User's Manual

DL950 Acquisition Application Programming Interface

YOKOGAWA
Yokogawa Test & Measurement Corporation

IM D033-01EN
1st Edition

This user's manual contains useful information about the precautions, functions, and API specifications of the DL950 series acquisition API (DL950ACQAPI.dll).
To ensure correct use, please read this manual thoroughly before operation. Keep this manual in a safe place for quick reference.
For information about the handling precautions, functions, and operating procedures of the DL950 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 10 is a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.
- In this manual, the TM and ® symbols do not accompany their respective registered trademark or trademark names.
- Other company and product names are trademarks or registered trademarks of their respective holders.

Revisions

1st Edition: November 2021

1st Edition: November 2021 (YMI)

Notes on Usage

Usage Precautions

- This software is a library designed exclusively for DL950 series acquisition. It cannot be used with other products.
- Check the version of this software and the firmware version of the DL950 prior to use. This software is compatible with DL950 firmware version 1.10 and later.
- For details on how to use the DL950, see the instruction manual provided with the instrument.

Software License Agreement

Yokogawa Test & Measurement Corporation

DL950 Acquisition Application Programming Interface License Agreement

Important: Read the following terms and conditions carefully.

By installing or using DL950 Acquisition Application Programming Interface (hereafter referred to as This Software), you accept all terms and conditions in this license agreement. All rights pertaining to the software—including property rights, ownership rights, and intellectual property rights—belong to Yokogawa Test & Measurement Corporation (hereafter referred to as YOKOGAWA), YOKOGAWA's affiliated company, or original proprietor that has granted rights for licensing the software to customers based on this agreement (hereafter referred to as the original proprietor). Customers do not have any other rights other than the right to use the software in accordance with this agreement. This Software is provided for free on an "as-is" basis. You are liable for all responsibilities arising from using This Software and all responsibilities arising from referring to This Software. Regardless of whether This Software is used, you are entirely responsible for its quality, technical requirements, and regulatory requirements or regulatory conformance.

This software may contain open source software (hereafter referred to as OSS) in addition to the software that YOKOGAWA holds the rights to or the software that YOKOGAWA has been authorized to license. License terms appropriate for each OSS component are applicable in place of the terms of this license. If there is a discrepancy between the terms of the OSS license and the terms of this license, the license terms of the corresponding OSS take precedence.

Article 1: No Warranty

1. This Software is provided for free on an "as-is" basis without any warranty. YOKOGAWA will not be liable for defects or non-fulfillment of any kind. YOKOGAWA gives no guarantee that (1) the functions included in This Software will meet your requirements or your customer's requirements, (2) This Software will run without errors (e.g., bugs) or interruptions, (3) the defects and errors (e.g., bugs) in This Software will be corrected, (4) there will be no inconsistencies, mutual interference, or other effects between This Software and other software, (5) This Software or the product of This Software is correct, accurate, reliable, or up-to-date, (6) This Software is compatible with specific software required for This Software to run, or (7) This Software will not be accessed illegally or attacked through its vulnerability or the like.
2. YOKOGAWA is not always able to repair defects in or respond to questions or inquiries about This Software. Further, the contents of the software are subject to change without prior notice as a result of continuing improvements to the software's performance and functions.

Article 2: Your Responsibilities

The following acts are prohibited unless YOKOGAWA agrees or stipulates otherwise in writing.

(1) Duplicate This Software.
(2) Sell, lend, distribute, transfer, pledge, or re-license This Software or the right to use This Software or transmit it to the public or make it transmittable.
(3) Share This Software in a virtual environment (regardless of the technical method such as physical computers, virtual computers, etc.).
(4) Convert or copy This Software to any human readable form (e.g., source program) by dumping, reverse assembling, reverse compiling, reverse engineering, or the like. Modify or attempt to modify This Software into another form by correcting or translating into another language.
(5) Remove or attempt to remove the protection mechanism (copy protection) used on or added to This Software.
(6) Delete the copyright, trademarks, logos, and other indications displayed on This Software.
(7) Unless YOKOGAWA has agreed otherwise in writing, create derivative software or other computer programs or allow the creation of such works.

Article 3: Restriction on Use

1. Unless a separate written agreement is drawn between you and YOKOGAWA, This Software is not designed, manufactured, or licensed to be used for aircraft operation, ship navigation, or the planning, construction, maintenance, operation, or use of on-ground support equipment or nuclear facilities.
2. If you are using This Software for a purpose described in the previous clause, YOKOGAWA will not be held liable for any claim or damage incurred as a result of using This Software, and you will take full responsibility in resolving the issue.

Article 4: Limitation of Liability

YOKOGAWA will not be held liable for any damages incurred in relation to This Software.

Article 5: Court with Jurisdiction

Should a dispute arise as a result of using This Software or in regards to this license agreement, both parties agree to discuss the issue in good faith. If an agreement cannot be reached, the Tokyo District Court shall be the exclusive agreement jurisdictional court of the first hearing.

Contents

1.1 Software Overview

Description

This software (DL950ACQAPI.dll) provides an application programming interface (API) for obtaining waveform data being acquired by the DL950 series.

Functions

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

- Initializing the API
- Connecting and disconnecting from measuring instruments
- Setting parameters
- Getting waveform data

Software structure

This software package contains the following items.

- DL950ACQAPI User’s Manual (this manual)
- API files (see below)

File name	Content
DL950ACQAPI.dll	ACQAPI library
DL950ACQAPI64.dll	ACQAPI Library 64-bit Version
DL950ACQAPI.lib	ACQAPI Import Library for C++
DL950ACQAPI.h	Function Declaration Header File for C++
DL950ACQAPINet.dll	Free Run API Library for .NET
tmctl.dll	Communication Library
tmctl64.dll	Communication Library 64-bit Version

System requirements

PC

A PC that meets the following conditions is required.

A PC running the English or Japanese version of Windows 10 (32 bit or 64 bit)

Note that when waveform acquisition in free run mode is performed using this software, data is saved in a specified buffer. For the memory size required by the API, see “Required memory size” in section 4.1.

Development Environment

Visual Studio 2017 or later, .NET Framework 4.7 or later

System requirements for running user programs

The following environment may be necessary to perform waveform acquisition in free run mode using a program that you create with this software depending on your waveform acquisition conditions and connection type.

When using 10Gbit Ethernet connection

- CPU
  Desktop PC
  Intel Core i7-1165G7 or better, quad core (8 threads) or better, 4.7 GHz or faster
- Memory
  16 GB or more
- SSD
  512 GB or more (M.2 slot SSD recommended, read/write performance 3 GB/s or better)

When using 1Gbit Ethernet or USB connection

- CPU
  Intel Core i5-10210U or better, quad core (8 threads) or better, 4.2 GHz or faster
- Memory
  8 GB or more
- SSD
  256 GB or more (read/write performance 400 MB/s or better)

USB driver

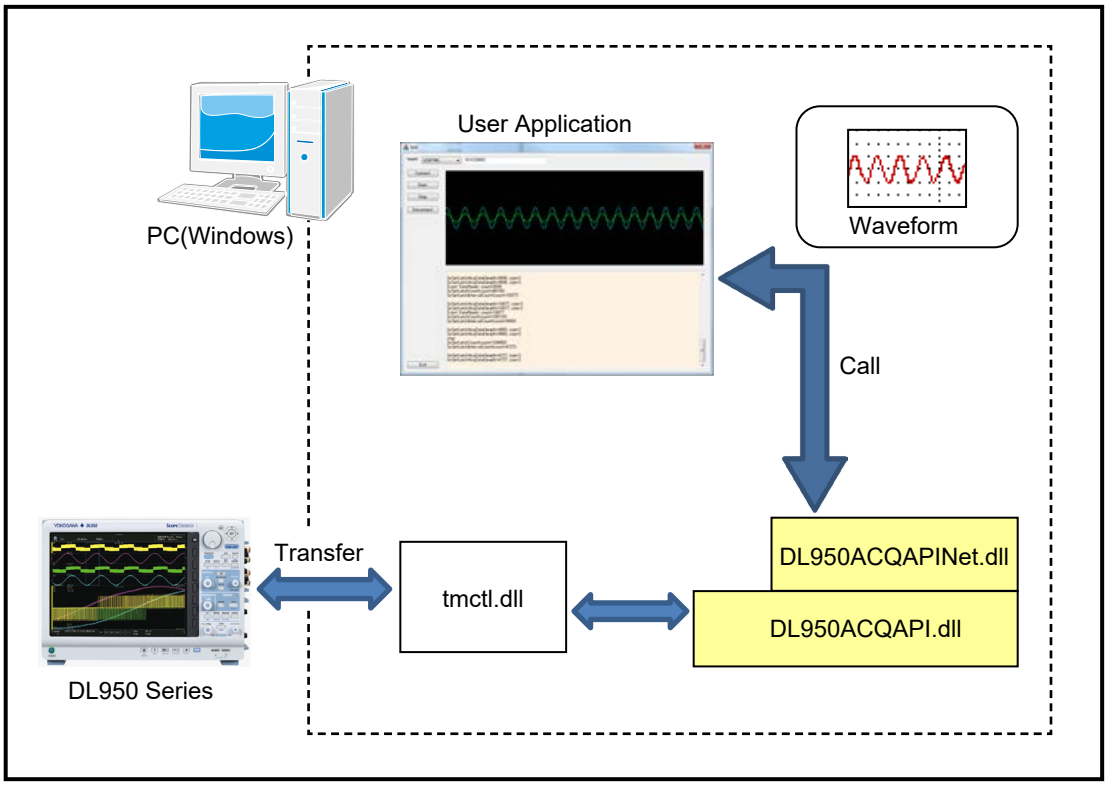
To use this software over a USB connection, you need a dedicated USB driver (YTUSB) or an IVI driver (VISA). You can download the latest USB driver from the following web page:

http://tmi.yokogawa.com/service-support/downloads/

Run Setup.exe in the YTUSB folder. The installation wizard starts. For details on the installation procedure, see the manual (ReadMe_en.pdf) in the YTUSB folder.

2.1 API Overview

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 obtaining waveform data being acquired by the instrument and setting waveform acquisition conditions.



The API only supports data acquisition in free run mode.

Free run mode
Free run mode is used to acquire data from the start to the end of waveform acquisition.

* Zoom waveform display is not possible on the DL950 while waveform acquisition in free run mode is in progress.

2.2 API Overview

This section provides an overview of the API functions.

Initialization and termination

The API functions for initialization and termination are as follows.

API Name	Function	Page
ScInit	Initialize the API	3-4
ScExit	Close the API	3-4

Connection and disconnection

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

API Name	Function	Page
ScOpenInstrument	Open an instrument and get the API handle	3-5
ScCloseInstrument	Close the instrument	3-6

Getting or setting waveform acquisition conditions

The API functions for getting and setting measurement conditions are as follows.

API Name	Function	Page
ScSetControl	Send a command to the instrument	3-6
ScGetControl	Receive a command response from the instrument	3-7
ScGetBinaryData	Receive binary data	3-8
ScQueryMessage	Send a command and receive a response	3-9
ScSet10GMode	Sets the 10G high-speed transmission mode	3-10
ScGet10GMode	Gets the 10G high-speed transmission mode	3-10
ScStart	Start acquisition	3-11
ScStop	Stop acquisition	3-11
ScSetSamplingRate	Set the sampling rate	3-16
ScGetSamplingRate	Get the sampling rate	3-16
ScGetChannelSamplingRate	Get the channel sampling rate	3-17

Getting waveform data

The API functions for getting free run waveform data are as follows.

API Name	Function	Page
ScLatchData	Latch the acquisition information	3-11
ScGetLatchRawData	Get waveform data after latching	3-12
ScGetChAcqData	Get data information of a specified channel from the block data obtained using ScGetLatchRawData	3-14

Converting waveform data

The API functions for converting waveform data into physical values are as follows.

API Name	Function	Page
ScChannelBits	Get the data bit count of the channel	3-17
ScGetChannelGain	Get the gain value of the channel (used to convert waveform data into actual data)	3-18
ScGetChannelOffset	Get the offset value of the channel (used to convert waveform data into actual data)	3-18
ScGetChannelType	Get the type of channel waveform data	3-19

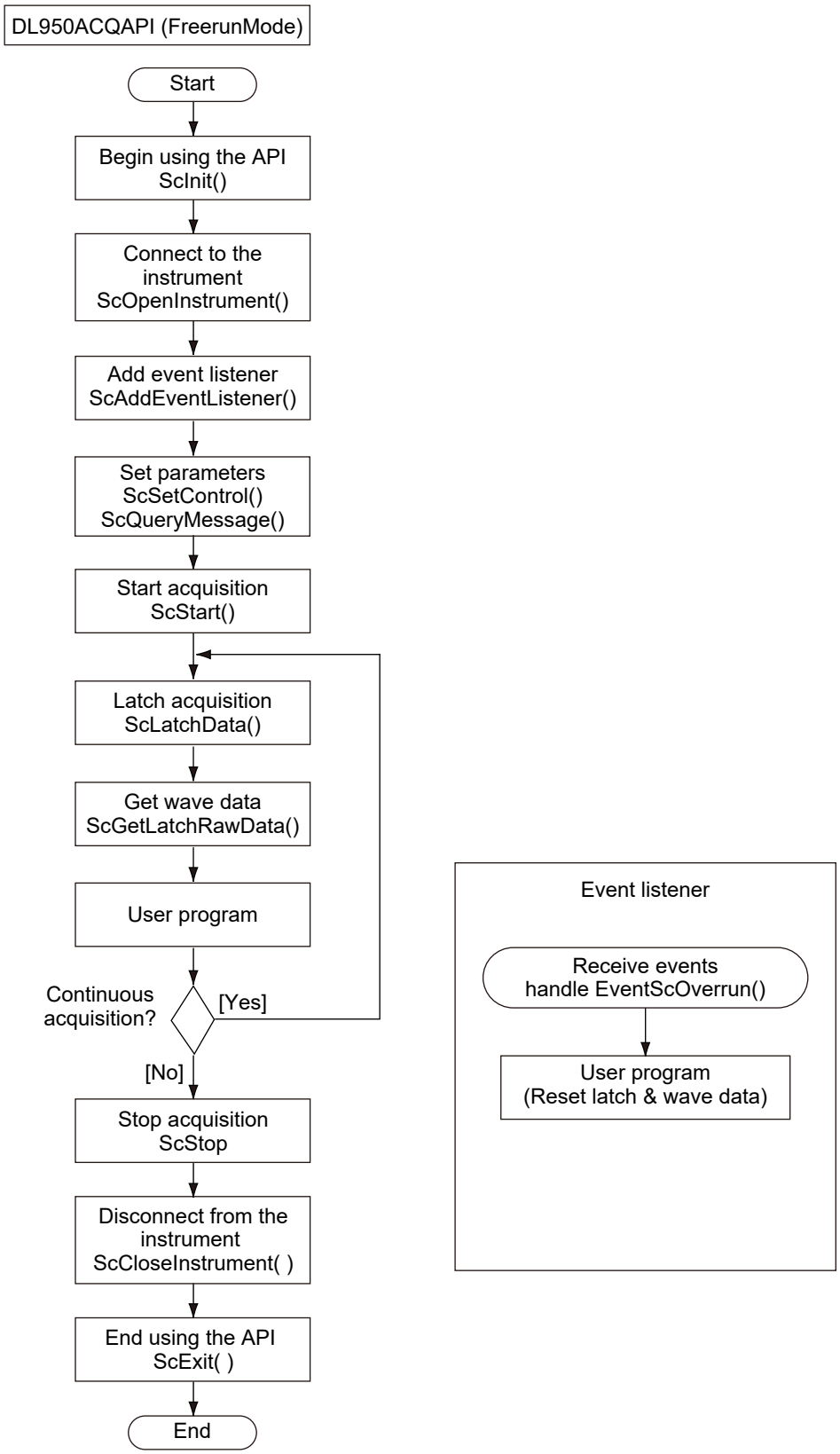
Event listener and callback functions

The event listener and callback API functions are as follows.

API Name	Function	Page
ScAddEventListener	Add an event listener (C++ only)	3-20
ScRemoveEventListener	Delete the event listener (C++ only)	3-20
ScAddCallback	Add a call back method (C# only)	3-21
ScRemoveCallback	Delete the call back method (C# only)	3-21

2.3 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 an API parameter.



Unmanaged application (free run mode)

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).

```
#include "ScAPI.h"
. . .
ScInit();
. . .
```

2. Open the instrument (DL950) and create a handle (required).
   After opening the instrument, use this handle to access the instrument.

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

3. Add an event listener.
   In a free run mode, when an interface other than 10GEther is in use, data overrun can be detected. To detect overruns, use overrun events. To use overrun events, create a class that inherits the ScEventListener class, and add it to the API. Overwriting the handleEventScOverrun() method causes the same method to be called when an overrun occurs. When an overrun is detected in free run mode, the data retrieved using waveform data acquisition becomes invalid (received data is no longer guaranteed). If this occurs, latch commands can be sent consecutively to clear this state.
   Note that if waveform acquisition sampling is slow and the communication environment allows data to be retrieved continuously, waveform acquisition is possible without adding overrun detection.

```
class cYourClass : public ScEventListener {
public:
    virtual void handleEventScOverrun(ScHandl handle);
};
. . .
cYourClass* yourClass = new YourClass();
ScAddEventListener(handle, yourClass);
```

4. Start acquisition.

```
ScStart(handle);
```

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

```
ScLatchData(handle);
```

6. Get the waveform.

```
char buff[100000];
ScGetLatchRawData(handle, buff, sizeof(buff), &recieveLen);
. . .
```

   Repeat steps 5 (latch) and 6 (waveform data acquisition) during waveform acquisition.

7. Stop acquisition.

```
ScStop(handle);
```

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

```
ScCloseInstrument(handle);
```

9. Close the API (required).

```
ScExit();
```

Managed application (free run mode)

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 ScAPINet, and the API is defined as methods in the ScAPI class.

```
using ScAPINet;
. . .
ScAPI api = new ScAPINet.ScAPI();
api.ScInit();
```

2. Open the instrument (DL950) and create a handle (required).
   After opening the instrument, use this handle to access the instrument.

```
int handle;
api.ScOpenInstrument(ScAPI.SC_WIRE_USB,"91K225903",
                     SC_FREERUN,out handle)
```

3. Add an event callback method.
   In a free run mode, when an interface other than 10GEther is in use, data overrun can be detected. To detect overruns, use overrun events. To use overrun events, add a callback method to the API. The same method will be called when overrun events occur. When an overrun is detected in free run mode, the data retrieved using waveform data acquisition becomes invalid (received data is no longer guaranteed). If this occurs, latch commands can be sent consecutively to clear this state.
   Note that if waveform acquisition sampling is slow and the communication environment allows data to be retrieved continuously, waveform acquisition is possible without adding overrun detection.

```
private void overrunCallback(int hndl, int type)
{
    ....
}
api.ScAddCallback(hndl, overrunCallback, SC_EVENTTYPE_OVERRUN);
```

4. Start acquisition.

```
api.ScStart(handle);
```

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

```
api.ScLatchData(handle);
```

6. Get the waveform.

```
byte[] buff = new byte[100000];
int recieveLen;
api.ScGetLatchRawData<byte>(handle, buff, buff.Length, out
                            receiveLen);
```

Repeat steps 5 (latch) and 6 (waveform data acquisition) during waveform acquisition.

7. Stop acquisition.

```
api.ScStop(handle);
```

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

```
api.ScCloseInstrument(handle);
```

9. Close the API (required).

```
api.ScExit();
```

3.1 Definition of Class

This section explains the API class definitions.

Class ScEventListener

Function:

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

Syntax:

```
class ScEventListener {
public:
      /*!
       * \brief Overrun handler
       * \param handle API handle
       * \param dataCount count of before LATCH posision
       */
      virtual void handleEventScOverrun(ScHandle handle){}

};
```

Details:

The overrun event in free run mode can be registered.
Overwriting handleEventScOverrun() causes the same method to be called automatically when an overrun occurs.
Use ScAddEventListener() to create instances.

3.2 Definition of Constants

SC_SUCCESS

Description:

Success

Syntax:

#define SC_SUCCESS 0

Details:

Definition of a result returned by API functions

SC_ERROR

Description:

Error

Syntax:

#define SC_ERROR 1

Details:

Definition of a result returned by API functions

SC_WIRE_USBTMC

Description:

USB wire type (YTUSB)

Syntax:

#define SC_WIRE_USBTMC

Details:

Definition of a wire type for connecting to the DL950 series

* Select this to use a USB (TMCTL standard driver) connection.

SC_WIRE_VISAUSB

Description:

USB wire type (VISAUSB)

Syntax:

#define SC_WIRE_USB

Details:

Definition of a wire type for connecting to the DL950 series

* Select this to use a USB (when a VISA standard driver is in use) connection.

SC_WIRE_VXI11

Description:

Ethernet wire type (VXI11)

Syntax:

#define SC_WIRE_VXI11

Details:

Definition of a wire type for connecting to the DL950 series

* Select this to use GigaBitEther.

SC_WIRE_HISLIP

Description:

Ethernet wire type (HiSLIP)

Syntax:

#define SC_WIRE_HISLIP

Details:

Definition of a wire type for connecting to the DL950 series
* Select this to use the 10G high-speed data transmission mode.

SC_FREERUN

Description:

Free run operation

Syntax:

#define SC_FREERUN

Details:

Specify this to perform free run mode.
Data received from the DL950 is passed as-is to the program as block data.

SC_EVENTTYPE_OVERRUN

Description:

Event type (overrun)

Syntax:

#define SC_EVENTTYPE_OVERRUN

Details:

Specify the event type for registering an overrun event callback in free run mode.
This is used only with the .NET version (C#).

3.3 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.

ScOpenInstrument

Description:

Open the instrument

Syntax:

[C++] ScResult ScOpenInstrument(int wire, char* address, int mode, ScHandle* rHndl);
[C#] int ScOpenInstrument(int wire, string address, int mode, out int rHndl);

Parameters:

[IN] wire Interface type
- SC_WIRE_USBTMC USBTMC(YTUSB)
- SC_WIRE_VISAUSB VISAUSB
- SC_WIRE_VXI11 VXI-11
- SC_WIRE_HISLIP HiSLIP

[IN] address Connection destination address (instrument serial number for USB)
[IN] mode Connection mode
- SC_FREERUN Free run

[OUT] rHndl Instrument handle

Return value:

SC_SUCCESS Connection successful
SC_ERROR Connection error

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 waveform acquisition conditions of the measuring instrument are set automatically according to the mode parameter.

Note:

Multiple connections to a single instrument is not possible.
To use 10Gbps Ethernet, select SC_WIRE_HISLIP.

Example [C++]:

```
ScHandle hndl;
if (ScOpenInstrument(SC_WIRE_USB, "91K225895", SC_FREERUN, &hndl)
== SC_SUCCESS) {
    ...
}
```

Example [C#]:

```
int hndl;
if (api.ScOpenInstrument(ScAPI.SC_WIRE_USB, "91K225895" , SC_
FREERUN, out hndl) == ScAPI.SC_SUCCESS) {
     ...
}
```

ScCloseInstrument

Description:

Close the instrument

Syntax:

[C++] ScResult ScCloseInstrument(ScHandle hndl);
[C#] int ScCloseInstrument(int hndl);

Parameters:

[IN] handle Instrument handle

Return value:

SC_SUCCESS Success
SC_ERROR Error (not connected or already disconnected)

Detail:

Disconnects from the instrument connected using ScOpenInstrument().
If the measuring instrument is in free run mode the connection is disconnected, the instrument is automatically changed from free run mode back to trigger mode.

Note:

The handle is invalidated when this API function is called.

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++]:

```
char buff[BUFSIZ];
int receiveLen;
if (ScGetControl(hndl, buff, sizeof(buff), &receiveLen) == SC_
SUCCESS) {
    ...
}
```

Example [C#]:

```
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);
}
```

ScGetBinaryData

Description:

Receive binary data

Syntax:

[C++] ScResult ScGetBinaryData(ScHandle hndl, char* command, char* buff, int buffLen, int* receiveLen, int* endFlg);

[C#] int ScGetBinaryData<DT>(int hndl, string command, DT[] buff, int buffLen, out int receiveLen, out endFlg);

Parameters:

[IN] hndl	Instrument handle
[IN] command	Communication command for requesting binary data Specify 0 (null pointer) to receive data being received.
[IN] buff	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)
[OUT] endFlg	Receive end flag
0	Receiving (remaining data available)
1	Receive end

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Detail:

Sends a command for querying binary data and receives the response.
To continue receiving data when the ScGetBinaryData, ScGetLatchAcqData, or ScGetAcqData flag is not raised, execute this API command with the parameter set to 0 (null pointer).

Note:

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

Example [C++]:

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

Example [C#]:

```
byte[] buff = new byte[1024];
int receiveLen;
if (api.ScGetBinaryData<byte>(hndl, ":MONitor:SEND:ALL?", ref
    buff, buff.Length, out receiveLen) == ScAPI.SC_SUCCESS)
{
    ...
}
```

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 command
[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#]:

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

Example [C#]:

```
string buff;
int receiveLen;
if (api.ScQueryMessage(hndl, "*idn?", out buff, 256, out
    receiveLen) == ScAPI.SC_SUCCESS)
{
    ...
}
```

ScSet10GMode

Description:

Set the 10Gbps high-speed data transmission mode

Syntax:

[C++] ScResult ScSet10GMode(ScHandle hndl, int onoff);
[C#] int ScSet10GMode(int hndl, int onoff);

Parameters:

[IN] hndl	Instrument handle	
[IN] onoff	10Gbps high-speed data transmission mode setting	
	0	10Gbps high-speed data transmission mode disabled
	1	10Gbps high-speed data transmission mode enabled

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Detail:

Set whether to use hardware-driven 10Gbps high-speed data transmission for ACQ data transmission.

Note:

This command is available when a 10Gbps Ethernet connection is established and the wire type is set to HiSlip.
Execute this command before starting acquisition (ScStart). The setting cannot be changed during waveform acquisition.
Data can be transferred via 10Gbps Ethernet even if this mode is disabled, but overruns are more likely to occur due to reduced transmission performance.

ScGet10GMode

Description:

Get the 10Gbps high-speed data transmission mode setting

Syntax:

[C++] ScResult ScGet10GMode(ScHandle hndl, int *onoff);
[C#] int ScGet10GMode<DT>(int hndl, out int onoff);

Parameters:

[IN] hndl	Instrument handle	
[OUT] onoff	10G data transmission mode setting	
	0	10G high-speed data transmission mode disabled
	1	10G high-speed data transmission mode enabled

Return value:

SC_SUCCESS	Success
SC_ERROR	Error (no data to be received)

Detail:

Checks whether hardware-driven 10Gbps high-speed data transmission mode is enabled for ACQ data transmission.

ScStart

Description:

Start acquisition

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 acquisition. (Sends a Start command.)

ScStop

Description:

Stop acquisition

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 acquisition. (Sends a Stop command.)

ScLatchData

Description:

Latch the waveform 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 acquisition position of the waveform data in the instrument.
This position is used as a reference for getting waveform data.

ScGetLatchRawData

Description:

Get latched waveform data in free run mode

Syntax:

[C++] ScResult ScGetLatchRawData(ScHandle hndl, char* buff, int buffLen, int* receiveLen, int* endFlg);

[C#] int ScGetLatchRawData<DT>(int hndl, DT[] buff, int buffLen, out int receiveLen, out endFlg)

Parameters:

[IN] hndl	Instrument handle
[OUT] buff	Save buffer
[IN] buffLen	Length of save buffer
[OUT] receiveLen	Size of the received binary data (bytes)
[OUT] endFlg	Receive end flag
	0 Receiving (remaining data available)
	1 Receive end

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Detail:

Gets latched waveform data.

Note:

The waveform data contains data of all acquisition channels and is provided in block format. For details on the block format, see “ScGetLatchRawData Data Structure” in section 4.1.

The returned waveform data is an AD value.

To convert to physical values, an appropriate data conversion is necessary according to the data type obtained with ScGetChannelType. The following formula is used.

Physical value = AD value × Gain + Offset (Gain is obtained with ScGetChannelGain and Offset with ScGetChannelOffset).

For the buffer size, see “Required memory size” in section 4.1, and specify a sufficient size.

10G high-speed data

If endFlag is 0, use ScGetBinaryData to receive the rest of the data.

Example [C++]:

```
char buff[100000];
int size;
int endFlg;
if (ScGetLatchRawData(hndl,  buff, sizeof(buff), &size, &endFlg)
                      == SC_SUCCESS) {
    ...
}
```

Example [C#]:

```
byte[] buff = new byte[100000];
int size;
int endFlg;
if (api.ScGetLatchRawData<byte>(hndl, buff, buff.Length, out
    size, out endFlg) == ScAPI.SC_SUCCESS)
{
    ...
}
```

ScGetChAcqData

Description:

Get the waveform data position of a specified channel from the data retrieved with ScGetLatchRawData

Syntax:

[C++] ScResult ScGetChAcqData(int chNo, int subChNo, char* buff, int length, int* chOffset, int* chSize, unsigned int* timeSec, , unsigned int* timeTick);

[C#] int ScGetChAcqData<DT>(int chNo, int subChNo, DT[] buff,int length, out int chOffset, out int chSize, out unsigned int timeSec, out unsigned int timeTick)

Parameters:

[IN] chNo	Channel number
[IN] subChNo	Sub channel number (specify 0 if there are none)
[IN] buff	Buffer containing data in block format
[IN] length	Size of the buffer containing data in block format
[OUT] chOffset	Offset position (number of bytes) to the head of the channel data
[OUT] chSize	Channel data size (number of bytes)
[OUT] timeSec	Time (UnixTime) at the head of the retrieved data
[OUT] timeTikc	Time (nanoseconds) at the head of the retrieved data

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Detail:

Gets the data position of the specified channel from the retrieved waveform data (block format).
The head time of the retrieved data is also obtained.

Programming tips:
When you use ScGetChAcqData to retrieve channel data in order to prevent data overruns when performing waveform acquisition at a high sampling rate, we recommend analyzing the retrieved data using a thread different from ScGetLatchRawData.
Further, when you perform waveform acquisition using 10G high-speed data streaming, we recommend not using ScGetChAcqData in order to prevent data overruns but rather using ScGetLatchRawData to only retrieve data and then using ScGetChAcqData to retrieve channel data after the waveform acquisition is completed.

Note:

Prepare a buffer large enough to store the channel data. Calculate the necessary buffer size based on the data size per point using ScGetChannelBits and the interval between latches.
Since the waveform data is AD values, to convert to physical values, an appropriate data conversion is necessary according to the data type obtained with ScGetChannelType.
The following formula is used.
Physical value = AD value × Gain + Offset (Gain is obtained with ScGetChannelGain and Offset with ScGetChannelOffset).
If the specified channel data is not available, an error will occur.
If there is no relevant channel data between latches, the data size will be 0.
For details on the block format, see “ScGetLatchRawData Data Structure” in section 4.1.

Example [C++]:

```
char buff[100000];
int size;
if (ScGetLatchRawData(hndl,  buff, sizeof(buff), &size) == SC_
    SUCCESS) {
    int chOffset;
    int chSize;
    unsigned int timeSec,timeTick;
    if (ScGetChAcqData(1, 0, buff, sizeof(buff), &chOffset,
        &chSize, &timeSec, &timeTick) == SC_SUCCESS) {
        ...
    }
    ...
}
```

Example [C#]:

```
byte[] buff = new byte[100000];
int size;
if (api.ScGetLatchRawData<byte>(hndl, buff, buff.Length, out
    size) == ScAPI.SC_SUCCESS)
{
    int chOffset;
    int chSize;
    unsigned int timeSec;
    unsigned int timeTick;
    if (api.ScGetChAcqData<byte>(1, 0, buff, buff.Length, out
        chOffset, out chSize, out timeSec, out timeTick) ==
        ScAPI.SC_SUCCESS)
    {
        ...
    }
    ...
}
```

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 waveform acquisition 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.

ScGetChannelSamplingRate

Description:

Get the channel sampling frequency

Syntax:

[C++] ScResult ScGetChannelSamplingRate(ScHandle hndl, int chNo, int subChNo, double* srate)

[C#] int ScGetChannelSamplingRate(int hndlhNo, int chNo, int subChNo, out double srate)

Parameters:

[IN] hndl	Instrument handle
[IN] chNo	Channel number
[IN] subChNo	Sub channel number (specify 0 if there are none)
[OUT] srate	Sampling frequency

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Detail:

Gets the channel sampling frequency.

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:

[IN] hndl	Instrument handle
[IN] chNo	Channel number (1 to 16)
[IN] subChNo	Sub channel number (1 to 64)
[OUT] bits	Data bit length (1 to 32)

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Detail:

Gets the bit length of the channel data (valid AD values) 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.

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 waveform 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 waveform data into physical values.

ScGetChannelType

Description:

Get the channel data type

Syntax:

[C++] ScResult ScGetChannelType(ScHandle hndl, int chNo, int subChNo, int* type);
[C#] int ScGetChannelType(int hndl, int chNo, int subChNo, out int type)

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] type	Data type

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Detail:

Gets the measurement data type.

0	ANALOG	Analog format (real value = data * gain + offset)
1	LOGIC	Logic format
2	FLOAT	Single-precision floating-point format
3	TIME	32-bit UNIX time and 32-bit fractional seconds in nanoseconds (applies to G5 sub channel number 63 or GPS sub channel number 7)

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.
The overrun event in free run mode can be registered.
Overwriting handleEventScOverrun() causes the same method to be called automatically when an overrun occurs.

Note:

The overrun event is valid when the connection type is not 10GEther.
This cannot be used with the .NET version (C#).

Example (free run mode):

```
class cMyEvent : public ScEventListener {
public:
    virtual void handleEventScOverrun(ScHandle hndl);
};

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#).

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, int type)

Parameters:

[IN] hndl	Instrument handle
[IN] func	Callback method
[IN] type	Event type

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Detail:

Adds a callback method that is called when events occur.
The overrun event in free run mode can be registered.
The event type will be SC_EVENTTYPE_OVERRUN.

Note:

The overrun event is valid when the connection type is not 10GEther.
This cannot be used with C++.

Example:

```
private void overrunCallback(int hndl, int type)
{
    ....
}
if (api.ScAddCallback(hndl, overrunCallback, SC_EVENTTYPE_
OVERRUN) != ScAPI.SC_SUCCESS)
{
    // error
}
```

ScRemoveCallback

Description:

Delete the call back method (C# only)

Syntax:

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

Parameters:

[IN] hnd	Instrument handle
[IN] func	Callback method

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Detail:

Deletes the call back method.

Note:

This cannot be used with C++.

3.4 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.

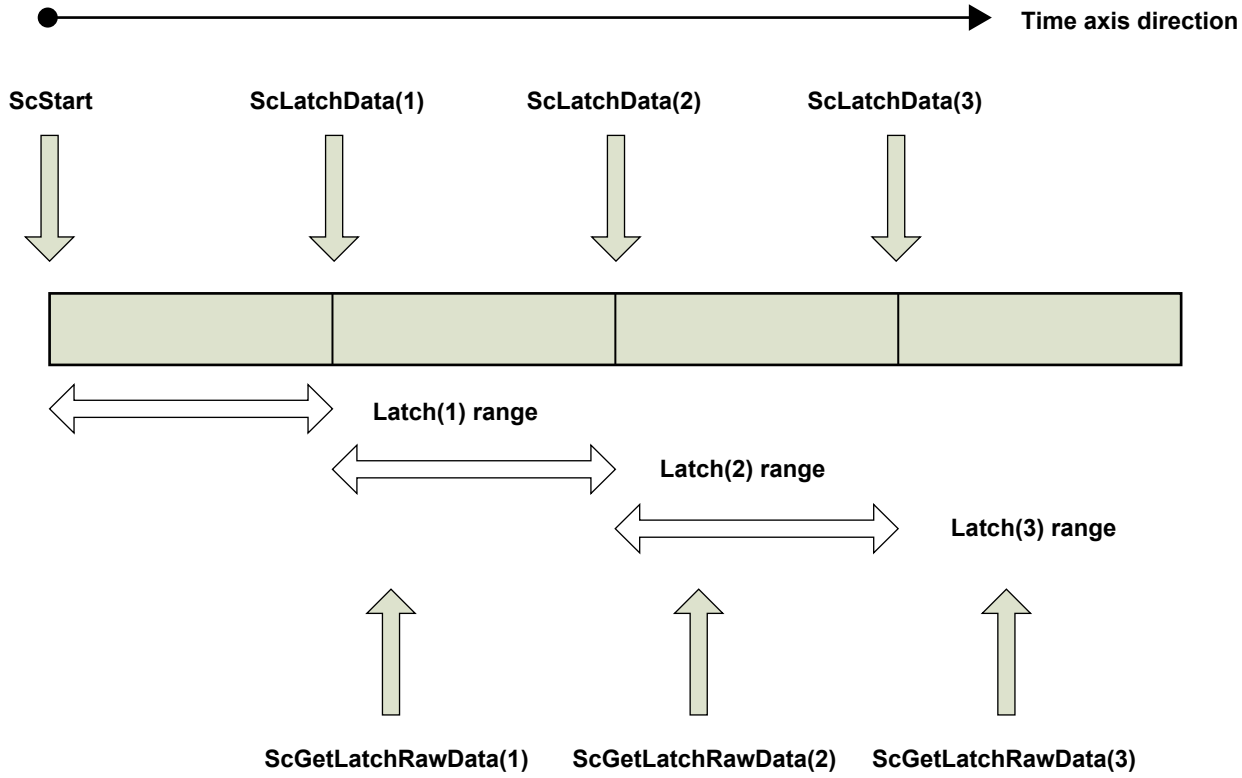
In addition, place the following DLLs in the same folder as the application (exe) that you create.

Project	C++ (unmanaged application)		C# (managed application)		
Architecture	32bit	64bit	32bit	64bit	Any CPU
DL950ACQAPI.dll	Y		Y		Y
DL950ACQAPI64.dll		Y		Y	Y
DL950ACQAPINet.dll			Y	Y	Y
tmctl.dll	Y		Y		Y
tmctl64.dll		Y		Y	Y

4.1 About Free Run Mode

Free run mode using this API and DL950 works as follows.

The DL950 starts acquiring waveforms when it receives an acquisition start (ScStart) command. It continues to acquire waveforms until it receives an acquisition stop (ScStop) command. Waveform data is temporarily stored in the instrument's acquisition memory. While the waveform acquisition is in progress, execute latches (ScLatchData) and waveform acquisitions (ScGetLatchRawData) through the API. Waveform data between latches can be retrieved.
In a single latch, the waveform data of all channels is sent from the DL950 to the API. Therefore, you need to be careful about the buffer size used by the API.



Sampling Rate, Wire Type, and Connection Mode

The available sampling rates vary depending on the type of connection used between the DL950 and the API.

10G high-speed transmission

Set the write type to Hislip (SC_WIRE_HISLIP) when establishing a connection. In this case, the DL950 can acquire using up to 10 MS/s × 16 channels.

Other types

If the connection is not 10G Hislip, the DL950 can acquire using up to 200 kS/s × 16 channels.

If the acquisition sampling rate is fast and the interval between data retrievals is long, waveform data in the DL950 memory may be overwritten.

Required memory size

When data is retrieved in free run mode, the waveform data of all channels is received in the data format described in “ScGetLatchRawData Data Structure” in section 4.1. The required memory size must be calculated using the following parameters and set with the ScGetLatchRawData command.

- Number of channels in use
- Sampling rate
- Latch interval

For example, if waveform acquisition in free run mode is executed at 200 kS/s on 16 channels (voltage module), 6400000 bytes (= 400000 bytes × 16 channels) of space are required every second.
Further, 32 bytes of space are required to store header information of each acquisition channel.
Thus, a total of 6400512 bytes (= 6400000 bytes + 32 bytes × 16 channels) of space is required every second.

ScGetLatchRawData Data Structure

In free run mode, the data received from the DL950 contains the data of all channels. The data format is shown below. The data of each acquisition channel is concatenated in the following format. All data is in Little Endian format.

1	Channel number (4 bytes)	0 to 31[1]
2	Sub channel number (4 bytes)	0 to 63[1, 2]
3	Reserved (8 bytes)	
4	Time of the first data value (8 bytes)	Unix Time (4Byte) + Tick (4 bytes, in nanoseconds (0 to 999999999))[4]
5	Data size (8 bytes)	0 or more The data size is equal to the number of ACQ data points converted into number of bytes.[3]
6	ACQ data	You can verify the data size of an ACQ point using ScGetChannelBits.[3]

1 Both channel numbers and sub channel numbers start at zero. (Acquisition channel CH1 is '0' and RMath1 is '16'.)

2 For 720240, 720241, 720242, and 720243, the number is not the sub channel number but the number of valid sub channels.
For example, if sub channels 1 and 3 are enabled and sub channel 2 is disabled, sub channel 1 is '0' and sub channel 3 is '1'.

3 For normal modules, a single data point is 2 bytes. If 17 bits or more bytes are set on CAN, for example, a single data point is 4 bytes. For RMath channels, a single data point is 4 bytes because the data is in floating point format. For sub channels of power math and harmonic math functions, a single data point is 4 bytes because the data is in floating point format. For GPS sub channels, a single data point is 4 bytes because the data is in 32-bit integer format.
For time information channels of power math, harmonic math, and GPS functions, a single data point is 8 bytes.

4 When a measurement is performed in external sampling mode, the value of this area is undefined.

Notes for when acquiring at multiple sample rates or low sample rates

If waveform acquistion is performed at multiple sample rates or low sample rate in free run mode, the data size is adjusted so that the number of data points retrieved during the waveform acquisition is fixed to a given number (integral multiple of 16). If the number becomes zero as a result of adjustment, the data of the current latch is included in the data retrieved in the next latch.

Data in timestamp format

If power analysis, harmonic analysis, or GPS position information is enabled on the analysis menu, the data for these channels will be stored in timestamp format. Data in timestamp format is always stored in pairs consisting of the computed result of each item and the time information of the computation. All data is in Little Endian format.

	Power analysis		Harmonic analysis		GPS position information
Channel	RMath13	RMath14	RMath15	RMath16	RMath1
Item's sub channel	1 to 62		1 to 62		1 to 6
	32-bit floating-point type		32-bit floating-point type		32-bit integer type
Time information sub channel	63		63		7
	64-bit time format (see below)				

Time information sub channels are recorded in the following format.

4-byte data	Unix Time (with 1970/1/1 as 0)
4-byte data	Tick (4 bytes, in nanoseconds (0 to 999999999))

If the waveform acquisition is performed using external sampling, the sample count, not the time information, is saved.

	Sample count (64-bit counter with the first data value set to 0)	
8-byte data	4-byte data	Sample count (upper 4 bytes)
	4-byte data	Sample count (lower 4 bytes)

* The sample count is not a simple 64-bit integer value, but is a value divided into upper and lower bytes. Each value is in Little Endian format.