User's
Manual

DL950 Acquisition Application Programming Interface

YOKOGAWA

Yokogawa Test & Measurement Corporation

IM D033-01EN

2nd 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
2nd Edition:	May 2022

2nd Edition: May 2022 (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 1.20 and later.
- For details on how to use the DL950, see the instruction manual provided with the instrument.

Disclaimers

By downloading and installing this software, the customer agrees to all of the following disclaimers.

- Yokogawa bears no liability for any problems occurring as a result of downloading or installing this software.
- Yokogawa bears no responsibility for any damage caused directly or indirectly as a result of using this software.
- This software is provided free of charge, however no unlimited warranty against software defects exists, nor is any claim made that the product is free of all defects whatsoever. Also, Yokogawa is not always able to repair defects ("bugs") in, or respond to questions or inquiries about this software.
- Yokogawa reserves all rights to this software, including but not limited to all property rights, ownership rights, and intellectual property rights.

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.

Features

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

Note

For features not covered by this API (mainly channel (vertical-axis) settings), implement them using communication commands by referring to the *DL950 ScopeCoder Communication Interface User’s Manual*, IM DL950-17EN.

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 (C++ only)
DL950ACQAPI.h	Function Declaration Header File (C++ only)
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 waveforms are acquired in free run mode 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 platform

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:

https://tmi.yokogawa.com/jp/library/

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.

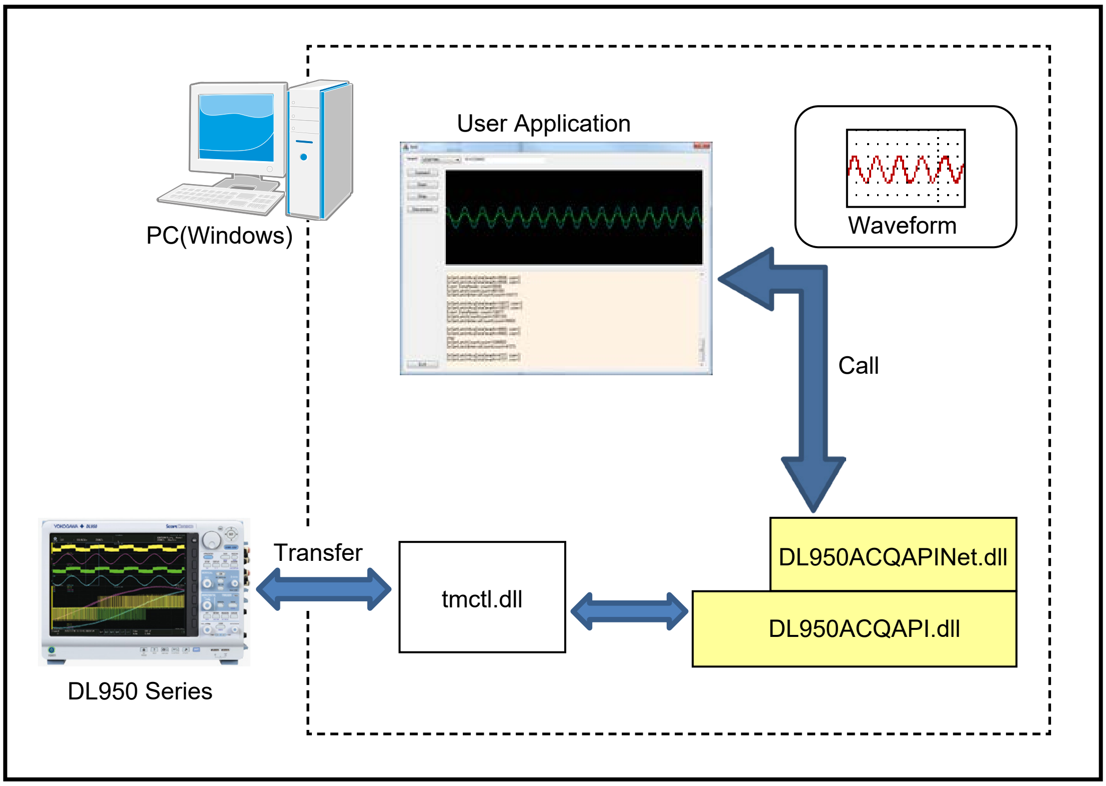
DL950 firmware version

To use this software, the DL950 firmware version need to be 1.20 or later. You can download the latest firmware from the following web page:

https://tmi.yokogawa.com/jp/library/

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 supports two acquisition modes: free run and trigger. The API supports connections to multiple DL950s but does not support multi-unit synchronization (/C50 option).

(1) 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 during waveform acquisition in free run mode.

Waveform acquisition specifications in free run mode
Maximum data rate: 320 MB/s (10 MS/s×16ch) for 10Gbit Ethernet connection
Maximum data rate: 6.4 MB/s (200 kS/s×16ch) for 1Gbit Ethernet/USB connection
Maximum waveform acquisition time: 10 days (maximum operation time guaranteed for this API)

(2) Trigger mode

Trigger mode is used to acquire waveform using triggers. There are two trigger modes available with the API: (1) synchronous mode in which the DL950 acquires waveforms synchronously with the PC and (2) asynchronous mode in which the DL950 acquires waveforms asynchronously with the PC.

Note that the API does not support the following features.

- Waveform acquisition in roll mode (the DL950 itself supports waveform acquisition in roll mode, but the API does not support waveform acquisition while the DL950 is acquiring waveforms in roll mode)
- DL950 trigger mode set to Single N
- Waveform acquisition using dual capture
- Real-time recording (SSD and flash acquisition) (/ST1, /ST2 option)
- Recorder mode

Trigger-based waveform acquisition specifications

Maximum waveform acquisition time: 10 days (maximum operation time guaranteed for this API)

When high-speed transmission mode using 10GbpsEthernet is enabled, the maximum record length that can be specified is as shown below due to the memory join limitation. For details on memory join, see the DL950 Getting Started Guide (IM DL350-03EN).

Standard model: 250 M
/M1 Model: 1 G
/M2 Model: 2 G

2.2 Overview of API Functions

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	3-5
ScExit	Close the API	3-5

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	3-6
ScCloseInstrument	Close the instrument	3-7

Getting or setting waveform acquisition conditions

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

API Name	Function	Page
ScSetControl	Send a command to the instrument	3-7
ScGetControl	Receive a command response from the instrument	3-8
ScGetBinaryData	Receive binary data	3-9
ScQueryMessage	Send a command and receive a response	3-10
ScSet10GMode	Sets the 10G high-speed transmission mode	3-11
ScGet10GMode	Gets the 10G high-speed transmission mode	3-11
ScStart	Start waveform acquisition	3-12
ScStop	Stop waveform acquisition	3-12
ScSetSamplingRate	Set the sampling rate	3-22
ScGetSamplingRate	Get the sampling rate	3-22
ScGetChannelSamplingRate	Get the channel sampling rate	3-23

Getting trigger-based waveform acquisition information

The API functions for getting trigger-based waveform acquisition information are as follows.

API Name	Function	Page
ScGetLatchAcqCount	Get the latest acquisition count at the latch point	3-19
ScGetAcqCount	Get the acquisition count for acquiring data	3-19
ScSetAcqCount	Set the acquisition count for acquiring data	3-19
ScGetTriggerTime	Get the trigger time for the specified acquisition count	3-20
ScResumeAcquisition	Resume waveform acquisition in synchronous mode	3-20
ScSetTriggerTimeout	Set the timeout value on the DL950 in synchronous mode	3-21
ScGetTriggerTimeout	Get the timeout value on the DL950 in synchronous mode	3-21

Getting waveform data

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

API Name	Function	Page
ScLatchData	Latch the waveform acquisition information	3-12
ScGetLatchRawData	Get waveform data after latching	3-13
ScGetChAcqData	Get data information of a specified channel from the block data obtained using ScGetLatchRawData	3-15

The API functions for getting waveform data in trigger mode are as follows.

API Name	Function	Page
ScLatchData	Latch the waveform acquisition information	3-12
ScGetAcqData	Get the measurement data for the specified acquisition count	3-17
ScGetAcqDataLength	Get the data length for the specified acquisition count	3-18

Converting waveform data

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

API Name	Function	Page
ScGetChannelBits	Get the data bit count of the channel	3-23
ScGetChannelGain	Get the gain value of the channel (used to convert waveform data into actual data)	3-24
ScGetChannelOffset	Get the offset value of the channel (used to convert waveform data into actual data)	3-24
ScGetChannelScale	Get the upper and lower limits of the channel display scale	3-25
ScGetChannelScale	Get the type of channel waveform data	3-25

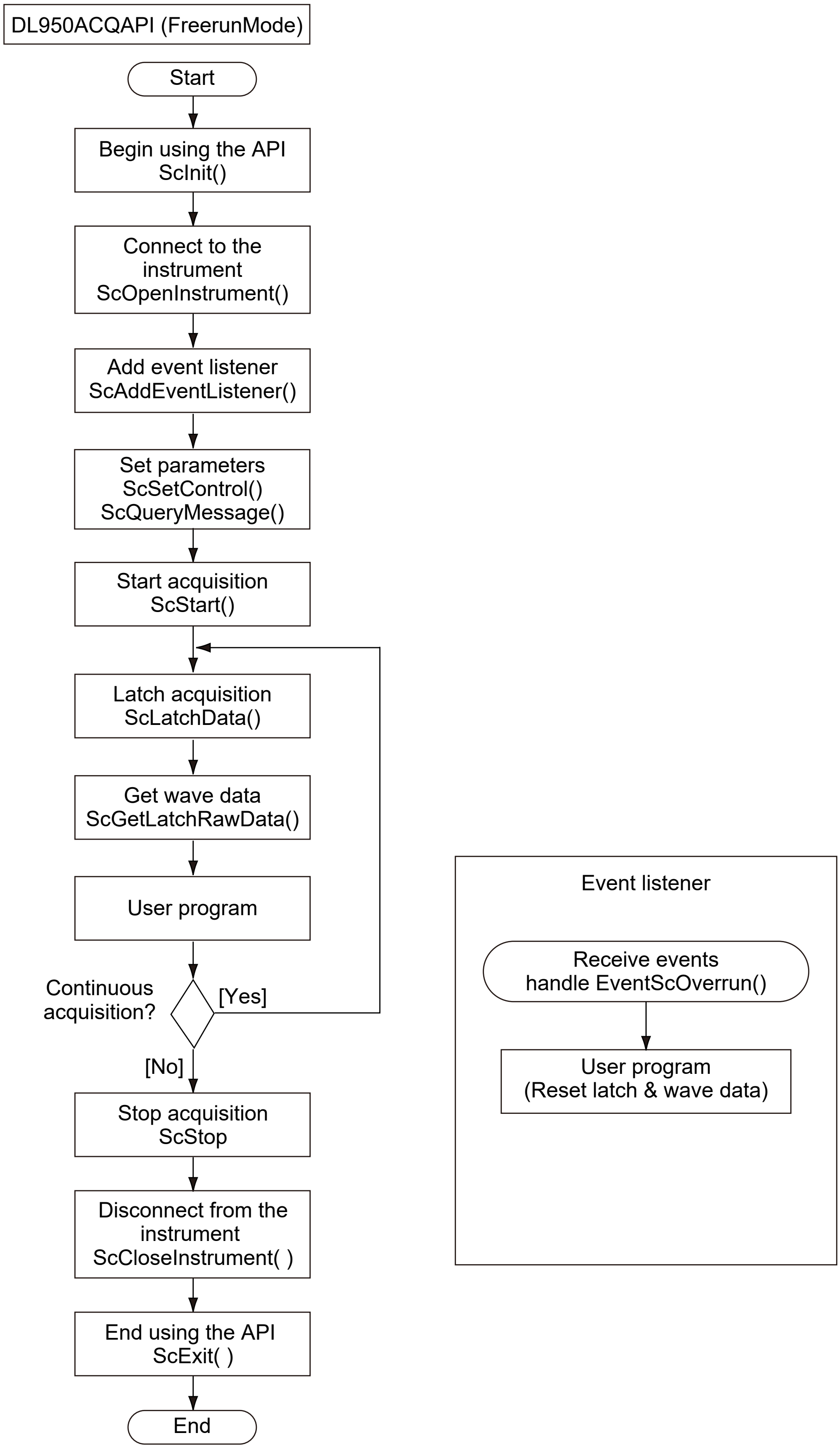
Event listener and callback functions

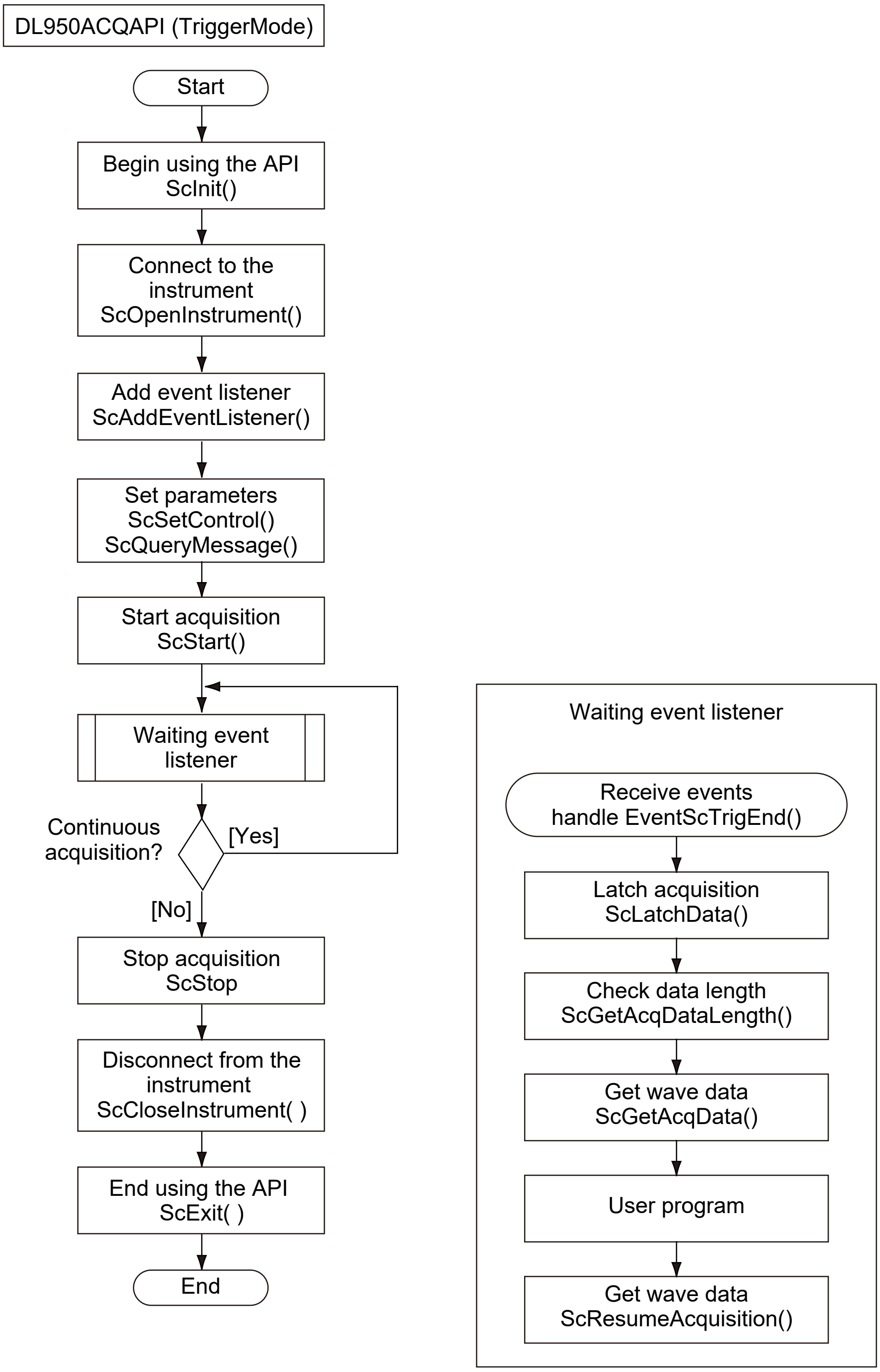
The event listener and callback API functions are as follows.

API Name	Function	Page
ScAddEventListener	Add an event listener (C++ only)	3-26
ScRemoveEventListener	Delete the event listener (C++ only)	3-27
ScAddCallback	Add a call back method (C# only)	3-27
ScRemoveCallback	Delete the call back method (C# only)	3-28

2.3 Basic Flow of Using 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.




DL950ACQAPI (TriggerMode)
Start
Begin using the API
ScInit()
Connect to the instrument
ScOpenInstrument()
Add event listener
ScAddEventListener()
Set parameters
ScSetControl()
ScQueryMessage()
Start acquisition
ScStart()
Waiting event listener
Continuous acquisition?
[Yes]
[No]
Stop acquisition
ScStop
Disconnect from the instrument
ScCloseInstrument( )
End using the API
ScExit( )
End
Waiting event listener
Receive events
handle EventScTrigEnd()
Latch acquisition
ScLatchData()
Check data length
ScGetAcqDataLength()
Get wave data
ScGetAcqData()
User program
Get wave data
ScResumeAcquisition()

Unmanaged application (free run mode)

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

1. Initialize the API (required).

```
#include “DL950ACQAPI.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 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 handleEventScCallListener() 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 handleEventScCallListener(ScHandl handle,
    int64 reserve);
};
. . .
cYourClass* yourClass = new YourClass();
ScAddEventListener(handle, yourClass);
```

4. Start waveform 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. Stops waveform acquisition

```
ScStop(handle);
```

8. Close 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 using the API and a sample code for C# (managed application) are provided below.
Error procedures are omitted.

1. Initialize the API (required).
   Add DL950ACQAPINet.dll to References of the Visual Studio Solution Explorer in advance. The name space is DL950ACQAPINet, and the API is defined as methods in the DL950ACQAPI class.

```
using DL950ACQAPINet;
. . .
DL950ACQAPI api = new DL950ACQAPINet.DL950ACQAPI();
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(DL950ACQAPI.SC_WIRE_USB,“91K225903”,
                     DL950ACQAPI.SC_FREERUN,out handle)
```

3. Add an event callback method.
   In 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,
DL950ACQAPI.SC_EVENTTYPE_OVERRUN);
```

4. Start waveform 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 a measurement.

7. Stops waveform acquisition

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

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

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

9. Close the API (required).

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

Unmanaged application (trigger mode)

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

1. Initialize the API (required).

```
#include "DL950ACQAPI.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_TRIGGER,&handle);
```

3. Add an event listener.
   In synchronous trigger mode, use trigger end events. To use trigger end events, create a class that inherits the ScEventListener class, and add it to the API. Overwriting the handleEventScTrigEnd() method causes the same method to be called when a trigger end event occurs.
   In asynchronous trigger mode, there is no need to create or register an event listener because events do not occur.

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

4. Start waveform acquisition.

```
ScStart(handle);
```

5. Latch (required to acquire waveforms).
   Measurement information (history information) is marked.

```
ScLatchData(handle);
```

6. Get the history information.
   Read the latched acquisition count, and check whether the history has been updated. If so, set the acquisition count for reading the data.

```
ScGetLatchAcqCount(handle, &acqCount);
ScGetAcqCount(handle, acqCount);
```

7. Get the waveform.
   When waveforms are acquired in trigger mode, the number of points that can be obtained with ScGetAcqDataLength() is the specified record length. The number of points depends on the record length and T/Div (sample rate). Since the size passed to ScGetAcqData is the number of bytes, determine the data point size with ScGetChannelBits in advance.

```
char buff[100000];
ScGetAcqDataLength(handle, 1,0, &length);
ScGetAcqData(handle, 1, 0, buff, sizeof(buff), &count,
&dataSize);
. . . .
ScResumeAcquisition(handle);
```

   The maximum number of bytes that can be obtained with a single ScGetAcqData call is 999,999,999 due to the communication specification limitation. Therefore, if the record length is 500 Mpoints or more (such as with voltage and other analog modules), you need to obtain the data using several ScGetAcqData calls.

   In synchronous mode, resume acquisition (ScResumeAcquisition) when the data is acquired from all necessary channels.

   Repeat steps 5 (latch) to 7 (waveform data acquisition) during waveform acquisition.

8. Stops waveform acquisition

```
ScStop(handle);
```

9. Close the instrument (required).
   The handle is invalidated when this API function is called.

```
ScCloseInstrument(handle);
```

10. Close the API (required).

```
ScExit();
```

Managed application (trigger mode)

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

1. Initialize the API (required).
   Add DL950ACQAPINet.dll to References of the Visual Studio Solution Explorer in advance. The name space is DL950ACQAPINet, and the API is defined as methods in the DL950ACQAPI class.

```
using DL950ACQAPINet;
. . .
DL950ACQAPI api = new DL950ACQAPINet.DL950ACQAPI();
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(DL950ACQAPI.SC_WIRE_USB,“91K225903”,
DL950ACQAPI.SC_TRIGGER,out handle)
```

3. Add an event callback method.
   In synchronous trigger mode, use trigger end events. To use trigger end events, add a callback method to the API. The same method will be called when trigger end events occur.
   In asynchronous trigger mode, there is no need to create or register an event listener because events do not occur.

```
private void trigEndCallback(int hndl, int type)
{
    ....
}
api.ScAddCallback(hndl, trigEndCallback,
DL950ACQAPI.SC_EVENTTYPE_TRIGEREND);
```

4. Start waveform acquisition.

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

5. Latch (required to acquire waveforms).
   Measurement information (history information) is marked.

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

6. Check the history information.
   Read the latched acquisition count, and check whether the history has been updated. If so, set the acquisition count for reading the data.

```
api.ScGetLatchAcqCount(handle, out acqCount);
api.ScGetAcqCount(handle, out acqCount);
```

7. Get the waveform.
   When waveforms are acquired in trigger mode, the number of points that can be obtained with ScGetAcqDataLength() is the specified record length. The number of points depends on the record length and T/Div (sample rate). Since the size passed to ScGetAcqData is the number of bytes, determine the data point size with ScGetChannelBits in advance.

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

   The maximum number of bytes that can be obtained with a single ScGetAcqData call is 999,999,999 due to the communication specification limitation. Therefore, if the record length is 500 Mpoints or more (such as with voltage and other analog modules), you need to obtain the data using several ScGetAcqData calls.

   In synchronous mode, resume acquisition (ScResumeAcquisition) when the data is acquired from all necessary channels.

   Repeat steps 5 (latch) to 7 (waveform data acquisition) during waveform acquisition.

8. Stops waveform acquisition

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

9. Close the instrument (required).
   The handle is invalidated when this API function is called.

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

10. 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\ reserve
       */

      virtual void handleEventScCallListener(ScHandle handle, __
int64 reserve ){}
      virtual void handleEventScTrigEnd(ScHandle handle){}

};
```

Details:

The events that you can register are the over run events for free run mode and the trigger end events for synchronous trigger mode.
Overwriting handleEventScCallListener() causes the same method to be called automatically when an overrun event occurs.
Overwriting handleEventScTrigEnd() causes the same method to be called automatically when a trigger end event occurs.
Use ScAddEventListener() to register 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_VISAUSB

Details:

Definition of a wire type for connecting to the DL950 series

* Select this to use a USB (when a VISA 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 implement waveform acquisition in free run mode.
Data received from the DL950 is passed as-is to the program as block data.

SC_TRIGGER

Description:

Synchronous trigger mode

Syntax:

#define SC_TRIGGER

Details:

Specify this to acquire waveform data in synchronous trigger mode.
The DL950 waveform acquisition sequence is explicitly controlled using the API.

SC_TRIGGER_ASYNC

Description:

Asynchronous trigger mode

Syntax:

#define SC_TRIGGER_ASYNC

Details:

Specify this to acquire waveform data in asynchronous trigger mode.
Waveform acquisition will take place on the DL950 regardless of whether data acquisition has been completed.

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#).

SC_EVENTTYPE_TRIGGEREND

Description:

Event type (trigger end)

Syntax:

```
#define SC_EVENTTYPE_TRIGEREND
```

Details:

Specify the event type for registering a trigger end event callback in trigger 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 "Dl950ACQAPI.h"
...
if (ScInit() == SC_SUCCESS) {
    ...
}
```

Example [C#]:

```
using DL950ACQAPINet;
...
DL950ACQAPINet.DL950ACQAPi api = new DL950ACQAPINet.DL950ACQAPI();
if (api.ScInit() == DL950ACQAPI.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	Wire 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
	SC_TRIGGER	Synchronous trigger mode
	SC_TRIGGER_ASYNC	Asynchronous trigger mode
[OUT] rHndl	Instrument handle	

Return value:

SC_SUCCESS	Connection successful
SC_ERROR	Connection error

Detail:

Connects to the instrument and returns the instrument handle.
This handle is passed to the APIs 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(DL950ACQAPI.SC_WIRE_USB, “91K225895” ,
DL950ACQAPI.SC_FREERUN, out hndl) == DL950ACQAPI.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 is called.

ScSetControl

Description:

Send a communication 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:

Sends a communication 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) == DL950ACQAPI.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.
When the buffer size specified by buffLen is smaller than the size of the binary data actually received, endFlg is set to zero.
To continue receiving binary data when the ScGetBinaryData, ScGetLatchAcqData, or ScGetAcqData's receive complete flag is not 1, 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) == DL950ACQAPI.SC_SUCCESS)
{
    ...
}
```

ScQueryMessage

Description:

Send a communication 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	Size of the 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 size of the receive buffer, 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) == DL950ACQAPI.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. This command can be used when the 10G Ethernet (/C60 option) is installed.

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 waveform acquisition. (ScStart). You cannot change this 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(int hndl, out int onoff);

Parameters:

[IN] hndl	Instrument handle	
[OUT] onoff	10Gbps 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 (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 waveform 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 waveform acquisition. (Sends a Start command.)

ScStop

Description:

Stop waveform 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 waveform 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.
In free run mode, this position is used as a reference for getting waveform data.
In trigger mode, history information (acquisition information) is also marked.

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	Size of the 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.
When the buffer size specified by buffLen is smaller than the size of the binary data actually received, endFlg is set to zero.

Note:

The waveform data contains data of all waveform acquisition channels and is provided in block format. For details on the block format, see “ScGetLatchRawData Data Structure.”
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.
You can use this method when SC_FREERUN is specified.

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) == DL950ACQAPI.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 (bytes) to the head of the channel data
[OUT] chSize	Channel data size (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 acquiring waveforms at a high sampling rate, we recommend analyzing the retrieved data using a thread different from ScGetLatchRawData.
Further, when you acquire waveforms using 10G high-speed data streaming, we recommend not using ScGetChAcqData during waveform acquisition 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.
You can use this method when SC_FREERUN is specified.

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) == DL950ACQAPI.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) ==
        DL950ACQAPI.SC_SUCCESS)
    {
        ...
    }
    ...
}
```

ScGetAcqData

Description:

Get latched waveform data in trigger mode

Syntax:

[C++] ScResult ScGetAcqData(ScHandle hndl, int chNo, int subChNo, char* buff, int buffLen, int* receiveLen, int* endFlg, unsigned int* timeSec, unsigned int* timeTick);

[C#] int ScGetAcqData<DT>(int hndl, int chNo, int subChNo, DT[] buff, int buffLen, out int recieveLen, out int endFlg, out unsigned int timeSec, out unsigned int timeTick)

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	Size of the save buffer
[OUT] receiveLen	Length of the acquired data (bytes)
[OUT] endFlg	Receive end flag 0 Receiving (remaining data available) 1 Receive end
[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

Details:

Gets latched waveform data.
The head time of the waveform data is also obtained.
For an overview of the operation when acquiring waveforms in trigger mode using this API, see section 4.2, “Trigger Mode.”
When the buffer size specified by buffLen is smaller than the size of the binary data actually received, endFlg is set to zero. In this case, use ScGetBinaryData to receive the rest of the data.
When waveforms are acquired using external sampling, timeSec and timeTick stores sample count values, not time information. For details, see section 4.2, “Trigger Mode.”

Note:

ScGetAcqDataLength need to be called immediately before calling this method.
The communication specifications limit the maximum number of binary data bytes that can be sent at once to 999999999 bytes. Therefore, this API needs to be executed several times depending on the set record length.
Note that the maximum number of binary data bytes sent by the DL950 in a single transmission is 999999872 bytes (499999936 data points worth for voltage modules and 249999968 data points worth for RMath). (The actual number of bytes transmitted on the communication line will be greater than 999999872 bytes as supplementary information (32 bytes worth) will be included.)
The acquired 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.
You can use this method when SC_TRIGGER or SC_TRIGGER_ASYNC is specified.

Example [C++]:

```
char buff[100000];
int size,endFlg;
unsigned int timeSec,timeTick;

if (ScGetAcqData(hndl, 1, 0, buff, sizeof(buff), &size, &endFlg, &timeSec, &timeTick) ==
    SC_SUCCESS) {
    ...
}
```

Example [C#]:

```
byte[] buff = new byte[100000];
int size,endFlg;
unsigned int timeSec,timeTick;

if (api.ScGetAcqData<byte>(hndl, 1, 0, buff, buff.Length, out size, out endFlg, out
    timeSec, out timeTick) == DL950ACQAPI.SC_SUCCESS)
{
    ...
}
```

ScGetAcqDataLength

Description:

Get the number of data points of latched waveform data in trigger mode.

Syntax:

[C++] ScResult ScGetAcqDataLength(ScHandle hndl, int chNo, int subChNo, int64* length)

[C#] int ScGetAcqDataLength(int hndl, int chNo, int subChNo, out long length)

Parameters:

[IN] hndl	Instrument handle
[IN] chNo	Channel number
[IN] subChNo	Sub channel number (specify 0 if there are none)
[OUT] length	Number of data points

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Details:

Gets the number of data points of the specified channel for the acquisition count specified by ScSetAcqCount.

Note:

What you can get with this API is the number of data points. Because the buffer size used by ScGetAcqData is specified in bytes, determine in advance the size of AD values from the value obtained by ScGetChannelBits and then calculate the necessary buffer size.

This API needs to be called before ScGetAcqData.

You can use this method when SC_TRIGGER or SC_TRIGGER_ASYNC is specified.

ScGetLatchAcqCount

Description:

Get the maximum latched acquisition count in trigger mode

Syntax:

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

Parameters:

[IN] hndl	Instrument handle
[OUT] count	Maximum acquisition count at the latch point

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Details:

Gets the maximum acquisition count at the latch point.
The obtained value is used by ScSetAcqCount.

Note:

You can use this method when SC_TRIGGER or SC_TRIGGER_ASYNC is specified.

ScGetAcqCount

Description:

Get the acquisition count to be accessed in trigger mode

Syntax:

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

Parameters:

[IN] hndl	Instrument handle
[OUT] count	Acquisition count to be accessed

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Details:

Gets the acquisition count to be accessed by ScGetAcqData, ScAcqDataLength, and ScGetTriggerTime.

Note:

You can use this method when SC_TRIGGER or SC_TRIGGER_ASYNC is specified.

ScSetAcqCount

Description:

Set the acquisition count to be accessed in trigger mode

Syntax:

[C++] ScResult ScSetAcqCount(ScHandle hndl, __int64 count)
[C#] int ScSetAcqCount(int hndl, long count)

Parameters:

[IN] hndl	Instrument handle
[IN] count	Acquisition count to be accessed

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Details:

Sets the acquisition count to be accessed by ScGetAcqData, ScAcqDataLength, and ScGetTriggerTime.

Note:

You can use this method when SC_TRIGGER or SC_TRIGGER_ASYNC is specified.

ScGetTriggerTime

Description:

Get the trigger time

Syntax:

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

Parameters:

[IN] hndl	Instrument handle
[OUT] buff	Trigger time string

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Details:

Gets the trigger time of the acquisition count specified by ScSetAcqCount as a string.
The time is returned as a comma separated character string.
Year (2007 or later), month (1 to 12), day (1 to 31), hour (0 to 23), minute (0 to 59), second (0 to 59), nanosecond (0 to 999999999),

Note:

You can use this method when SC_TRIGGER or SC_TRIGGER_ASYNC is specified.

ScResumeAcquisition

Description:

Resume waveform acquisition in synchronous trigger mode

Syntax:

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

Parameters:

[IN] hndl	Instrument handle

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Details:

Resumes the waveform acquisition on a DL950 whose waveform acquisition is being held in synchronous trigger mode.

Note:

If the DL950 is not being held in synchronous trigger mode, nothing will occur.
If the DL950 detects a timeout, waveform acquisition will be resumed even when this command is not received.
You can use this method when SC_TRIGGER is specified.

ScSetTriggerTimeout

Description:

Set the timeout value on the DL950 in synchronous trigger mode

Syntax:

[C++] ScResult ScSetTriggerTimeout(ScHandle hndl, int timeout);
[C#] int ScSetTriggerTimeout(int hndl, int timeout)

Parameters:

[IN] hndl	Instrument handle
[IN] timeout	Timeout value (0 to 497664 s, default value: 600 s)

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Details:

Sets the timeout value (in seconds) for the waveform acquisition resume command from the PC in the synchronization process with the DL950 in synchronous trigger mode.
If set to zero, the DL950 waits until a waveform acquisition resume command is received from the PC.
If set to a value between 1 and 497664, the DL950 acquires the next waveform when the specified time elapses, without waiting for a waveform acquisition resume command from the PC.
Note that if any of the following procedures is executed before a waveform acquisition resume command is received from the PC, the timer on the DL950 will restart.
ScGetAcqData, ScGetAcqDataLength, ScGetLatchAcqCount, ScGetAcqCount, ScSetAcqCount, ScGetTriggerTime

Note:

If the DL950 is not being held in synchronous trigger mode, nothing will occur.
You can use this method when SC_TRIGGER is specified.

ScGetTriggerTimeout

Description:

Get the timeout value on the DL950 in synchronous trigger mode

Syntax:

[C++] ScResult ScGetTriggerTimeout(ScHandle hndl, int *timeout);
[C#] int ScSetTriggerTimeout(int hndl, out int timeout)

Parameters:

[IN] hndl	Instrument handle
[OUT] timeout	Timeout value (0 to 497664)

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Details:

Gets the timeout value (in seconds) for the waveform acquisition resume command from the PC in the synchronization process with the DL950 in synchronous trigger mode.

ScGetMaxHistoryCount

Description:

Get the maximum number of histories in trigger mode

Syntax:

[C++] ScResult ScGetMaxHistoryCount(ScHandle hndl, int *count);

[C#] int ScGetMaxHistoryCount(int hndl, out int count)

Parameters:

[IN] hndl	Instrument handle
[OUT] count	Maximum number of histories

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Details:

Gets the maximum number of histories that can be stored in the DL950 in trigger mode.

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:

You cannot set this during waveform acquisition.

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
[IN] subChNo	Sub channel number (specify 0 if there are none)
[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
[IN] subChNo	Sub channel number (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
[IN] subChNo	Sub channel number (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.

ScGetChannelScale

Description:

Get the upper and lower limits of the channel display scale

Syntax:

[C++] ScResult ScGetChannelScale(ScHandle hndl, int chNo, int subChNo, duoble* upper, duoble* lower);

[C#] int ScGetChannelScale(int hndl, int chNo, int subChNo, out double upper, out double lower)

Parameters:

[IN] hndl	Instrument handle
[IN] chNo	Channel number
[IN] subChNo	Sub channel number (specify 0 if there are none)
[OUT] upper	Upper limit of display scale
[OUT] lower	Lower limit of display scale

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Details:

Gets the upper and lower limits of the display scale set on the DL950 screen.

ScGetChannelType

Description:

Get the channel data type

Syntax:

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

[C#] int ScGetChannelType(int hndl, int chNo, int subChNo, out string type)

Parameters:

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

Return value:

SC_SUCCESS	Success
SC_ERROR	Error

Detail:

Gets the type of waveform data to be acquired as a string. The string is normally in abbreviated form as it conforms to the response specifications of communication commands.

ANALog	Analog format (real value = data * gain + offset)
LOGic	Logic format
FLOat	Single-precision floating-point format (applies to RMath channels)
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 events that you can register are the over run events for free run mode and the trigger end events for synchronous trigger mode.
Overwriting handleEventScCallListener() causes the same method to be called automatically when an overrun event occurs.
Overwriting handleEventScTrigEnd() causes the same method to be called automatically when a trigger end event 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 handleEventScCallListener(ScHandle hndl, int64
                                           reserve);
};

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

Example (synchronous trigger mode):

```
class cMyEvent : public ScEventListener {
public:
    virtual void handleEventScTrigEnd(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 events that you can register are the over run events for free run mode and the trigger end events for synchronous trigger mode.
The event type will be SC_EVENTTYPE_OVERRUN or SC_EVENTTYPE_TRIGGEREND.

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) != DL950ACQAPI.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.

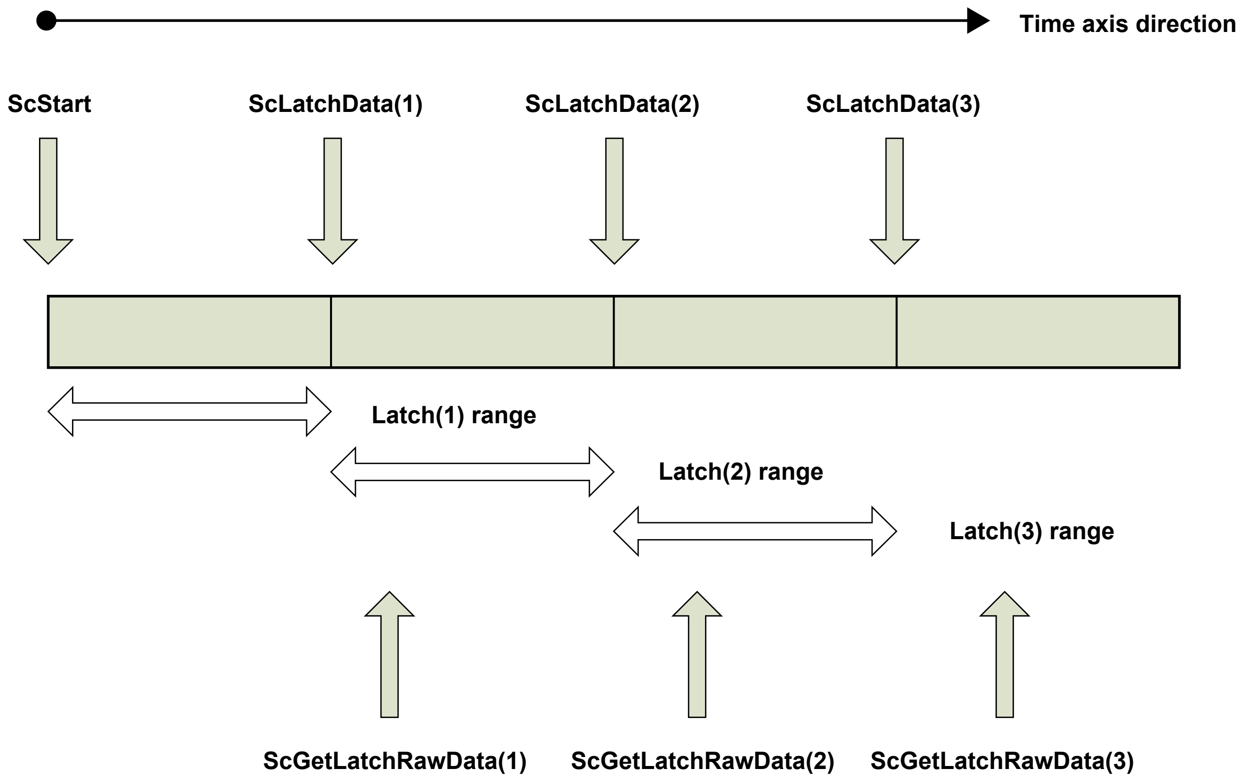
In addition, place the following DLLs in the same folder as the application (exe) that you create.

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 Free Run Mode

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

The DL950 starts acquiring waveforms when it receives a waveform acquisition start (ScStart) command. It continues to acquire waveforms until it receives a waveform 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 for waveform acquisition 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 waveforms using up to 10 MS/s × 16 channels.

Other types

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

If the sampling rate for acquiring waveforms 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 data of all waveform acquisition 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 waveforms are acquired in free run mode at 200 kS/s on 16 channels (voltage module), 6400000 bytes (= 400000 bytes × 16 channels) of space are required every second.
Furthermore, 32 bytes of data are required for storing header information for each channel acquiring waveforms.
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 acquiring waveforms.

The data format is shown below. The data of each channel is concatenated in the following format. All data is in Little Endian format.

1	Channel number (4 bytes)	0 to 31[1]	Framed area (1)
2	Sub channel number (4 bytes)	0 to 63[1, 2]	Framed area (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]	Framed area (3)
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]	Framed area (4)
6	ACQ data	You can verify the data size of an ACQ point using ScGetChannelBits. [3]	Framed area (5)

```
                                          (2)
ADDRESS       00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 (1)  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000010 (3)  E4 14 CC 61 0A 68 FA 0B E0 2F 00 00 00 00 00 00 (4)
00000020      90 04 90 04 A0 04 90 04 08 04 80 04 90 04 70 04
00000030      70 04 70 04 70 04 70 04 50 04 60 04 50 04 50 04
00000040      50 04 40 04 40 04 30 04 30 04 30 04 20 04 20 04
00000050      10 04 10 04 10 04 10 04 00 04 F0 03 00 04 F0 03
                 (5)
```

1 Both channel numbers and sub channel numbers start at zero. (Waveform 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 multiple sample rates and low sample rates

If waveforms are acquired 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 waveform acquisition is fixed to a given number (integral multiple of 16). If the number becomes zero as a result of adjustment, 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)	Framed area (1)
4-byte data	Tick (4 bytes, in nanoseconds (0 to 999999999))	Framed area (2)

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	EA	78	CC	61	28	97	A7	25	EA	78	CC	61	28	C4	D8	26
00000010	EA	78	CC	61	38	18	0A	28	EA	78	CC	61	38	45	3B	29
00000020	EA	78	CC	61	28	EB	6C	2A	EA	78	CC	61	38	9F	9D	2B

(1) (2)

If the waveforms are acquired using external sampling, the sample count, not the time information, is saved.

8-byte data	Sample count (64-bit counter with the first data value set to 0)		
	4-byte data	Sample count (upper 4 bytes)	Framed area (1)
	4-byte data	Sample count (lower 4 bytes)	Framed area (2)

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	14	00	00	00	00	00	00	00	15	00	00	00
00000010	00	00	00	00	A4	01	00	00	00	00	00	00	A5	01	00	00
00000020	00	00	00	00	6C	02	00	00	00	00	00	00	6D	02	00	00

(1) (2)

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

4.2 Trigger Mode

Trigger mode using this API and DL950 works as follows.

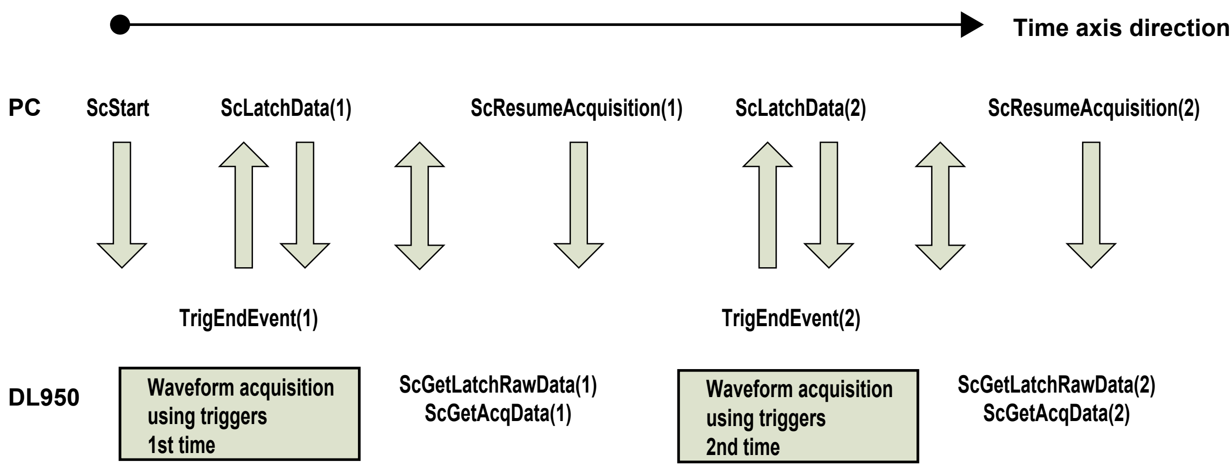
The DL950 acquires waveforms using triggers from when it receives a waveform acquisition start (ScStart) command until it receives a waveform acquisition stop (ScStop) command. Waveform data is stored in the instrument's acquisition memory. (The number of history entries that can be stored varies depending on the settings.)
While the waveform acquisition is in progress, execute latches (ScLatchData) and waveform acquisitions (ScGetAcqDataLength and ScGetAcqData) through the API.
There are two trigger modes: synchronous and asynchronous.

Synchronous mode

In synchronous mode, the DL950 acquires the next waveform after waiting for a response from the PC for the previous waveform acquisition. Use synchronous mode when you want to ensure that waveform data is transferred to the PC after waveform acquisition.
There are two ways to determine the completion of a waveform acquisition on the DL950. The first way is to implement the application to wait for trigger end events (ScAddEventListener(c++), ScAddCallBack(c#)). The second is to monitor the acquisition count (ScGetLatchAcqCount) periodically and decide that a waveform acquisition has been completed when the value is updated. Execute ScLatchData first and then ScGetLatchAcqCount.

The following figure shows the waveform acquisition sequence using triggers with trigger end events.

Synchronous trigger mode sequence

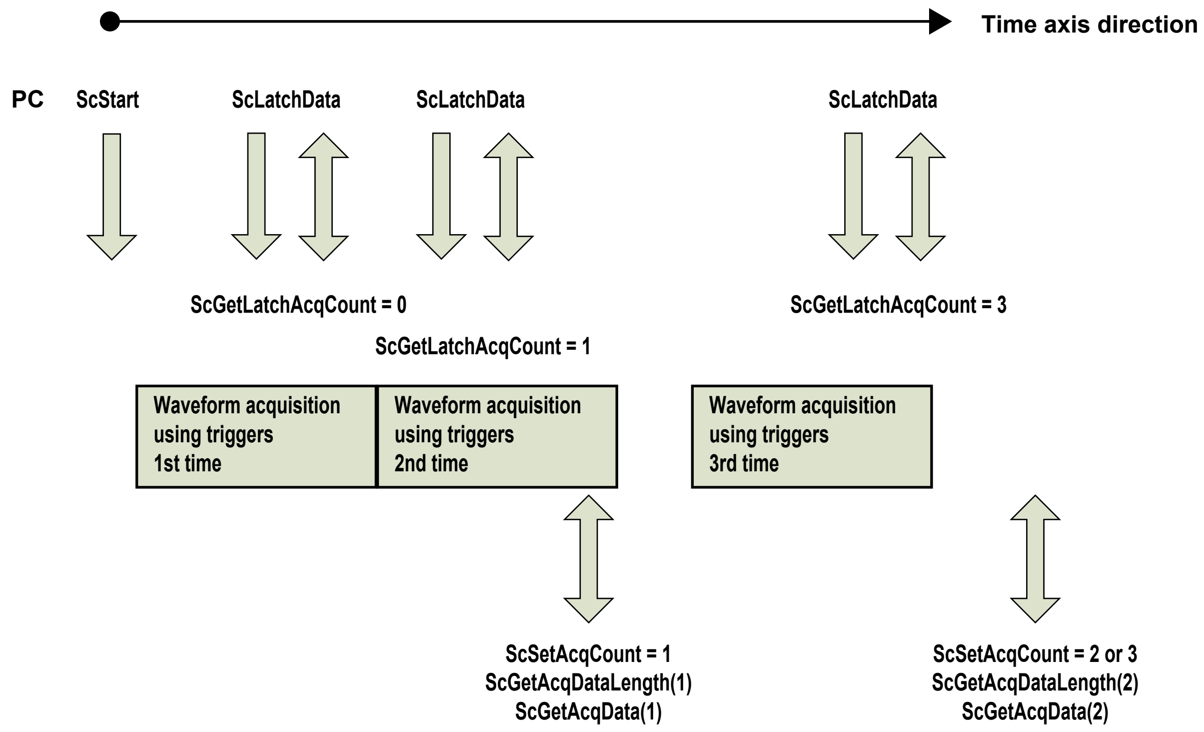


Asynchronous mode

In asynchronous mode, the DL950 continues waveform acquisition regardless of the command control on the PC.
The PC monitors the acquisition count by periodically executing ScLatchData and ScGetLatchAcqCount. When it verifies that the acquisition count has been updated, the acquisition count sent to the PC is set using ScSetAcqCount and waveform data is transferred.
Asynchronous mode is mainly used when you want to reduce the interval (dead time) between waveform acquisitions using triggers.
In asynchronous mode, the DL950 repeatedly acquires waveforms regardless of the waveform acquisition from the PC. If the measurement time (T/Div) is short, data may be overwritten due to waveform acquisition on the DL950 when waveforms are acquired on the PC by specifying an old acquisition count.
(This depends greatly on the number of history entries on the DL950. For details on the number of history entries, see appendix 5 in the *DL950 ScopeCorder Getting Started Guide*, IM DL950-03EN. In this case, waveform data cannot be acquired even when a waveform acquisition (ScGetAcqData) command is executed on the PC. The number of valid data points will be zero.

The following figure shows the waveform acquisition sequence in asynchronous mode.

Asynchronous trigger mode sequence



Waveform acquisition using external samples

When waveforms are acquired in trigger mode using external samples, timeSec and timeTick that are obtained using ScGetAcqData will contain sample counts, not time data, in the following format. (The handling is different from the timestamp format. For details on the timestamp format, see the explanation for free run mode.)

8-byte data	Sample count (64-bit counter of the first data value set to 0 after starting waveform acquisition)	
	4-byte data (timeSec)	Sample count (lower 4 bytes)
	4-byte data (timeTick)	Sample count (upper 4 bytes)

- The sample count is not a simple 64-bit integer value but a value divided into upper and lower bytes. Each value is in Little Endian format.
- The sample count is zero for the first data after waveform acquisition is started. In trigger mode, the first data obtained by this API may not necessarily be zero. (The sample count is continuously incremented until the first acquisition is completed after a trigger occurs. Therefore, the sample count at the data start point is obtained by subtracting the record length from the sample count at the point acquisition is completed.)

Further, because values are output for each calculation period for channels in timestamp format, the sample count included in the sub channel of time information in timestamp format may differ in range from the range obtained by ScGetAcqData for normal channels.
For details on the calculation period in timestamp format (power analysis), see appendix 13 in the the *DL950 ScopeCorder Features Guide*, IM DL950-01EN.