Script SDK Quick Start Guide

Topics in this document:

- Overview
- Creating executable program scripts
- ScriptSDK API functions by category
- ScriptSDK API functions by name
- ScriptSDK commands
- Sample ScriptSDK Programs

Overview

The Quantum Data Software Development Kit (SDK) provides Application Programming Interfaces (APIs) for two categories of automation: 1) custom images (Image SDK) and 2) executable scripts (ScriptSDK). Both of these SDKs use the C++ programming language.

The Script SDK API provides a programming interface for executing any set of commands in the 880 series command language. Additional functions allow for program control, user input/output, and debugging. ScriptSDK is a much more powerful alternative to the test sequence feature in the 881/882 generators. By providing full access to the 880 series command language, ScriptSDK allows for a broad range of control and functionality in executable scripts that reside within the generator. The executable program scripts can be executed via the ScriptSDK selection in the 882's TOOLS menu.

The entire ScriptSDK system includes components that reside on the personal computer (PC) and components that reside in the Quantum Data 881/882 generator. The PC components include:

- C++ compiler
- Libraries
- Header files
- Examples
- Templates
- Graphical User Interface

These are automatically downloaded from the Quantum Data website and installed on the PC the first time the Quantum Data SDK is run. The graphical user interface (GUI) program includes the text editor and menus to compile programs and load the executable object files into the generator. The GUI is a Java application that is loaded from the 881/882 generator each time the Quantum Data SDK is run.

The other components of this system lie within the embedded firmware in the Quantum Data 881/882 generator. This includes the Scriptrunner functionality for running the executable program scripts, as well as the proper linkages to allow the custom images to load.

Installation

The Quantum Data SDK can be accessed from the generator's home page beginning with firmware version 2.18.0. It is implemented as a Java application, and the Java Runtime Environment (JRE) is required on the host computer. JRE version 5 is recommended.

The SDK is launched from the Quantum Data SDK list item on the generator's home page. If the SDK is not yet installed, the user will be prompted through the installation process which includes connecting to the Quantum Data website. Therefore, an internet connection is required. If the older command-line version of the Image SDK is already installed on the system, it will be replaced by the Quantum Data SDK. The installation program can save the original SDK on the computer.

For first-time installation, an internet connection is required so that the automatic installation process can retrieve the installation package from the Quantum Data website. If the host computer does not yet have JRE, it can be downloaded from the java.com website, or via a link near the bottom of the generator's home page.

Image programs versus script programs

When the Quantum Data SDK is launched, the user has a choice of Image SDK or ScriptSDK, depending on the type of application to be developed. When the SDK is running, the user can freely switch between Image SDK and Script SDK at any time.

The Image SDK is used to create resolution-independent test patterns and other display images. When an image is loaded from the SDK into the generator via the Load & Execute menu selection, it is displayed immediately. An image load consists of a file transfer and the commands IMGP, IMGL, and IMGU. After loading the image via the SDK, the image is then available to be selected for display via browse mode.

The ScriptSDK is used to create executable programs that can be stored in the generator and run from the SDK menu. A ScriptSDK program can contain any commands in the 880 command language, as well as user menus, responses to key presses, numeric input, and output to the serial port.

Creating executable program scripts

This sections provides information and procedures for using the ScriptSDK application.

Getting Started

The following are the important points to keep in mind when creating executable scripts:

- Filing naming
- Return values on exit
- Enabling output
- Softkey functions

These are described in the following subsections.

Naming the file and the executable script

The executable script will have the same base name as the source file. This name must be lower-case, and it should be 8 characters or less in length. This name must be appended to "Script_" to name the function.

For example, for an executable script called "demo" the source file will be demo.cpp. The main function will be Script_demo(), and the resulting object file will be demo.o. The "load & execute" command will send this object file to the generator, and it will be visible in the Scripts menu as "demo."

Returning “true” on exit

The script's main function (e.g., Script_demo()) must return "true" on exit. As in any C or C++ program, it is okay to have multiple return points. Refer to the sample programs for examples.

Enabling output to a serial terminal

By executing the CIOY command sc.Exec("CIOY"); you can enable output to the serial terminal for debugging and/or status output via printf(). You can connect a simple serial terminal (such as Hyperterm running on a PC). Serial port settings default to 9600 baud, 8 bits, no parity, 1 stop bit, no handshake. The required DB-9 null cable is included with all 880 series generators.

Generator soft key functions during execution

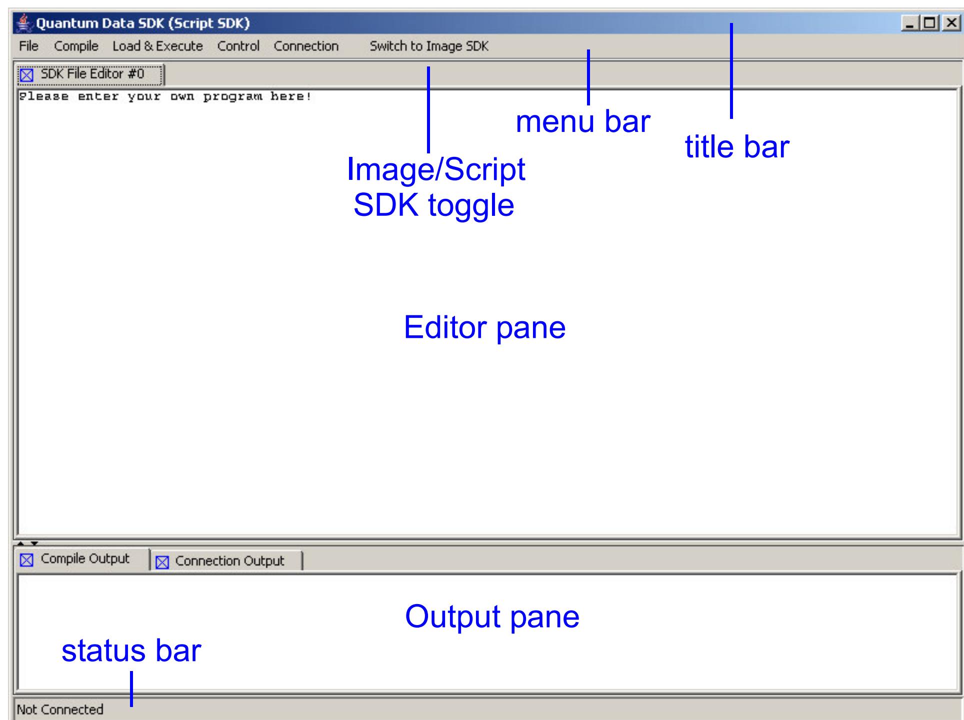
When a script is executing, the top right soft key on the 882 is assigned to the STOP function. If the STOP key is pressed, the sc.Canceled() method will return true. The script must check this state in order to detect whether the STOP key has been pressed. A script should always return true when exiting. See example programs testapi.cpp and tcan.cpp.

Some scripts may take over the top right soft key so it is not available for the STOP function. The example program testapi is an example of this. Therefore, the generator's OPTIONS key is always available to interrupt scripts that are running. After pressing the OPTIONS key, pressing the bottom left soft key (!Yes) will stop the script.

About the ScriptSDK main window

The Quantum Data SDK GUI is shown below. It is centered around the text editor. The title bar at the top indicates whether the program is currently set for ScriptSDK or ImageSDK. Beneath the title bar is the menu bar, featuring pull-down menu categories of File, Compile, Load&Execute, and Connection. The menu bar also contains a button for toggling between ScriptSDK and ImageSDK modes.

Under the menu bar is the main section of the GUI, the tabbed Editor pane. Multiple files can be edited, and each file has its own tab. Under the Editor pane is the output pane. The two tabs in this pane contain compiler output and connection output. Finally, the status bar at the bottom of the window shows the IP address of the 880 series generator that is currently connected.



ScriptSDK menu summary

Refer to the table below for a list of menu commands.

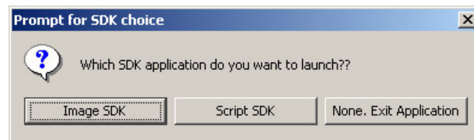
Menu	Command	Description
File	New	Opens a new empty file in the tabbed editor pane.
	New File from Template	Only active in ScriptSDK mode, this selection presents a dialog for opening a template file from the script_templates folder.
	Open	Presents a dialog for opening an existing source file. In ScriptSDK mode the default folder is scripts; in ImageSDK mode the default folder is examples.
	Close File	Closes the file in the current editor tab.
	Save	Saves the file in the current editor tab.
	Save as	Presents a dialog for saving the current file, allowing the user to choose a new filename and/or folder.
	Preference	Allows the user to choose between the Quantum Data SDK text editor and another text editor on the host PC
	Exit	Exits the Quantum Data SDK. This will also cause the web browser to immediately terminate, so it is recommended to instead use the web browser's back button to exit the SDK.
Compile	Current File	Compiles the file in the curent editor tab.
	Another File	Presents a dialog for choosing a source file to compile.
Load& Execute	Object File	Presents a dialog for choosing the object file to load to the generator.
Control	Pause	
	Terminate	
Connection	Connect to	Presents a dialog for entering the IP address of the generator to connect to. The dialog defaults to the generator that the application was launched from.
Switch to Image SDK		Allows the user to toggle between ImageSDK and ScriptSDK. This switch changes the default folders on the host PC for source and object files; it also changes the destination path on the generator for storing executable object files.

Starting ScriptSDK

To start ScriptSDK:

1. Connect to the generator through your Web browser. The generator Home page appears.

2. On the generator Home page, click **Script SDK**. The ScriptSDK Home page appears, and then a message appears asking you to choose whether you want to connect to ScriptSDK or ImageSDK.

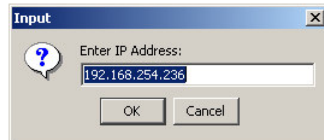


3. Click **Script SDK**. The ScriptSDK main window appears as shown below.

Creating, compiling, and executing a script

To start a scripting session:

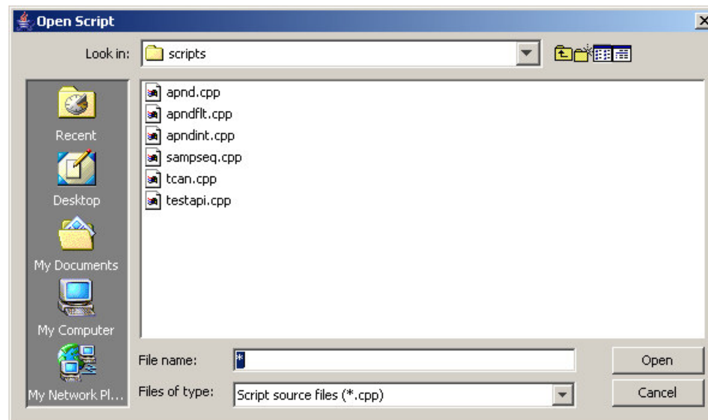
1. With the ScriptSDK main window open, click **Connection**, and then click **Connect to**. The Input dialog box appears.



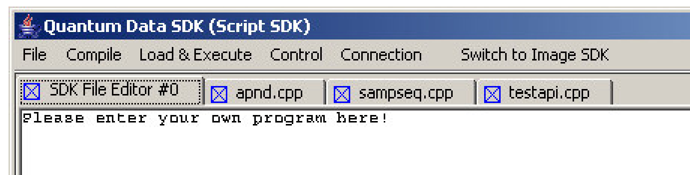
2. Enter the IP address of the generator you want to connect to. (The default address is the address of the generator from which you launched ScriptSDK.) Click **OK**. ScriptSDK attempts to connect to the generator and displays the message **Successfully Connected to Unit** when connected. Click **OK**.

To open an existing script:

1. Click **File**, and then click **Open**. The Open Script dialog box appears.



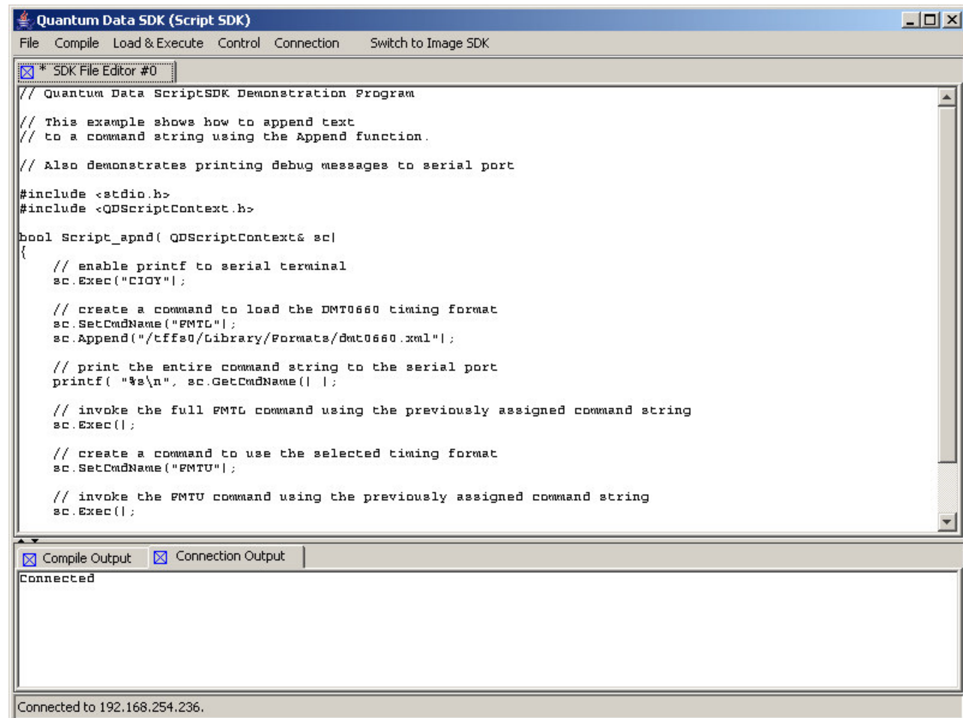
2. Select the script you want to open, and then click **Open**. The script appears in the Editor pane. You can have multiple scripts open at the same time. Each script appears as a separate tab in the Editor pane.



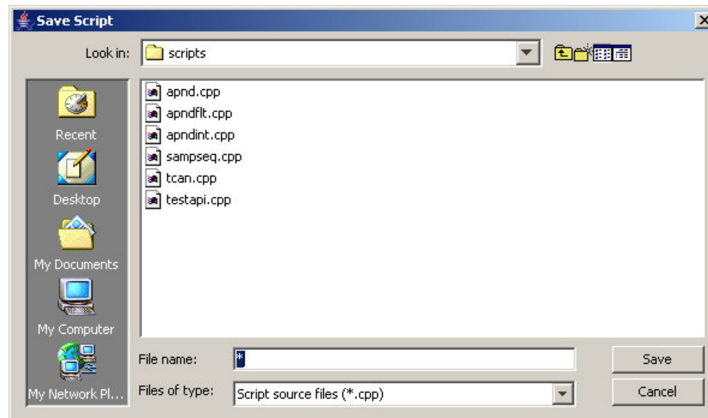
To create a new script:

1. Click **File**, and then click **New**. A new tab appears in the Editor pane.

2. Type the script in the Editor pane.



3. When you are finsihed, click **File**, and then click **Save**. The Save Script dialog box appears. In the **Filename** box, type a name for the script, and then click **Save**.

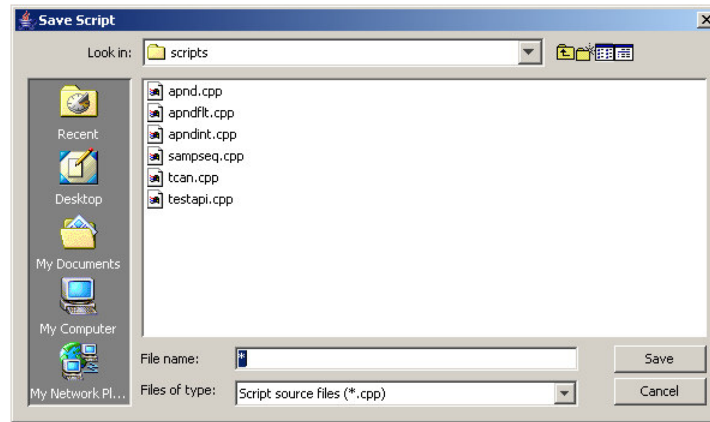


To create a new script using a template:

1. Click **File**, and then click **New File from Template**. The Select Template dialog box appears.

2. Select the template you want to use, and then click **Open**. The template appears in the Editor pane.

3. Modify the template as needed to match your requirements. When you are finsihed, click **File**, and then click **Save**. The Save Script dialog box appears. In the **Filename** box, type a name for the script, and then click **Save**.



To compile a script:

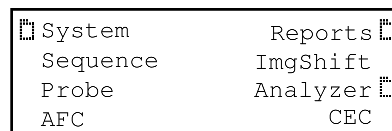
1. To compile a script, click the tab in the Editor Pane that contains the script you want to compile.
2. Click **Current File** from the Compile pull down menu. The file is compiled and any compiler messages appear in the Compile Output pane. The output of the compile process is an executable object file that is placed in Scripts folder, which is in the SDK folder on your 882 generator library directory.

To execute a script from the ScriptSDK GUI:

1. Click **Object File** from the **Load&Execute** pull down menu. A dialog box called Choose Object File dialog box appears.
2. Select the file you want to execute, and then click **Choose**. A message appears telling you if the execution was successful.
3. You can also execute a scrip

To execute a script from the 882 front panel:

1. Press the **Tools** key. The Tools menu appears on the generator's display as shown below.



2. Scroll down until you see the ScriptSDK selection item.

   The list of available scripts is presented.

```
Sampseq1      Sampseq2
Sampseq3      Sampseq4
```

3. Execute the desired script by pressing the adjacent soft key.

   The script that has been selected and is now running is shown along with an option to stop the script.

```
*Sampseq1           stop
```

To execute a script from the 882 command line:

1. Load the script file that you want to execute by entering the following command.

   ```
   SCRX:LOAD testapi.o //loads script object from current scripts path
   ```

   For a list of scripts, enter the following command:

   ```
   SCRX:LIST?
   ```

2. Execute the script by entering the following command:

   ```
   SCRX:EXEC   // executes currently loaded script
   ```

   To stop a script that is executing enter the following:

   ```
   SCRX:STOP   // stops execution of currently running script
   ```

ScriptSDK API functions by category

The ScriptSDK API functions are all public methods of the QDScriptContext class. They are presented in categories of control, command, response, front panel display, and operators.

Control functions

Command	Description
GetGC	Get the handle of TGC
Canceled	Status of script cancel
Pause	Calibrate generator (using self-calibration circuitry)

Command functions

Command	Description
SetCmdName	Set name of command to execute
GetCmdName	Get name of command to execute
Exec	Execute current command
Append	Append to command string
Reset	Reset the command string
SetDefaultTimeOut	Set default command timeout
GetDefaultTimeOut	Get default command timeout
SetCancelOnError	Set "cancel on error" status
GetCancelOnError	Get "cancel on error" status

Response functions

Command	Description
GetResponse	Get most recent response string
GetIntResponse	Get response as signed int
GetUIntResponse	Get response as unsigned int
GetDoubleResponse	Get response as double
GetRespLineCount	Get number of lines in response
ClearResponse	Reset the response
Failed	Get fail state of most recent command
Succeed	Get response state of most recent command
GetError	Get error value for failed command

Front panel functions

Command	Description
Write	Write a string to front panel display
InputInteger	Get integer user input via front panel
InputFloat	Get float user input via front panel
WaitForKeyPress	Wait for user to press a soft key
ClearLCD	Clear the front panel display

Operator functions

Command	Description
=	Assign name of command to be executed
+=	Append to command to be executed

ScriptSDK API functions by name

This subsections lists the ScriptSDK functions by name and provides details about how to use them.

Append

Class Command

Description Adds characters to an existing command string, created with a call to SetCmdName(). Allowable arguments are char pointer, integer, and double-precision float. In the case of an integer argument, the radix for the base of the resulting string can be specified optionally. The default radix is 10.

Command syntax sc.Append(const char* txt)

sc.Append(UINT32 value, UINT32 RADIX=10)

sc.Append(double value)

Example Refer to example programs apndflt.cpp, apndint.cpp, sampseq.cpp, testapi.cpp

Related commands Exec(), GetCmdName(), Reset(), SetCmdName,

Canceled

Class	Control
Description	Returns script cancel state. A script can be set to cancel state by pressing the STOP (upper right) soft key during execution, or by pressing the OPTIONS key and then the lower-left (!Yes) soft key to confirm. A script can also be set to cancel state if its CancelOnError status is true, and the script encounters an error while executing. To cancel script execution, the script must check the return value of Canceled(); if it is true, the script should immediately return a value of true.
Command syntax	sc.Canceled(void)
Example	Refer to example programs apndflt.cpp, apndint.cpp, sampseq.cpp, tcan.cpp, testapi.cpp
Related commands	Exec(), GetCmdName(), Reset(), SetCmdName

ClearLCD

Class	Front Panel
Description	Clears the entire display on the front panel of the generator
Command syntax	sc.ClearLCD(void)
Example	Refer to example program testapi.cpp
Related commands	Write(), InputInteger(), InputFloat()

ClearReponse

Class	Response
Description	Resets the command response string to a default empty state. This can be used to clear a response after it is queried; therefore the next non-empty response will be a new response.
Command syntax	sc.ClearResponse(void)
Related commands	GetResponse(), GetIntResponse(), GetUIntResponse(), GetDoubleResponse()

Exec

Class	Command
Description	Execute a command. Command can be pre-built using SetCmdName() and Append(), or a pointer to a command string can be passed. Command timeout in milliseconds can be specified optionally.
Command syntax	sc.Exec(void) sc.Exec(UINT32 timeout) sc.Exec(char* cmdString) sc.Exec(char* cmdString, UINT32 timeout)
Return type	UNIT32
Example	Refer to sample programs apnd.cpp, apndflt.cpp, apndint.cpp, sampseq.cpp, tcan.cpp, testapi.cpp
Related commands	SetCmdName( ), Append( ), GetCmdName( ), SetDefaultTimeOut(), GetDefaultTimeOut(), Command Reference in Appendix A of 881/882 Series User Guide

Failed

Class	Response
Description	Returns true if previous command execution failed.
Command syntax	sc.Failed(void)
Return type	bool
Related commands	Exec(), Succeed(), GetError()

GetBoolResponse

Class	Response
Description	Gets the most recent query response string and returns it as a bool. If the first character of the response string is "T" or "t" the return value is "true." If the first character of the response string is "F" or "f" the return value is "false." Otherwise, it will attempt to convert the string to unsigned long via strtoul(). If the result of the conversion is zero, the return value is "false." If the result is not zero, the return value is "true."
Command syntax	sc.GetBoolResponse(void)
Return type	bool
Example	Refer to sample program testapi.cpp
Related commands	GetResponse(), GetIntResponse(), GetUIntResponse(), GetDoubleResponse(), GetRespLineCount(), ClearResponse()

GetCancelOnError

Class	Command
Description	Returns CancelOnError status. Return value of true indicates that an execution error will cause Canceled() to be true. To terminate script execution on error, program code should check the state of Canceled(); if it is true the program should exit via "return true;"
Command syntax	sc.GetCancelOnError(void)
Return type	bool
Related commands	Canceled(), SetCancelOnError(), GetError()

GetCmdName

Class	Command
Description	Returns the current contents of the command string. The command string can be built using SetCmdName(), Append(), =, and +=.
Command syntax	sc.GetCmdName(void)
Return type	const char*
Example	Refer to sample programs apnd.cpp, apndflt.cpp, testapi.cpp
Related commands	SetCmdName(), Exec(), Append(), operator =, operator +=

GetDefaultTimeOut

Class	Command
Description	Returns the current timeout value for commands, in milliseconds.
Command syntax	sc.GetDefaultTimeOut(void)
Return type	UNIT32
Related commands	SetDefaultTimeOut()

GetDoubleResponse

Class	Response
Description	Gets the most recent query response string and returns it as a double-precision floating point value.
Command syntax	sc.GetDoubleResponse(void)
Return type	double
Example	refer to sample program testapi.cpp
Related commands	GetResponse(), GetIntResponse(), GetUIntResponse(), ClearResponse()

GetError

Class	Response
Description	Returns the error value of the previous command.
Command syntax	sc.GetError(void)
Return type	UNIT32
Example	refer to sample program testapi.cpp
Related commands	Exec(), Failed(), Succeed()

GetGC

Class	Control
Description	Returns a pointer to an instance of the graphics core object that provides dynamic linking with internal generator firmware.
Command syntax	GetGC(void)
Return type	TGC*

GetIntResponse

Class	Response
Description	Gets the most recent query response string and returns it as a signed 32-bit integer value.
Command syntax	sc.GetIntResponse(void)
Return type	INT32
Example	refer to sample program testapi.cpp
Related commands	GetResponse(), GetUIntResponse(), GetDoubleResponse(), ClearResponse()

GetRespLineCount

Class	Response
Description	Some query responses can consist of multiple lines of output. This method returns the number of lines of output from the most recent query.
Command syntax	sc.GetRespLineCount(void)
Return type	INT32
Example	refer to sample program testapi.cpp
Related commands	GetResponse(), GetIntResponse(), GetUIntResponse(), GetDoubleResponse, ClearResponse()

InputFloat

Class	Front Panel
Description	Presents a dialog for user input of a floating point value via front panel display and keys. Parameters allow specification of the displayed prompt, number of digits (whole and fractional,) and initial value. The prompt string is required. The whole and frac parameters are optional and both default to 4. The initValue parameter is optional and defaults to zero.
Command syntax	sc.InputFloat(const char*prompt, UINT32 whole, UINT32 frac, double initValue)
Return type	double
Example	refer to example program testapi.cpp.
Related commands	InputInteger(), ClearLCD()

InputInteger

Class	Front Panel
Description	Presents a dialog for user input of an integer value via front panel display and keys. Parameters allow specification of the displayed prompt, number of digits, and initial value. The prompt string is required. The numDigits parameter is optional and defaults to 4. The initValue parameter is optional and defaults to zero.
Command syntax	sc.InputInteger(const char* prompt, UINT32 numDigits, INT32 initValue)
Return type	INT32
Example	refer to example program testapi.cpp.
Related commands	InutFloat(), ClearLCD()

Pause

Class	Control
Description	Forces a delay in program execution, for specified number of milliseconds.
Command syntax	sc.Pause(UINT32 timeValue)
Return type	void
Example	refer to example programs apndflt.cpp, apndint.cpp, sampseq.cpp, tcan.cpp, testapi.cpp.

Reset

Class	Command
Description	Clears the command string; resets it to the default empty state.
Command syntax	sc.Reset(void)
Return type	void
Example	refer to example program testapi.cpp.
Related commands	SetCmdName(), Append(), GetCmdName()

SetCancelOnError

Class	Command
Description	Sets or clears the CancelOnError status. If set (with an argument of true) an error or failure will cause the Canceled() state to be set to true. To terminate script execution on error, program code should check the state of Canceled(); if it is true the program should exit via "return true;"
Command syntax	sc.SetCancelOnError(bool)
Return type	void
Example	refer to example program testapi.cpp.
Related commands	Canceled(), GetCancelOnError(), GetError()

SetCmdName

Class	Command
Description	Sets a command string to be executed later. The command string can be added to with Append(). The command can be executed with Exec().
Command syntax	sc.SetCmdName(const char*)
Return type	void
Example	refer to example programs apnd.cpp, apndflt.cpp. testapi.cpp
Related commands	refer to example programs apnd.cpp, apndflt.cpp. testapi.cpp

SetDefaultTimeOut

Class	Command
Description	Sets the current timeout value for commands, in milliseconds.
Command syntax	sc.SetDefaultTimeOut(UINT32)
Return type	void
Example	sc.SetDefaultTimeOut(5000); // commands will time out in 5 seconds
Related commands	GetDefaultTimeOut()

Succeed

Class	Response
Description	Returns true if previous command execution was successful.
Command syntax	sc.Succeed(void)
Return type	bool
Example	refer to example program testapi.cpp.
Related commands	Exec(), Fail(), GetError()

WaitForKeyPress

Class	Front Panel
Description	Delays program execution until a soft key is pressed. Optional argument indicates desired timeout in milliseconds. Void argument or zero will result in wait for ever. Returns a value representing the key pressed; one of the following: QD_KEY_LEFT_1 QD_KEY_LEFT_2 QD_KEY_LEFT_3 QD_KEY_LEFT_4 QD_KEY_RIGHT_1 QD_KEY_RIGHT_2 QD_KEY_RIGHT_3 QD_KEY_RIGHT_4
Command syntax	sc.WaitForKeyPress(void) sc.WaitForKeyPress(UINT32)
Return type	QDKeyId
Example	refer to example program testapi.cpp.
Related commands	Write(), ClearLCD()

Write

Class	Front Panel
Description	Write a string to the generator's front panel display. The display is 20 characters by 4 lines. Allowable values of column are in the range of 0 - 19. Allowable values of row are 0 - 3.
Command syntax	sc.Write(UINT32 column, UINT32 row, char* string)
Return type	void
Example	refer to example programs apndflt.cpp, testapi.cpp
Related commands	ClearLCD(), WaitForKeyPress(), InputFloat(), InputInteger()

Operator =

Class	Operator
Description	Assigns a string to the command string. The command string is represented by the pointer to the script context; sc. This is similar to SetCmdName().
Command syntax	sc = "string"

Example	sc = "SCRX:LOAD";// set the command string
	also refer to example program testapi.cpp.

Related commands	Operator +=, SetCmdName(), Append(), Exec()

Operator +=

Class	Operator
Description	Appends characters to the command string. The argument will be converted to ASCII character string if necessary. The command string is represented by a pointer to the script context; sc. This is similar to Append().
Command syntax	sc += "string" sc += char sc += double sc += UINT32

Example	sc += "testapi.o";// append filename to the command string
	also refer to example program testapi.cpp

Related commands	Operator =, Append(), SetCmdName(), Exec()

ScriptSDK commands

This subsection lists the commands related to the ScriptSDK that can be executed through the 882's command line interface. These can be executed through the serial port, a telnet session or through the command utility available through the 882 home page.

SCRX:LIST?

Class	Scriptrunner
Description	Lists the executable script object files currently residing in the generator
Query syntax	SCRX:LIST?
Query response	list of executable script object files
Example	`SCRX:LIST?  // list executable scripts`

SCRX:PATH

Class: Scriptrunner

Description: Sets the current path for storage of SDK scripts. The query returns the current script path name.

Command syntax: SCRX:PATH <*name*>

name

a valid MS-DOS compatible path and filename (8 characters minus any extension)

Query syntax: SCRX:PATH?

name

returns a valid MS-DOS compatible path and filename for the name of the path

Query response: path name

Example:

```
SCRX:PATH /tffs0/Library/Scripts // sets script path to default directory
```

SCRX:EXEC, SCRX:EXEC?

Class	Scriptrunner
Description	Executes the currently loaded script. The query returns the name of the currently executing script.
Command syntax	SCRX:EXEC
Query syntax	SCRX:EXEC?
Query response	path and name of currently executing script object file
Example	`SCRX:EXEC   // execute currently loaded script`

SCRX:LOAD, SCRX:LOAD?

Class: Scriptrunner

Description: Loads a script object file in preparation for execution. Script object file must already be stored in generator file system. If the load command also includes a directory path, then this command will also change the current scripts path (SCRX:PATH) to the specified path. The query returns the name of the currently loaded script.

Command syntax: SCRX:LOAD <*name*>

name

a valid script file, including ".o" extension which can optionally include a fully-qualified path

Query syntax: SCRX:LOAD?

Query response: name of currently loaded script object file

Example:

```
SCRX:LOAD testapi.o //loads script object from current scripts path
SCRX:LOAD /tffs0/library/userdata/testapi.o  //loads script object
                                               testapi.o from specified
                                               path, and also sets
                                               current script path to
SCRX:LOAD?         //returns name of currently loaded script object file
```

SCRX:KILL

Class
Scriptrunner

Description
Deletes an executable object file from the generator

Command syntax
SCRX:KILL name

name

a valid script file, including ".o" extension which can optionally include a fully-qualified path

Example

```
SCRX:KILL testapi.o // deletes script object testapi.o from current
                       scripts path
SCRX:KILL /tffs0/library/userdata/testimg.o // deletes file testimg.o
                                               from specified path
```

SCRX:STOP

Class	Scriptrunner
Description	Stops the execution of a script
Command syntax	SCRX:STOP
Example	`SCRX:STOP // stops execution of currently running script`

Sample ScriptSDK Programs

This subsections provides some example ScriptSDK scripts to help you understand how to create scripts for specific applications.

apnd.cpp

This example shows how to append text to a command string using the Append function and demonstrates printing debug messages to a serial port.

```
#include <stdio.h>
#include <QDScriptContext.h>
bool Script_apnd( QDScriptContext& sc)
{
// enable printf to serial terminal
    sc.Exec("CIOY");
// create a command to load the DMT0660 timing format
    sc.SetCmdName("FMTL");
    sc.Append("/tffs0/Library/Formats/dmt0660.xml");
// print the entire command string to the serial port
    printf( "%s\n", sc.GetCmdName() );
// invoke the full FMTL command using the previously assigned command
string
    sc.Exec();
// create a command to use the selected timing format
    sc.SetCmdName("FMTU");
// invoke the FMTU command using the previously assigned command string
    sc.Exec();
return true;
}
```

apndflt.cpp

This example shows how to add a floating point numeric value to a command string using the Append function and also demonstrates writing to the 882 front panel display.

```
#include <stdio.h>
#include <QDScriptContext.h>
bool Script_apndflt( QDScriptContext& sc )
{
// For RGB: step the analog video signal swing from 0.1 volts
// to 1.0 volts in steps of 0.10 volts, pausing at each level
    double aswing;
// select a safe format, VGA
    sc.Exec("FMTL /tffs0/Library/Formats/DMT0660.xml");
    // use this selected format
    sc.Exec ("FMTU");
    // select a colorful image for this example
    sc.Exec("IMGL /Cache0/Images/colorbar.img");
    // select RGB video out
    sc.Exec("XVSI 9");
    // enable RGB video
    sc.Exec("AVST 2");
    // apply the above image and interface selections
    sc.Exec("ALLU");
    // step the analog video signal swing from 0.1 volts to 1.0
    // volts in steps of 0.10 volts, pausing at each level
    for(aswing=0.10; aswing<1.01; aswing+=0.10)
    {
        // build the command to set analog video swing
        sc.SetCmdName("AVSS");
        sc.Append(aswing);
// write the command to the bottom of the front-panel display
        sc.Write( 0,3,(char *)sc.GetCmdName() );
// issue the command to set analog video swing
        sc.Exec();
// use the selected format parameters
        sc.Exec("FMTU");
// pause for 10 seconds at each level
        sc.Pause(10000);
// this will cause script to immediately exit if STOP is pressed
        if (sc.Canceled())
return true;
}
    return true;
}
```

apndflt.cpp

This example demonstrates appending an integer to a command string. This example will sequentially test 3 different video interfaces: HDMI, S-video, and composite.

```
#include <stdio.h>
#include <QDScriptContext.h>
bool Script_apndint( QDScriptContext& sc )

    UINT32 formatnum;

    sc.Exec("CIOY");        // enable printf to serial terminal

    // first load an image to use for this example:
    sc.Exec("IMGL /Cache0/Images/colorbar.img");
    // display the image
    sc.Exec("IMGU");

    // step through 3 interfaces: 4=HDMI-H, 5=SVideo, 6=CVBS
    for(formatnum=4; formatnum<=6; formatnum++)
    {
        // build command string to set video output interface
        // XVSI specifies which video output interface to use
        sc.SetCmdName("XVSI ");

        // use Append function to convert formatnum to character and
insert it into
        // command string.  Base-10 conversion.
        // For this example, it will be either 4, 5, or 6.
        sc.Append(formatnum, 10);

        // print the entire command string to the serial port
        printf( "%s\n", sc.GetCmdName() );
        // execute the completed XVSI command
        sc.Exec();

        // This switch will select a video format that is appropriate
        // for the output interface selected above
        switch(formatnum)
        {
            case 4:
                printf("Now testing HDMI...\n");
                // load a format that is compatible with HDMI interface
                sc.SetCmdName("FMTL /tffs0/Library/Formats/dmt0660.xml");
                break;

            case 5:
                printf("Now testing SVideo...\n");
                // load a format that is compatible with SVideo interface
                sc.SetCmdName("FMTL /tffs0/Library/Formats/ntsc.xml");
                break;

            case 6:
```

```
                printf("Now testing CVBS...\n");
                // load a format that is compatible with CVBS interface
                sc.SetCmdName("FMTL /tffs0/Library/Formats/ntsc.xml");
        }
        // invoke the selected format
        sc.Exec();

        // use the format that was selected above
        sc.Exec("FMTU");

        // delay for 10 seconds on each interface
        sc.Pause(10000);

        // this will cause script to immediately exit if STOP is pressed
        if (sc.Canceled())
return true;
    }
    return true;
}
```

sampseq.cpp

```
// Quantum Data ScriptSDK Demonstration Program

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include <stdio.h>

#include <QDScriptContext.h>

//~~~~~~~~~~~~~~~~~~~~~~

/******************************************************************

*  Script_sampseq

******************************************************************
```

This is a conversion of "Sample Sequence" from chapter 8 of the 881/882 User Guide. This demonstrates how a sequence can be converted to an SDK script. Note that arguments are not always the same as in the sequence XML files, as shown in the XVSG conversions below.

In ScriptSDK, commands have the same syntax as if they were being entered at a command prompt. For full details on command syntax, refer to the Command Reference in Appendix A of the 881/882 User Guide available at: http://www.quantumdata.com/downloads/index.asp

Note: *Note: this script (and the sequence that it evolved from) expects you to have previously selected a compatible interface, such as VGA or HDMI.*

```
******************************************************************

*  Usage:

Script_sampleseq( QDScriptContext& sc );

*

*  Arguments:

*     sc - the script context being executed on.

******************************************************************/

bool Script_sampseq( QDScriptContext& sc )

{

//<?xml version="1.0" encoding="UTF-8" ?>

//<DATAOBJ>

   //<HEADER TYPE="SEQ" VERSION="1.0" ></HEADER>

   //<DATA>

   //<MODE>STEP</MODE>
```

```
//<STEP>
//<FMT>/tffs0/Library/Formats/DMT0660.xml</FMT>
sc.Exec("FMTL /tffs0/Library/Formats/DMT0660.xml");
sc.Exec("FMTU");
//<IMG>/Cache0/Images/Acer1.img</IMG>
sc.Exec("IMGL /Cache0/Images/Acer1.img");
sc.Exec("IMGU");
//<DELY>+3.0000000E+00</DELY>
sc.Pause(3000);
//<VERS>0</VERS>
sc.Exec("IVER 0");
//<XVSG>7</XVSG>
sc.SetCmdName("XVSG ");
sc.Append("1 1 1");
sc.Exec();

if (sc.Canceled()) {
    return true;
}
printf("Check Point 1\n");
//</STEP>
//<STEP>
//<IMG>/Cache0/Images/Cubes.img</IMG>
sc.Exec("IMGL /Cache0/Images/Cubes.img");
sc.Exec("IMGU");
//<DELY>+4.0000000E+00</DELY>
sc.Pause(4000);

if (sc.Canceled()) {
    return true;
}
printf("Check Point 2\n");
```

```
//</STEP>
//<STEP>
//<IMG>/Cache0/Images/Linearty.img</IMG>
sc.Exec("IMGL /Cache0/Images/Linearty.img");
sc.Exec("IMGU");
//<DELY>+5.5000000E+00</DELY>
sc.Pause(5500);

if (sc.Canceled()) {
   return true;
}
printf("Check Point 3\n");
//</STEP>
//<STEP>
//<FMT>/tffs0/Library/formats/DMT0860.xml</FMT>
sc.Exec("FMTL /tffs0/Library/Formats/DMT0860.xml");
sc.Exec("FMTU");
//<IMG>/Cache0/Images/RampX.img</IMG>
sc.Exec("IMGL /Cache0/Images/RampX.img");
sc.Exec("IMGU");
//<DELY>+3.0000000E+00</DELY>
sc.Pause(3000);

if (sc.Canceled()) {
   return true;
}
printf("Check Point 4\n");
//</STEP>
//<STEP>
//<XVSG>1</XVSG>
sc.Exec("XVSG 0 0 1");
```

```
    //</STEP>
    //<STEP>
    //<XVSG>2</XVSG>
    sc.Exec("XVSG 0 1 0");

    //</STEP>
    //<STEP>
    //<XVSG>4</XVSG>
    sc.Exec("XVSG 1 0 0");

    //</STEP>
    //<STEP>
    //<XVSG>7</XVSG>
    sc.Exec("XVSG 1 1 1");

    if (sc.Canceled()) {
        return true;
    }
    printf("Check Point 5\n");
    //</STEP>
    //<STEP>
    //<FMT>/tffs0/Library/Formats/DMT1060.XML</FMT>
    sc.Exec("FMTL /tffs0/Library/Formats/DMT1060.xml");
    sc.Exec("FMTU");
    //<IMG>/Cache0/Images/Master.img</IMG>
    sc.Exec("IMGL /Cache0/Images/Master.img");
    sc.Exec("IMGU");
    //<DELY>+5.0000000E+00</DELY>
    sc.Pause(5000);
    return true;
}
```

sampseq.cpp

This example delays and does nothing except waits to be cancelled, then reports to a serial terminal whether it was cancelled.

```
#include <stdio.h>
#include <QDScriptContext.h>
bool Script_tcan( QDScriptContext& sc )
{
    sc.Exec("CIOY");        // enable printf to serial terminal
    printf("Pause for 10 seconds\n");
    sc.Pause( 10000 );                            // pause for 10 seconds

    if (sc.Canceled())
    {
        printf("Script was canceled\n");
    }
    else
    {
        printf("Script completed\n");
    }
    return true;
}
```

testapi.cpp

This example demonstrates the following functions of the API:

- Sending commands and getting responses back
- Pause()
- Multi-line response
- Operators “=” and “+=”
- QDLcd associated API commands: InputInteger(), InputFloat(), and WaitForButtonPress()
- GetUIntResponse()
- GetIntResponse()
- GetDoubleResponse()
- GetBoolResponse()
- Detecting an invalid command

```
#include <QDScriptContext.h>
#include <stdio.h>
bool Script_testapi( QDScriptContext& sc )
{
    // ==========================================================
    // Demonstration of sending command and getting response back
    // ==========================================================
    // one way to execute a command/query
    sc.Exec("CIOY");        // enable printf to serial terminal
    // another way to execute a command/query
    sc.SetCmdName("VERF?"); // query generator firmware version
    sc.Exec();              // execute the command
    printf("The return value of command \"%s\" is %s\n", sc.GetCmdName(),
sc.GetResponse());
    // ==========================================================
    // Demonstration of Pause()
    // ==========================================================
    printf("Pause for 5 seconds\n");
    sc.Pause( 5000 );
    if (sc.Canceled()) {
        return true;
    }
    printf("\nWake up from Pause(5000)\n");
    sc.SetCmdName("VERG?"); // query generator gateware versions
    sc.Exec();              // execute the command
    if (sc.Succeed()) {     // was previous command successful?
        printf("\nThe return value of command \"%s\" is %s\n",
sc.GetCmdName(), sc.GetResponse());
    }
```

```
    else
        printf("\nThere is no response.\n");
    // ============================================================
    // Demonstration of multi-line respones
    // ============================================================
    if (sc.Canceled()) {
        return true;
    }
    // query the first 10 test images in the generator
    sc.SetCmdName("IMGQ?");
    sc.Append(" 1 10");
    sc.Exec();
    printf("\nThe response of command \"%s\" has %d lines\n",
sc.GetCmdName(), sc.GetRespLineCount());
    printf("\nThe 9th line of response is %s\n", sc.GetResponse(9));
    for ( UINT32 index=1; index<sc.GetRespLineCount()+1; index++ ) {
        printf("Response line %d is: %s \n", index,
sc.GetResponse(index));
    }
    // ============================================================
    // Demonstration of Operator "=" and "+=" overloading
    // ============================================================
    sc.Reset();
    if (sc.Canceled()) {
        return true;
    }
    sc  = "SCRX:";
    sc += "LIST";
    sc += "?";
    sc.Exec();
    printf("\nThe return value of command \"%s\" is %s\n",
sc.GetCmdName(), sc.GetResponse());
    // ============================================================
    // Demonstration of QDLcd associated APIs, InputInteger() &
InputFloat()
    // ============================================================
    if (sc.Canceled()) {
        return true;
    }
    // Test "InputInteger( const char* prompt )" API
    INT32 test = sc.InputInteger( "My test:" );
    printf("\nThe input of my test is: %d\n", test);
    if (sc.Canceled()) {
        return true;
    }
    test = sc.InputInteger( "Input another one:" );
    printf("\nThe input of second input is: %d\n", test);
    if (sc.Canceled()) {
        return true;
    }
    test = sc.InputInteger( "Test limity:", 3, 300 );
    printf("\nThe input of test limit is: %d\n", test);
    if (sc.Canceled()) {
```

```
        return true;
    }
    double test2 = sc.InputFloat( "Test Float:" );
    printf("\nThe input of Test Float is: %f\n", test2);
    // ===========================================================
    // Demonstration of QDLcd associated APIs, WaitForButtonPress()
    // ===========================================================
    if (sc.Canceled()) {
        return true;
    }
    sc.ClearLCD();
    sc.Write(1, 1, "Test WaitForButtonPress()");
    sc.Write(0, 3, "<-Yes4");
    sc.Write(16, 3, "No->");
    QDKeyId pressedKey = sc.WaitForKeyPress(5000);
    if ( pressedKey == QD_KEY_LEFT_4 ) {
        printf("\nUser pressed \"Yes\"\n");
    }
    else if ( pressedKey == QD_KEY_RIGHT_4 ) {
        printf("\nUser pressed \"No\"\n");
    }
    else
    {
        printf("\nPressed Invalid Key\n");
    }
// ===========================================================
    // Demonstration of GetUIntResponse()
    // ===========================================================
    if (sc.Canceled()) {
        return true;
    }
    sc.SetCmdName("DPTR?");
    //sc.ClearScreen();
    sc.Exec();
    if (sc.Succeed()) {
        printf("\nTest GetUIntResponse(): The return value of command
\"%s\" is %d\n", sc.GetCmdName(), sc.GetUIntResponse());
    }
    else
        printf("\nCommand execution failed.\n");
    // ===========================================================
    // Demonstration of GetIntResponse()
    // ===========================================================
    if (sc.Canceled()) {
        return true;
    }
    sc.SetCmdName("HRES?");
    //sc.ClearScreen();
    sc.Exec();
    if (sc.Succeed()) {
        printf("\nTest GetIntResponse(): The return value of command
\"%s\" is %d\n", sc.GetCmdName(), sc.GetIntResponse());
    }
```

```
    else
        printf("\nCommand execution failed.\n");
    // ============================================================
    // Demonstration of GetDoubleResponse()
    // ============================================================
    if (sc.Canceled()) {
        return true;
    }
    sc.SetCmdName("PRAT?");
    //sc.ClearScreen();
    sc.Exec();
    if (sc.Succeed()) {
        printf("\nTest GetDoubleResponse(): The return value of command
\"%s\" is %f\n", sc.GetCmdName(), sc.GetDoubleResponse());
    }
    else
        printf("\nCommand execution failed. Error is %d\n",
sc.GetError());
    // ============================================================
    // Demonstration of GetBoolResponse()
    // ============================================================
    if (sc.Canceled()) {
        return true;
    }
    sc.SetCmdName("PDAX:RPTG?");
    //sc.ClearScreen();
    sc.Exec();
    if (sc.Succeed()) {
        printf("\nTest GetBoolResponse(): The return value of command
\"%s\" is %d\n", sc.GetCmdName(), sc.GetBoolResponse());
    }
    else
        printf("\nCommand execution failed.\n");
    // ============================================================
    // Demonstration of Pause()
    // ============================================================
    printf("Pause for 6 seconds\n");
        sc.Pause( 6000 );
    if (sc.Canceled()) {
        return true;
    }
    printf("\nWake up from Pause(6000)\n");
    if (sc.Canceled()) {
        return true;
    }
    sc.SetCmdName("VERG?");
    //sc.ClearScreen();
    sc.Exec();

    if (sc.Succeed()) {
        printf("\nThe return value of command \"%s\" is %s\n",
sc.GetCmdName(), sc.GetResponse());
    }
```

```
    else
        printf("\nCommand execution failed.\n");
// ============================================================
    // Demonstration of detecting invalid command
    // ============================================================
    // First, set CancelOnError to be FALSE, the test should continue to
next
    sc.SetCancelOnError( false );
    sc.SetCmdName("VERP?");
    sc.Exec();
    printf("\nThe error status of the invalid command %s is %d\n",
sc.GetCmdName(), sc.GetError());
    if (sc.Canceled()) {
        printf("The test exited because the command is not valid\n");
        return true;
    }
    else
    {
        printf("Script not cancelled.  Execution should continue\n");
    }
    // Then, set CancelOnError to be TRUE, the test should continue to
next
    sc.SetCancelOnError( true );
    sc.SetCmdName("VERP?");
    sc.Exec();
    printf("\nThe error status of the invalid command %s is %d\n",
sc.GetCmdName(), sc.GetError());
    if (sc.Canceled()) {
        printf("The test exited because the command is not valid\n");
        return true;
    }
    else
    {
        printf("Script not cancelled.  Execution should continue\n");
    }
    return true;
}
```