.ScStart(handle);
   ```

5. Latch (required to acquire waveforms).
   This marks the acquisition position of the waveform data.
   ```csharp
   api.ScLatchData(handle);
   ```

6. Get the waveform.
   ```csharp
   byte[] buff = new byte[100000];
   int count, dataSize;
   api.ScGetLatchAcqData<byte>(handle, 1, 0, buff, buff.Length, out count, out dataSize);
   ```

7. Disconnect from the instrument (required).
   The handle is invalidated when this API function is called.
   ```csharp
   api.ScCloseInstrument(handle);
   ```

8. Close the API (required).
   ```csharp
   api.ScExit();
   ```
Chapter 4  API Functional Specifications

4.1 Definition of Class

This section explains the API class definitions.

Class ScEventListener

Function:
Event listener class for receiving events (C++ only)

Syntax:
```cpp
class ScEventListener {
public:
    virtual void handleEventScDataReady(ScHandle handle, __int64 dataCount);
};
```

Detail:
To receive data ready events, override the handleEventScDataReady() method. Use ScAddEventListener() to create instances.
4.2 Definition of Constants

**SC_SUCCESS**

Description: Success

Syntax:

[C++] #define SC_SUCCESS 0
[C#] ScAPI.SC_SUCCESS

Detail:
Definition of a result returned by API functions

**SC_ERROR**

Description: Error

Syntax:

[C++] #define SC_ERROR 1
[C#] ScAPI.SC_ERROR

Detail:
Definition of a result returned by API functions

**SC_WIRE_USB**

Description: USB wire type (USBTMC)

Syntax:

[C++] #define SC_WIRE_USB 7
[C#] ScAPI.SC_WIRE_USB

Detail:
Definition of a wire type for connecting to the DL850 series

**SC_WIRE_LAN**

Description: LAN wire type (VXI-11)

Syntax:

[C++] #define SC_WIRE_LAN 8
[C#] ScAPI.SC_WIRE_LAN

Detail:
Definition of a wire type for connecting to the DL850 series
4.3 Detailed API Specifications

This section provides the details of the API.

ScInit

Description:
Initialize the API

Syntax:

[C++] ScResult ScInit(void);
[C#] int ScInit();

Parameters:
None

Return value:
- SC_SUCCESS: Success
- SC_ERROR: Initialization error (already initialized)

Detail:
Call once at the start of using the library.

Example [C++]:
```
#include "ScAPI.h"
...
if (ScInit() == SC_SUCCESS) {
    ...
}
```

Example [C#]:
```
using ScAPINet;
...
ScAPINet.ScAPI api = new ScAPINet.ScAPI();
if (api.ScInit() == ScAPI.SC_SUCCESS) {
    ...
}
```

ScExit

Description:
End using the API

Syntax:

[C++] ScResult ScExit(void);
[C#] int ScExit();

Parameters:
None

Return value:
- SC_SUCCESS: Success
- SC_ERROR: Error (already terminated or not initialized)

Detail:
Call once at the end of using the API.
4.3 Detailed API Specifications

**ScOpenInstrument**

**Description:**
Open the instrument

**Syntax:**
[C++]  
```c++
ScResult ScOpenInstrument(int wire, char* address, ScHandle* rHndl);
```
[C#]  
```csharp
int ScOpenInstrument(int wire, string address, out int rHndl);
```

**Parameters:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[IN]</td>
<td>wire</td>
</tr>
<tr>
<td>[IN]</td>
<td>address</td>
</tr>
<tr>
<td>[OUT]</td>
<td>rHndl</td>
</tr>
</tbody>
</table>

**Return value:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC_SUCCESS</td>
<td>Connection successful</td>
</tr>
<tr>
<td>SC_ERROR</td>
<td>Connection error</td>
</tr>
</tbody>
</table>

**Detail:**
Connects to the instrument and returns the instrument handle. Each API passes this handle to communicate with the instrument. When a connection is established, the instrument is automatically set to FreeRun mode.

**Note:**
Multiple connections to a single instrument is not possible.

**Example [C++]:**
```c++
ScHandle hndl;
if (ScOpenInstrument(SC_WIRE_USB, "91K225895", &hndl)
   == SC_SUCCESS) {
   ...
}
```

**Example [C#]:**
```csharp
int hndl;
if (api.ScOpenInstrument(ScAPI.SC_WIRE_USB, "91K225895",
                           out hndl) == ScAPI.SC_SUCCESS)
   {
   ...
}
```

**ScCloseInstrument**

**Description:**
Close the instrument

**Syntax:**
[C++]  
```c++
ScResult ScCloseInstrument(ScHandle hndl);
```
[C#]  
```csharp
int ScCloseInstrument(int hndl);
```

**Parameters:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[IN]</td>
<td>handle</td>
</tr>
</tbody>
</table>

**Return value:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC_SUCCESS</td>
<td>Success</td>
</tr>
<tr>
<td>SC_ERROR</td>
<td>Error (not connected or already disconnected)</td>
</tr>
</tbody>
</table>

**Detail:**
Disconnects from the instrument connected using ScOpenInstrument(). When disconnecting, the instrument is automatically changed from FreeRun mode back to trigger mode.

**Note:**
The handle is invalidated when this API method is called.
4.3 Detailed API Specifications

ScSetControl

Description:
Send a command

Syntax:
[C++]  ScResult ScSetControl(ScHandle hndl, char* command);
[C#]    int ScSetControl(int hndl, string command);

Parameters:
[IN] hndl   Instrument handle
[IN] command Communication command string

Return value:
SC_SUCCESS  Success
SC_ERROR    Error

Detail:
Send a command to the instrument

Note:
The return value cannot be used to determine communication command errors. It only indicates whether the command was sent successfully.

ScGetControl

Description:
Receive a response to a communication command

Syntax:
[C++]  ScResult ScGetControl(ScHandle hndl, char* buff, int buffLen, int* receiveLen);
[C#]    int ScGetControl<DT>(int hndl, ref DT[] buff, int buffLen, out int receiveLen);

Parameters:
[IN] hndl   Instrument handle
[OUT] buff   Receive buffer
[IN] buffLen Buffer size
[OUT] receiveLen Length of the received response

Return value:
SC_SUCCESS  Success
SC_ERROR    Error (no data to be received)

Detail:
Receives a response to a communication command sent in advance from the instrument.

Note:
An error occurs if a communication command has not been sent in advance.

Example [C++]:
```cpp
char buff[BUFSIZ];
int receiveLen;
if (ScGetControl(hndl, buff, sizeof(buff), &receiveLen) == SC_SUCCESS) {
    ...
}
```

Example [C#]:
```csharp
byte[] buff = new byte[256];
int receiveLen;
if (api.ScGetControl<byte>(hndl, ref buff, buff.Length, out receiveLen) == ScAPI.SC_SUCCESS) {
    string msg = System.Text.Encoding.ASCII.GetString(buff);
    printMessage(msg);
}
```
4.3 Detailed API Specifications

**ScGetBinaryData**

**Description:**
Receive binary data

**Syntax:**
[C++]
```cpp
ScResult ScGetBinaryData(ScHandle hndl, char* command, char* buf, int buffLen, int* receiveLen);
```

[C#]
```csharp
int ScGetBinaryData<DT>(int hndl, string command, DT[] buf, int buffLen, out int receiveLen);
```

**Parameters:**
- [IN] hndl Instrument handle
- [IN] command Communication command for requesting binary data
- [IN] buf Buffer for receiving binary data
- [IN] buffLen Size of the buffer for receiving binary data (bytes)
- [OUT] receiveLen Size of the received binary data (bytes)

**Return value:**
- SC_SUCCESS Success
- SC_ERROR Error

**Detail:**
Sends a command for querying binary data and receives the response.

**Note:**
The behavior when a command that does not send binary data is specified is undefined.

**Example [C++]:**
```cpp
char buff[1024];
int receiveLen;
if (ScGetBinaryData(hndl, "*:MONitor:SEND:ALL?", buff, sizeof(buff), &receiveLen)== SC_SUCCESS) {
    ...
}
```

**Example [C#]:**
```csharp
byte[] buff = new byte[1024];
int receiveLen;
if (api.ScGetBinaryData<byte>(hndl, "*:MONitor:SEND:ALL?", ref buff, buff.Length, out receiveLen) == ScAPI.SC_SUCCESS) {
    ...
}
```
4.3 Detailed API Specifications

ScQueryMessage

Description:
Send a command and receive its response

Syntax:
[C++] ScResult ScQueryMessage(ScHandle hndl, char* command, char* buff, int buffLen, int* receiveLen);
[C#] int ScQueryMessage(int hndl, string command, out string buff, int getLen, out int receiveLen);

Parameters:
- [IN] hndl Instrument handle
- [IN] command Communication Commands
- [OUT] buff Receive buffer
- [IN] buffLen Length of receive buffer (bytes). The length of data to receive in the case of the .NET version.
- [OUT] receiveLen Length of the received response

Return value:
- SC_SUCCESS Success
- SC_ERROR Error

Detail:
You can perform communication command transmission and response reception with this single API method.

Note:
You cannot use this API method for commands that do not return responses.
In the case of C# (.NET version), specify the number of bytes to receive, not the receive buffer size, in the fourth parameter.

Example [C#]:
```c#
char buff[256];
int receiveLen;
if (ScQueryMessage(hndl, "*idn?", buff, sizeof(buff), &receiveLen) == SC_SUCCESS) {
    ...
}
```

Example [C#]:
```c#
string buff;
int receiveLen;
if (api.ScQueryMessage(hndl, "*idn?", out buff, 256, out receiveLen) == ScAPI.SC_SUCCESS) {
    ...
}
```
4.3 Detailed API Specifications

ScStart

Description:  
Start measurement

Syntax:
[C++] ScResult ScStart(ScHandle hndl)
[C#] int ScStart(int hndl)

Parameters:
[IN] hndl  Instrument handle

Return value:
SC_SUCCESS Success
SC_ERROR   Error

Detail:  
Starts measurement. (Sends a Start command.)

ScStop

Description:  
Stop measurement

Syntax:
[C++] ScResult ScStop(ScHandle hndl)
[C#] int ScStop(int hndl)

Parameters:
[IN] hndl  Instrument handle

Return value:
SC_SUCCESS Success
SC_ERROR   Error

Detail:
Stops measurement. (Sends a Stop command.)

ScLatchData

Description:  
Latch FreeRun data

Syntax:
[C++] ScResult ScLatchData(ScHandle hndl)
[C#] int ScLatchData(int hndl)

Parameters:
[OUT] hndl  Instrument handle

Return value:
SC_SUCCESS Success
SC_ERROR   Error

Detail:
Marks the present measurement position of the FreeRun measurement data in the instrument.  
This position is used as a reference for getting measured data.
4.3 Detailed API Specifications

**ScGetLatchCount**

Description:  
Get the sample count from the LATCH position

Syntax:  
[C++] ScResult ScGetLatchCount(ScHandle hndl, __int64* count)  
[C#] int ScGetLatchCount(int hndl, out long count)

Parameters:  
[IN] hndl  Instrument handle  
[OUT] count  Latch position (sample count)

Return value:  
SC_SUCCESS  Success  
SC_ERROR  Error

Detail:  
Gets the latch position.  
The latch position is the sample count from when a measurement is started to the position where latching is executed with ScLatchData().

Note:  
The sample count is the number of data points acquired using a 2-channel module, regardless of whether a 2-channel module is actually used.

**ScGetLatchIntervalCount**

Description:  
Get the sample count between latches

Syntax:  
[C++] ScResult ScGetLatchIntervalCount(ScHandle hndl, __int64* count)  
[C#] int ScGetLatchIntervalCount(int hndl, out long count)

Parameters:  
[IN] hndl  Instrument handle  
[OUT] count  Sample count between latches

Return value:  
SC_SUCCESS  Success  
SC_ERROR  Error

Detail:  
Get the sample count from the previous LATCH position

Note:  
The sample count between latches is the number of data points acquired using a 2-channel module, regardless of whether a 2-channel module is actually used.
4.3 Detailed API Specifications

**ScGetLatchAcqData**

**Description:**
Get latched measurement data

**Syntax:**

[C++]
```c++
ScResult ScGetLatchAcqData(ScHandle hndl, int chNo, int subChNo, char* buff, int buffLen, int* dataCount, int* dataSize);
```

[C#]
```csharp
int ScGetLatchAcqData<DT>(int hndl, int chNo, int subChNo, DT[] buff, int buffLen, out int dataCount, out int dataSize)
```

**Parameters:**

- [IN] hndl Instrument handle
- [IN] chNo Channel number
- [IN] subChNo Sub channel number (specify 0 if there are none)
- [OUT] buff Save buffer
- [IN] buffLen Length of save buffer
- [OUT] dataCount Length of saved data (sample count)
- [OUT] dataSize Size of a point of data saved (bytes)

**Return value:**

- SC_SUCCESS Success
- SC_ERROR Error

**Detail:**
Gets latched measurement data.

**Note:**
The returned measurement data is an AD value.
To convert this into a physical value, multiply the returned value by the gain obtained by ScGetChannelGain() and add the offset obtained by ScGetChannelOffset().

**Example [C++]:**
```c++
char buff[100000];
int count;
int size;
if (ScGetLatchAcqData(hndl, 1, 0, buff, sizeof(buff),
   &count, &size) == SC_SUCCESS) {
   ...
}
```

**Example [C#]:**
```csharp
byte[] buff = new byte[100000];
int count;
int size;
if (api.ScGetLatchAcqData<byte>(hndl, 1, 0, buff, buff.Length,
   out count, out size) == ScAPI.SC_SUCCESS)
{
   ...
}
```
4.3  Detailed API Specifications

ScGetChannelDelay

Description:
Get the phase difference of the channel

Syntax:
[C++]  ScResult ScGetChannelDelay(ScHandle hndl, int chNo, int* delay)
[C#]    int ScGetChannelDelay(int hndl, int chNo, out int delay)

Parameters:

[IN]  hndl    Instrument handle
[IN]  chNo    Channel number
[OUT] delay  Phase difference

Return value:
SC_SUCCESS  Success
SC_ERROR    Error

Detail:
Gets the phase difference of the channel.
If the target channel has sub channels, phase difference may occur according to the
sample rate ratio.
This API method returns the phase difference sample count.

Note:
The phase difference between sub channels of a multi-channel module is the same.

ScGetStartTime

Description:
Get the measurement start time and date

Syntax:
[C++]  ScResult ScGetStartTime(ScHandle hndl, char* buff);
[C#]    int ScGetStartTime(int hndl, out string buff)

Parameters:

[IN]  hndl    Instrument handle
[OUT] buff   Measurement start time string

Return value:
SC_SUCCESS  Success
SC_ERROR    Error

Detail:
Gets the measurement start time as a character string.
The time is returned as a comma separated character string.
Year (2007 or later), month (1 to 12), day (1 to 32), hour (0 to 23), minute (0 to 59),
second (0 to 59), microsecond (0 to 999999), nanosecond (10 to 990)

Note:
If this method is called when measurement is stopped, the time the previous
measurement was started is returned.
### 4.3 Detailed API Specifications

#### ScSetSamplingRate

**Description:**
Set the sampling frequency

**Syntax:**
- [C++] ScResult ScSetSamplingRate(ScHandle hndl, double srate);
- [C#] int ScSetSamplingRate(int hndl, double srate)

**Parameters:**
- [IN] hndl Instrument handle
- [IN] srate Sampling frequency (Hz)

**Return value:**
- SC_SUCCESS Success
- SC_ERROR Error

**Detail:**
Sets the sampling frequency.

**Note:**
This cannot be set while measurement is in progress.

#### ScGetSamplingRate

**Description:**
Get the sampling frequency

**Syntax:**
- [C++] ScResult ScGetSamplingRate(ScHandle hndl, double* srate)
- [C#] int ScGetSamplingRate(int hndl, out double srate)

**Parameters:**
- [IN] hndl Instrument handle
- [OUT] srate Sampling frequency

**Return value:**
- SC_SUCCESS Success
- SC_ERROR Error

**Detail:**
Gets the sampling frequency.

#### ScGetBaseSamplingRate

**Description:**
Get the base sampling frequency

**Syntax:**
- [C++] ScResult ScGetBaseSamplingRate(ScHandle hndl, double* srate)
- [C#] int ScGetBaseSamplingRate(int hndl, out double srate)

**Parameters:**
- [IN] hndl Instrument handle
- [OUT] srate Sampling frequency

**Return value:**
- SC_SUCCESS Success
- SC_ERROR Error

**Detail:**
Gets the base sampling frequency (sampling frequency of a 2-channel module).
4.3 Detailed API Specifications

**ScGetChannelSamplingRatio**

Description:
Get the ratio of the base sampling frequency to the channel’s sampling frequency.

Syntax:

[C++]  ScResult ScGetChannelSamplingRatio(ScHandle hndl, int chNo, int* ratio)

[C#] int ScGetChannelSamplingRatio(int hndl, int chNo, out int ratio)

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[IN] hndl</td>
<td>Instrument handle</td>
</tr>
<tr>
<td>[IN] chNo</td>
<td>Channel number (1 to 16)</td>
</tr>
<tr>
<td>[OUT] ratio</td>
<td>Sampling frequency ratio (1 to 1000)</td>
</tr>
</tbody>
</table>

Return value:

- SC_SUCCESS Success
- SC_ERROR Error

Detail:
Get the ratio of the base sampling frequency to the channel’s sampling frequency.
If the channel’s sampling frequency is the same as the base sampling frequency, the ratio is 1. If it is half, the ratio is 2.
For a channel with sub channels, the sampling frequency may be lower than the base sampling frequency (sampling frequency of a 2-channel model). Likewise, the sample count is lower according to the ratio.

**ScGetChannelBits**

Description:
Get the channel’s data bit length.

Syntax:

[C++]  ScResult ScGetChannelBits(ScHandle hndl, int chNo, int subChNo, int* bits);

[C#] int ScGetChannelBits(int hndl, int chNo, int subChNo, out int bits)

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[IN] hndl</td>
<td>Instrument handle</td>
</tr>
<tr>
<td>[IN] chNo</td>
<td>Channel number (1 to 16)</td>
</tr>
<tr>
<td>[IN] subChNo</td>
<td>Sub channel number (1 to 64)</td>
</tr>
<tr>
<td>[OUT] bits</td>
<td>Data bit length (1 to 32)</td>
</tr>
</tbody>
</table>

Return value:

- SC_SUCCESS Success
- SC_ERROR Error

Detail:
Gets the bit length of the channel data to be acquired.

Note:
For CAN modules and the like, the returned value may not necessarily be the same as the number of bits specified with Bit Cnt.
4.3 Detailed API Specifications

**ScGetChannelGain**

**Description:**
Get the channel gain

**Syntax:**
[C++] ScResult ScGetChannelGain(ScHandle hndl, int chNo, int subChNo, double* gain);
[C#] int ScGetChannelGain(int hndl, int chNo, int subChNo, out double gain)

**Parameters:**

- [IN] hndl Instrument handle
- [IN] chNo Channel number (1 to 16)
- [IN] subChNo Sub channel number (1 to 64; specify 0 if there are none)
- [OUT] gain Gain

**Return value:**
SC_SUCCESS Success
SC_ERROR Error

**Detail:**
Gets the gain used to convert acquired measurement data into physical values.

**ScGetChannelOffset**

**Description:**
Get the channel's data offset.

**Syntax:**
[C++] ScResult ScGetChannelOffset(ScHandle hndl, int chNo, int subChNo, double* offset);
[C#] int ScGetChannelOffset(int hndl, int chNo, int subChNo, out double offset)

**Parameters:**

- [IN] hndl Instrument handle
- [IN] chNo Channel number (1 to 16)
- [IN] subChNo Sub channel number (1 to 64; specify 0 if there are none)
- [OUT] offset Offset

**Return value:**
SC_SUCCESS Success
SC_ERROR Error

**Detail:**
Gets the offset used to convert acquired measurement data into physical values.
4.3 Detailed API Specifications

**ScSetDataReadyCount**

**Description:**
Set the measurement count used to raise a DataReady event.

**Syntax:**
[C++]  ScResult ScSetDataReadyCount(ScHandle hndl, int sampleCount)
[C#]   int ScSetDataReadyCount(int hndl, int sampleCount)

**Parameters:**
- [IN] hndl Instrument handle
- [IN] sampleCount Sample count

**Return value:**
- SC_SUCCESS Success
- SC_ERROR Error

**Detail:**
During FreeRun measurement, it is possible to raise a data ready event every time a given number of points is measured. Set the measurement count used to raise DataReady events. If the count is set to the same value as the sampling frequency (100,000 if the sampling frequency is 100 kHz), an event occurs every second.

**ScGetDataReadyCount**

**Description:**
Get the measurement count used to raise a DataReady event.

**Syntax:**
[C++]  ScResult ScGetDataReadyCount(ScHandle hndl, int* sampleCount)
[C#]   int ScGetDataReadyCount(int hndl, out int sampleCount)

**Parameters:**
- [IN] hndl Instrument handle
- [OUT] sampleCount Sample count

**Return value:**
- SC_SUCCESS Success
- SC_ERROR Error

**Detail:**
Gets the measurement count used to raise DataReady events.
4.3 Detailed API Specifications

**ScAddEventListener**

**Description:**
Add an event listener

**Syntax:**

```
[C++] ScResult ScAddEventListener(ScHandle hndl, ScEventListener* listener)
```

**Parameters:**

- **[IN]** hndl: Instrument handle
- **[IN]** listener: Pointer to the event listener class

**Return value:**

- SC_SUCCESS: Success
- SC_ERROR: Error

**Detail:**
A class that inherits the ScEventListener can be added as an event listener class. Overwriting handleEventScDataReady() causes the same method to be called automatically when a data ready event occurs.

**Note:**
Currently the only event that can be acquired is the data ready event. The dataCount parameter that is passed when handleEventScDataReady() is called is the previous value. This cannot be used with the .NET version (C#).

**Example:**

```cpp
class cMyEvent : public ScEventListener {
public:
    virtual void handleEventScDataReady(ScHandle hndl, __int64 dataCount);
};

cMyEvent* ep = new cMyEvent();
ScAddEventListener(hndl, ep);
```

**ScRemoveEventListener**

**Description:**
Delete the event listener

**Syntax:**

```
[C++] ScResult ScRemoveEventListener(ScHandle hndl, ScEventListener* listener);
```

**Parameters:**

- **[IN]** hndl: Instrument handle
- **[IN]** listener: Pointer to the event listener class

**Return value:**

- SC_SUCCESS: Success
- SC_ERROR: Error

**Detail:**
Deletes a registered event listener.

**Note:**
An error will occur if you specify an event listener that has not been added. This cannot be used with the .NET version (C#).
4.3 Detailed API Specifications

ScAddCallback

Description:
Add a call back method (C# only)

Syntax:
[C#]    public delegate void ScCallback(int hndl, int type)
        int ScAddCallback(int hndl, ScCallback func)

Parameters:
[IN]  hndl Instrument handle
[IN]  func Callback method

Return value:
SC_SUCCESS Success
SC_ERROR Error

Detail:
Adds a callback method that is called when data ready events occur.

Note:
Currently the only event that can be acquired is the data ready event.
This cannot be used with C++.
The event type is passed through the type parameter of the callback method, but it is
currently not used.
Example:
private void dataReadyCallback(int hndl, int type)
{
    ....
}
if (api.ScAddCallback(hndl, dataReadyCallback) != ScAPI.SC_SUCCESS)
{
    // error
}

ScRemoveCallback

Description:
Delete the call back method (C# only)

Syntax:
[C#]    int ScRemoveCallback(int hndl, ScCallback func)

Parameters:
[IN]  hndl Instrument handle
[IN]  func Callback method

Return value:
SC_SUCCESS Success
SC_ERROR Error

Detail:
Adds a callback method that is called when data ready events occur.

Note:
This cannot be used with C++.
4.4 DLL Linking Method

For C++, only implicit linking is currently assumed for DLL linking.
To use the API through implicit linking, specify and link to the import library (.lib file), and call the API in the same manner as calling normal functions.

In addition, place the following DLLs in the same folder as the application (exe) that you create.

<table>
<thead>
<tr>
<th>Project</th>
<th>C++ (Unmanaged Application)</th>
<th>C# (Managed Application)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>32 bit</td>
<td>64 bit</td>
</tr>
<tr>
<td>ScAPI.dll</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>ScAPI64.dll</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>ScAPINet.dll</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>tmctl.dll</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>tmctl64.dll</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>YKMUSB.dll</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>YKMUSB64.dll</td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>