

# .NET programming interface for R&S®GTSL and R&S®EGTSL

## Application Note

### Products:

- R&S®CompactTSVP   ▪ R&S®PowerTSVP
- R&S®GTSL           ▪ R&S®EGTSL

This application note describes the use of R&S®GTSL and R&S®EGTSL in a .NET programming environment. It shows the user how libraries and drivers generated using the C programming language can be used in applications based on .NET technology.

### Note:

Please find the most up-to-date document on our homepage <http://www.rohde-schwarz.com/appnote/1SE001>

This document is complemented by software. The software may be updated even if the version of the document remains unchanged

# Table of Contents

<b>1</b>	<b>Introductory Note</b> .....	<b>3</b>
<b>2</b>	<b>Introduction</b> .....	<b>4</b>
<b>3</b>	<b>The GTSL-.Net Wrapper Classes</b> .....	<b>5</b>
<b>3.1</b>	<b>Technology</b> .....	<b>5</b>
3.1.1	Adding .NET Wrapper Classes.....	5
3.1.2	Tips for Creating GTSL-.NET Wrapper Classes.....	7
<b>3.2</b>	<b>GTSL Device Driver Example: The Pfg Class</b> .....	<b>10</b>
<b>3.3</b>	<b>GTSL Library Example: The ResMgr Class</b> .....	<b>14</b>
<b>4</b>	<b>GTSL Programming Examples</b> .....	<b>17</b>
4.1	Direct Driver Calls.....	17
4.2	Combi Test .....	20
<b>5</b>	<b>Installation Procedure</b> .....	<b>22</b>
<b>6</b>	<b>Development Tools</b> .....	<b>23</b>
6.1	Compiler csc.exe from the .NET Framework SDK.....	23
6.2	Visual Studio Community .....	23
<b>7</b>	<b>Resources</b> .....	<b>24</b>

# 1 Introductory Note

This application note uses the following abbreviations for Rohde & Schwarz instruments and software products:

Unless a distinction must be made between the two, both the R&S<sup>®</sup>CompactTSVP TS-PCA3 production test platform and the R&S<sup>®</sup>PowerTSVP TS-PWA3 production test platform are referred to as the TSVP (test system versatile platform).

Unless a distinction must be made between the two, both R&S<sup>®</sup>GTSL (generic test software library) and R&S<sup>®</sup>EGTSL (enhanced generic test software library) are referred to as GTSL.

## 2 Introduction

The TSVP production test platform is a standardized, modular platform for program-controlled testing of modules and instruments during production or in the lab.

GTSL is a collection of software libraries and the supporting software device drivers. These libraries control the TSVP hardware modules for performing test tasks such as measurements, wiring configurations or signal generation.

GTSL was generated using the C/C++ programming language. To permit GTSL to be used from programming languages such as C#, Visual Basic .NET and other languages based on the .NET framework, Rohde & Schwarz provides a software layer to link the two programming worlds.

This software layer is presented in this application note. It is referred to here as the GTSL-.NET wrapper classes. This application note explains how the existing GTSL software libraries and software instrument drivers are encapsulated so that they can be accessed from the .NET world. Finally, it explains how programming examples that are part of the GTSL installation can be used in a slightly modified form in the .NET world under C#.

## 3 The GTSL-.Net Wrapper Classes

### 3.1 Technology

In order to create a link layer between code written in C and code run in a .NET environment, the 'unmanaged C code' must be embedded in the 'managed code' of a .NET-based programming language such as C#. Running 'managed code' requires the use of a runtime library such as that made available by the .NET framework, whereas the compiled 'unmanaged C code' is built into the machine language. Special care must be taken to replace the parameter interface for the embedded C functions with equivalent, i.e. compatible data types in the .NET world.

The GTSL-.NET wrapper classes for GTSL libraries and GTSL device drivers are provided to the user as the GTSL.dll class library in .NET format, compatible with .NET framework 3.5. The .dll extension is the only reminder of the old Win32 DLLs in GTSL, although the class library is not compatible with these in any way. The source code for all wrapper classes and the corresponding Microsoft Visual Studio solution are included.

For each of a large number of GTSL libraries and GTSL device drivers, the class library for the GTSL-.NET wrapper layer includes a class with public methods representing the functions provided by that library or device driver.

#### 3.1.1 Adding .NET Wrapper Classes

In some situations, the user will have test libraries created in C or GTSL libraries for which no GTSL-.NET wrapper classes exist. This section describes how to create a .NET wrapper class to allow the user to make these libraries available in a .NET environment. This newly created wrapper class can be included in the published source code in the GTSL.dll.

Wrapper classes insert functions from an existing DLL created in the C programming language by means of `DllImport`:

```

/// <summary>
/// class to define DLL imports of GTSL device driver
/// </summary>
private class PInvoke
{
    ...

    [DllImport("rspfg.dll", EntryPoint = "rspfg_ConfigureOperationMode",
              CallingConvention = CallingConvention.StdCall)]
    public static extern int ConfigureOperationMode(HandleRef Instrument_Handle,
                                                  string Channel_Name,
                                                  int Operation_Mode);

    ...
}

```

GTSL.dll - source code module: Pfg.cs

The `PInvoke` class lists all functions of a C GTSL device driver (in this example the device driver for the R&S TS-PFG signal generator), which are then published to the .NET world using `DllImport`. The function `rspfg_ConfigureOperationMode` was used here as being representative for all functions of the driver. Its parameter list includes the three parameter types `HandleRef`, `string` and `int`.

The header file of the C device driver contains the `rspfg_ConfigureOperationMode` function with the following parameter list:

```
ViStatus _VI_FUNC rspfg_ConfigureOperationMode (ViSession    vi,
                                              ViConstString channelName,
                                              ViInt32      operationMode);
```

rspfg.dll - source code module: rspfg.h

The type definitions for the parameter list is taken from the National Instruments VISA software layer (VISA: Virtual Instrument Software Architecture). The VISA type definitions correspond to the ANSI-C data types as well as to the C# data types used in the .NET wrapper classes as listed in the following table. Instead of the C# data types, the wrapper classes could just as easily use the equivalent .NET data types.

NI VISA	ANSI-C	C#	.NET
<b>ViSession*</b>	unsigned long*	out IntPtr (1)	out IntPtr (1)
<b>ViSession</b>	unsigned long	HandleRef (2)	HandleRef (2)
<b>ViConstString</b>	const char*	string	String
<b>ViRsrc</b>	char*	string	String
<b>ViStatus</b>	signed long	int	Int32
<b>ViStatus*</b>	signed long *	out int	out Int32
<b>ViAttr</b>	unsigned long	uint	UInt32
<b>ViString</b>	char*	string	String
<b>ViChar[]</b>	char[]	StringBuilder	StringBuilder
<b>ViBoolean</b>	unsigned short	ushort	UInt16
<b>ViBoolean*</b>	unsigned short	out ushort	out UInt16
<b>ViInt16</b>	signed short	short	Int16
<b>ViInt16*</b>	signed short*	out short	out Int16
<b>ViUInt32</b>	unsigned long	uint	UInt32
<b>ViUInt32*</b>	unsigned long*	out uint	out UInt32
<b>ViInt32</b>	signed long	int	Int32
<b>ViInt32*</b>	signed long *	out int	out Int32
<b>ViReal64</b>	double	double	Double
<b>ViReal64*</b>	double*	out double	out Double
<b>ViReal64[]</b>	double[]	double[]	Double[]

- (1) `IntPtr` is a handle to a resource such as is returned by a GTSL device driver when the C function `<driver_name>_InitWithOptions (...)` is called.
- (2) `HandleRef` is an object that contains a handle to a resource (`IntPtr`). The Invoke mechanism passes it to the unmanaged code, i.e. to the C functions of the GTSL device driver when they are called.

All C functions of a GTSL device driver added to the private class `PInvoke` by means of `DllImport` are again encapsulated in public methods in order to provide error handling and also to simplify the parameter signature somewhat. These methods can then be called by users in their own applications.

All GTSL-.NET wrapper classes that control the TSVP hardware modules are derived from the `TsvpInstrument` base class. This base class defines abstract methods that must be included in all wrapper classes. The wrapper class uses methods of the same name to overwrite these with the `override` modifier.

The `TsvpInstrument` base class additionally defines data types in the form of enums that are used by all derived wrapper classes.

The base class also includes classes for error handling and administration of the attributes for the individual GTSL device drivers and their wrappers. Attributes that are found in all wrappers are included here in the attribute dictionary, which lists the attributes such as name, data type, reuse in various channels, read/write access and a brief description. The attribute dictionary can additionally include special attributes for a specific GTSL-.NET wrapper class.

### 3.1.2 Tips for Creating GTSL-.NET Wrapper Classes

The .NET programming environment handles several items more elegantly than was the case in the C world. Some brief examples are provided here.

#### 3.1.2.1 Length of array parameters

The C language GTSL device drivers and libraries contain functions that receive an array of elements in the parameter list. If the length of the array is variable instead of fixed, an additional parameter must be specified here with the length of the array.

For example, the `rspfg_CreateArbWaveform` function in the C device driver for the R&S TS-PFG signal generator:

```
ViStatus _VI_FUNC  rspfg_CreateArbWaveform (ViSession vi,
                                           ViInt32  wfmSize,
                                           ViReal64  wfmData[],
                                           ViInt32*  wfmHandle);
```

rspfg.dll - source code module: rspfg.h

The `wfmSize` parameter defines the number of elements in the array `wfmData[]`.

The length parameter is no longer required in the corresponding method in the GTSL-.NET wrapper. The length of the array is determined via the `Length` method in the array object and thus transferred in the call of the corresponding `PInvoke` method:

```

public int CreateArbWaveform(double[] Waveform_Data_Array, out int Waveform_Handle)
{
    int pInvokeResult = PInvoke.CreateArbWaveform(this._handle, Waveform_Data_Array.Length,
                                                Waveform_Data_Array, out Waveform_Handle);
    return this.TestForError(pInvokeResult);
}

```

GTSL.dll - source code module: Pfg.cs

### 3.1.2.2 Using the StringBuilder class: size adjustments

Several GTSL device driver functions return information to the user in the form of one or more strings. One example is the C function named `<driver_name>_revision_query`, which returns version information about the software device driver and about the firmware for the addressed hardware. The call to this function must include two strings with a minimum length of 256 characters. If the strings transferred to this function in a C application are too short, it can result in difficult-to-find errors or system crashes.

C# provides a convenient security mechanism for preventing this problem. It ensures that in all GTSL-.NET wrapper classes, objects of the `StringBuilder` class that the user transfers to the corresponding methods either provide sufficient memory or are increased in size as needed.

The base class `TsvpInstrument` provides the `EnsureStringBuilderCapacity` method for this purpose. It checks the size of the transferred `StringBuilder` object. If not sufficient, the capacity of the object is increased:

```

internal void EnsureStringBuilderCapacity(ref StringBuilder sb, int minimumSize)
{
    if (sb.Capacity < minimumSize)
    {
        sb.EnsureCapacity(minimumSize);
    }
}

```

GTSL.dll - source code module: TsvpInstrument.cs

The wrapper class derived from the base class uses this method to check the `StringBuilder` objects that the user transfers to a given method. The example below uses the `RevisionQuery` method:

```

public override int RevisionQuery(ref StringBuilder Instrument_Driver_Revision,
                                ref StringBuilder Firmware_Revision)
{
    this.EnsureStringBuilderCapacity(ref Instrument_Driver_Revision, 256);
    this.EnsureStringBuilderCapacity(ref Firmware_Revision, 256);
    int pInvokeResult = PInvoke.revision_query(this._handle,
                                                Instrument_Driver_Revision,
                                                Firmware_Revision);
    return this.TestForError(pInvokeResult);
}

```

GTSL.dll - source code module: Pfg.cs



### 3.1.2.3 Enum and classes of parameter values

In the C language GTSL device drivers and libraries, the parameter values that can be transferred to functions are usually defined in the associated header file using the compiler statement `#define`.

In a .NET programming environment, groups of parameter values are typically combined either in an enum (in the case of integers) or in a class. During the method definition, the name of the enum or the class of parameter values is defined as the parameter type. This is a benefit to the user when using the method in an application because the code editor in Microsoft Visual Studio uses the IntelliSense function to display a list of all possible parameter values for the current position. The user selects the desired value from the list using the arrow keys and then presses the Enter key to accept. This reduces the likelihood of invalid parameter values or mistyped parameter names. The IntelliSense function offered by the Microsoft Visual Studio code editor is described in more detail in the following chapter.

### 3.1.2.4 Converting parameter types: string and integer <> enum

The previous section explained why enum parameters are preferred in the GTSL-.NET wrappers, while the C programming world often uses predefined integer parameter values or strings (character arrays) as the input and output parameters.

This means that in the wrapper classes, input parameters of the enum type must be converted into strings or integer values, which are then transferred to the C device drivers. This process is performed in reverse for output parameters.

For string input parameters, the conversion can be performed easily using the `GetName()` method of the enum class. The only consideration is to ensure that the names of the enum elements match the required strings.

In the case of integer input parameters, the enum element must be forced into an integer data type by prefixing the `(int)` type conversion operator to perform an explicit type conversion.

An additional consideration for output parameters is that parameter values returned by a C function might not have a matching value in the enum. In this case, integer parameters are checked using the enum method `IsDefined()` to determine whether the returned value is defined in the enum. If so, the parameter value can be converted into the enum type by means of the type conversion.

In the case of string parameters returned by a C function, the enum method `TryParse()` can check whether the string matches an element name for the enum. If so, the method returns the identified enum value.

## 3.2 GTSL Device Driver Example: The Pfg Class

In this example, a C application calls the R&S TS-PFG signal generator function `rspfg_ConfigureStandardWaveform` for the GTSL device driver DLL `rspfg.dll` as follows:

```
/* Error handling is not considered in this example in order
   to keep it easy to read. The return status should be checked
   after each driver call */

/* rspfg ivi-driver header file */
#include "rspfg.h"

main()
{
    ViSession viPfg;
    ViStatus status = VI_SUCCESS;

    /* Initialize the pfg device driver */
    status = rspfg_init ("PXI7::12::0::INSTR", VI_TRUE, VI_TRUE, &viPfg);

    /*
     Configure sine wave output on channel 1 with
     20.0 volts amplitude peak to peak, frequency: 100 kHz,
     phase: 0 Hz, dc offset voltage: 0.0 volt.
    */
    status = rspfg_ConfigureStandardWaveform (viPfg, "CH1", RSPFG_VAL_WFM_SINE,
                                             20.0, 0.0, 1.0e5, 0.0);

    /* Close the pfg device driver */
    status = rspfg_close(viPfg);
}
```

PfgCExample.exe - source code module: PfgCExample.c

The same functionality in a C# application is shown on the next page:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RohdeSchwarz.Gtsl;

namespace PfgDeviceDriverCall
{
    class Program
    {
        static void Main(string[] args)
        {
            Pfg signalGenerator = new Pfg();

            try
            {
                /* Initialize pfg signal generator instance */
                signalGenerator.Init("PXI7::12::INSTR");

                /*
                 Configure sine wave output on channel 1 with
                 20.0 volts amplitude peak to peak, frequency: 100 kHz,
                 phase: 0 Hz, dc offset voltage: 0.0 volt.
                */
                signalGenerator.ConfigureStandardWaveform(Pfg.Channel.CH1,
                                                         Pfg.Wfm.Sine,
                                                         20.0, 0.0, 1.0e5, 0.0);

                /* Close the pfg signal generator instance */
                signalGenerator.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}

```

PfgCSharpExample.exe - source code module: PfgCSharpExample.cs

First, the `new` operator generates an instance of the `Pfg` class. In this example, `signalGenerator` was used as the name of the instance. All public methods of that class can then be called, e.g.

```
signalGenerator.ConfigureStandardWaveform(...).
```

The namespace of the `GTSL.dll` being used, which includes the `Pfg` class, is named `RohdeSchwarz.Gtsl`. In the above example, this is included in the `using` list at the start of the code. To permit the `GTSL.dll` to be used in the user's application, it must be inserted into the project, for example by using the "Add reference..." command in Microsoft Visual Studio.

The parameters are transferred to the methods more conveniently than with the C functions. The signal generator channel being configured is easily selected using the IntelliSense function in the Visual Studio development environment, for example. In the corresponding C function, a string containing the channel name would have to be transferred at this point.

```

/*
    Configure sine wave output on channel 1 with
    20.0 volts amplitude peak to peak, frequency: 100 kHz,
    phase: 0 Hz, dc offset voltage: 0.0 volt.
*/
signalGenerator.ConfigureStandardWaveform(Pfg.Channel.
/* Close the pfg signal generator instance */
signalGenerator.Close();
    
```

The class diagram created by Visual Studio for the Pfg class displays all methods, enum types (e.g. the two signal generator channels) and additional subclasses representing parameter constants.

All methods of the Pfg class:

The screenshot shows the Visual Studio IDE with the Pfg class selected. The left pane displays the class hierarchy and methods, while the right pane shows a list of properties and methods.

**Pfg Class**  
 → TsvpInstrument

**Methods**

- ~Pfg
- AbortGeneration
- CanConnect
- ClearArbMemory
- ClearArbSequence
- ClearArbWaveform
- ClearErrorInfo
- CloseInstr
- ConfigureArbFrequency
- ConfigureArbMarker
- ConfigureArbSequence
- ConfigureArbWaveform
- ConfigureBurstCount
- ConfigureCoupling
- ConfigureDCOffsetRange
- ConfigureDutyCycle
- ConfigureFilter
- ConfigureGround
- ConfigureMarkerPolarity
- ConfigureOperationMode
- ConfigureOutputEnabled
- ConfigureOutputImpedance
- ConfigureOutputMode
- ConfigureOutputModeChannel
- ConfigureSampleRate (+ 1 overload)
- ConfigureSampleRateChannel (+ 1 overload)
- ConfigureStandardWaveform
- ConfigureTriggerDelay
- ConfigureTriggerSource
- Connect (+ 1 overload)
- ConvChan
- CreateArbSequence
- CreateArbWaveform
- Disconnect
- DisconnectAll (+ 1 overload)

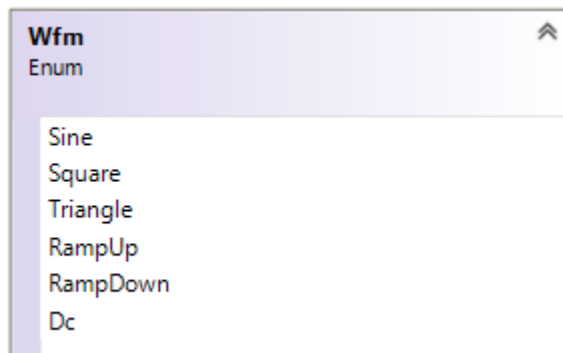
**Properties and Methods**

- ErrorMessage
- ErrorQuery
- FillAttributeDictionary
- GetAttribute (+ 1 overload)
- GetAttributeBoolean
- GetAttributeDouble
- GetAttributeInt32
- GetAttributeSessionHandle
- GetAttributeString
- GetErrorInfo
- GetNextCoercionRecord
- GetPath
- GetSeqHandleFromName
- GetSeqIndexInfo
- GetSeqLength
- GetSeqNameFromHandle
- GetSeqNames
- GetWfmHandleFromName
- GetWfmNameFromHandle
- GetWfmNames
- GetWfmSize
- InitiateGeneration
- InitInstr
- IsDebounced
- Pfg (+ 1 overload)
- QueryArbSeqCapabilities
- QueryArbWfmCapabilities
- Reset
- RevisionQuery
- SelfTest
- SendChannelSoftwareTrigger
- SendSoftwareTrigger
- SetAttribute (+ 1 overload)
- SetAttributeBoolean
- SetAttributeDouble
- SetAttributeInt32
- SetAttributeSessionHandle
- SetAttributeString
- SetPath
- WaitForDebounce

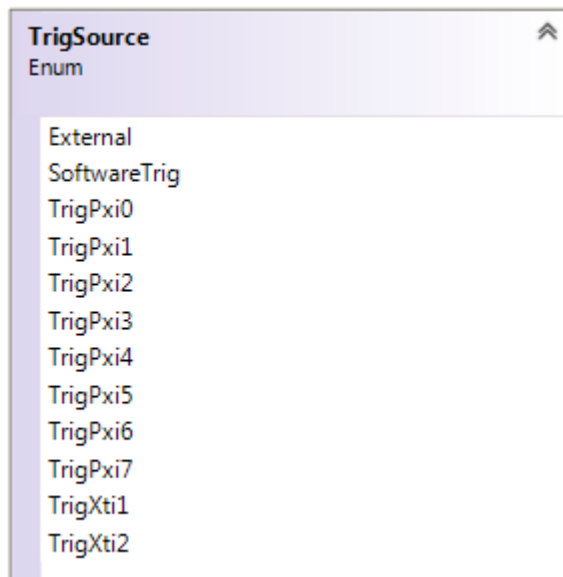
Examples of enum types defined in the Pfg class:



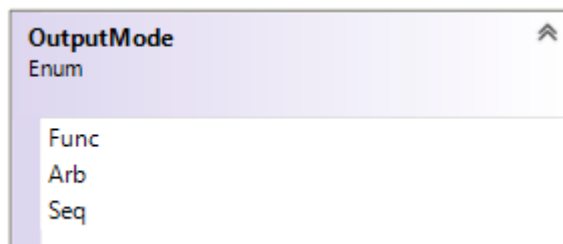
Selects one of the two channels of the PFG signal generator.



Possible default waveforms.



Signal generator trigger inputs for starting a waveform output.



Three output modes for the generator:  
 Default waveforms  
 Arbitrary waveforms  
 Arbitrary waveform sequences

### 3.3 GTSL Library Example: The ResMgr Class

The resource manager is a central component for GTSL libraries. It includes functions for administering the instrument setup in a test system. Like all other GTSL software libraries, the resource manager is written using the C programming language.

In a C application, the `RESMGR_Setup` function is called as follows:

```
#include "resmgr.h"

/* test system configuration files */
#define PHYSICAL_INI    "physical.ini"
#define APPLICATION_INI "application.ini"

int main (int argc, char *argv[])
{
    /* variables for error handling */
    short errorOccurred = 0;
    long  errorCode = 0;
    char  errorMessage[GTSL_ERROR_BUFFER_SIZE] = "";

    /* The resource manager loads the two configuration files
       required for the setup of all subsequent GTSL libraries */
    RESMGR_Setup (0, PHYSICAL_INI, APPLICATION_INI,
                  &errorOccurred, &errorCode, errorMessage);

    /* Setup of other GTSL Libraries */
    ...

    /* Performing tests */
    ...

    /* Cleanup of all used GTSL Libraries - ends with Resource Manager */
    ...
    RESMGR_Cleanup(0, &errorOccurred, &errorCode, errorMessage);
}

```

GtsLibraryExampleC.exe - source code module: GtsLibraryExampleC.c

In a C# application, the same function is called as follows:

```
namespace LibraryCallExe
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Runtime.InteropServices;
    using System.Text;
    using System.Threading.Tasks;
    using RohdeSchwarz.Gtsl;

    class Program
    {
        static void Main(string[] args)
        {
            string physicalIniName = "physical.ini";
            string applicationIniName = "application.ini";

            try
            {
                ResMgr resMgr = new ResMgr();

                resMgr.Setup(physicalIniName, applicationIniName);

                /* Setup of other GTSL libraries */
                ...

                /* Performing tests */
                ...

                /* Cleanup of all used GTSL libraries,
                 ends with Resource Manager */
                ...
                resMgr.Cleanup();
            }
            catch (Exception ex)
            {
                ...
            }
        }
    }
}
```

GtslLibraryExampleCSharp.exe - source code module: GtslLibraryExampleCSharp.cs

In this example, it can be seen that the parameter list of the setup method is significantly shorter than the equivalent setup function in C. This is because the return parameters for assessing any errors could be eliminated. The `Exception` mechanism from C# is used for error handling.

Like all other wrapper classes for GTSL libraries, the `ResMgr` class is derived from the base class `GtslCommon`. This includes classes for error handling and methods used by all library wrapper classes.



The class diagram generated by Visual Studio for the `ResMgr` class shows all methods, defined constants and error codes in the class:

The screenshot displays the Visual Studio class explorer for the `ResMgr` class. The class is located under `GtslCommon`. The `Methods` section lists 44 methods, including `~ResMgr`, `Alloc_Memory`, `Alloc_Resource`, `Alloc_Shared_Memory`, `Cleanup`, `Close_Session`, `Close_SubSession`, `Compare_Value`, `Enable_Tracing`, `Expand_Path`, `Free_Memory`, `Free_Resource`, `Free_Shared_Memory`, `Get_Key_Value`, `Get_Mem_Ptr`, `Get_Resource_Name`, `Get_Resource_Type`, `Get_Session_Handle`, `Get_Session_SubHandle`, `Get_Trace_Flag`, `Get_Value`, `Lib_Version`, `Lock_Device`, `Lock_Shared_Memory`, `Nth_Key_Name`, `Nth_Key_Value`, `Nth_Section_Name`, `Number_Of_Keys`, `Number_Of_Sections`, `Open_Session`, `Open_SubSession`, `Read_ROM`, `ResMgr (+ 1 overload)`, `Set_Session_Handle`, `Set_Session_SubHandle`, `Set_Trace_Flag`, `Setup`, `Trace`, `Unlock_Device`, and `Unlock_Shared_Memory`. The `Nested Types` section is currently collapsed. To the right, two static classes are shown: `Constants` and `Error`. The `Constants` class contains four fields: `MAX_NAME_LENGTH`, `MAX_RESOURCE_IDS`, `MAX_VALUE_LENGTH`, and `ROM_LENGTH`. The `Error` class contains 36 fields, including `BENCH_NOT_FOUND`, `DEVICE_NOT_FOUND`, `DIFFERENT_APPL_LAYER`, `DIFFERENT_PHYS_LAYER`, `INDEX_OUT_OF_RANGE`, `INTERNAL`, `INVALID_LAYER`, `INVALID_RESOURCE_ID`, `LOCK_DEVICE`, `LOCK_MANAGER`, `LOGNAME_NOT_FOUND`, `MEMORY_ALLOCATION`, `MEMORY_ALREADY_EXISTS`, `MEMORY_LOCK`, `MEMORY_NAME`, `MEMORY_NOT_EXISTS`, `MEMORY_NOT_FOUND`, `MEMORY_SIZE`, `MEMORY_TABLE_FULL`, `MEMORY_UNLOCK`, `NAME_TOO_LONG`, `PROCESS_TABLE_FULL`, `READING_APPL_LAYER`, `READING_PHYS_LAYER`, `READING_ROM`, `RESOURCE_NOT_ALLOCATED`, `RESOURCE_TABLE_FULL`, `SESSION_ALREADY_EXISTS`, `SESSION_NOT_EXISTS`, `SESSION_NOT_OPEN`, `SETUP_MISSING`, `UNLOCK_DEVICE`, and `UNLOCK_MANAGER`.



## 4 GTSL Programming Examples

### 4.1 Direct Driver Calls

This example shows how to combine GTSL library calls and GTSL device driver calls within an application.

The setup method of the GTSL-.NET wrapper class for a GTSL library opens all of the device drivers required by the library. If a device driver is used by more than one GTSL library, the session handle for the driver is automatically shared by all participating libraries. The GTSL resource manager manages the exchange of the session handle for device drivers. It administers the session handle for each instrument contained in the physical.ini configuration file for the test system.

If a device driver must be called directly from the application, for example because the GTSL library does not support a specific method call, the corresponding session handle must be requested of the resource manager.

In a mixed application of GTSL libraries and GTSL device drivers, a device driver must not be opened twice, i.e. via the library and in parallel directly via the device driver. Although it is theoretically possible, it conflicts with the 'state caching' mechanism of the internal device state used by the IVI drivers. This data is no longer valid if the device is accessed via two different driver instances.

Note that in the following C# source code, the session handle for the PFG signal generator requested of the resource manager is transferred to the overloaded constructor of the `Pfg` class when the class is instantiated. This serves to inform the instance that it need not (and in fact must not) open the device driver itself. A subsequent call of `Pfg.Init(...)` would generate an exception with the following error text:

"This instance, created by using a resource manager handle, does not allow calling this method!" : "The device is already initialized!".

This same applies to the `Pfg.Close(...)` method, which also must not be called in this situation. The `Cleanup` method of the GTSL-.NET wrapper class of the last GTSL library which uses the PFG signal generator is responsible for closing the device driver. The session handle becomes invalid after the `Cleanup` method of the library ends. No more method calls of the instance of class `Pfg` are possible then.

```

namespace SampleDirectDriverCall
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;
    using RohdeSchwarz.Gtsl;

    class ProgramDirectDriverCall
    {
        static ResMgr resMgr = new ResMgr();
        static Route route = new Route();
        static Pfg myDirectDevice;

        static void Main(string[] args)
        {
            string physicalIniName = "physical.ini";
            string applicationIniName = "application.ini";
            string benchName = "bench->PFG";
            string benchDevice = "SwitchDevice1";

            Console.WriteLine("GTSL: Direct Driver Call Sample:\r\n");

            try
            {
                Console.WriteLine("ResMgr.Setup()");
                resMgr.Setup(physicalIniName, applicationIniName);

                Console.WriteLine("Route.Setup()");
                route.Setup(benchName);

                IntPtr sessionHandle;
                Console.WriteLine("ResMgr.Get_Session_Handle()");
                resMgr.Get_Session_Handle(route.ResourceId, benchDevice, out sessionHandle);

                Console.WriteLine("Create new Pfg class instance");
                myDirectDevice = new Pfg(sessionHandle);

                Pfg.OutputMode modeIn = Pfg.OutputMode.Arb;
                Console.WriteLine("Pfg.ConfigureOutputMode() to '" +
                    modeIn.ToString() + "'");
                myDirectDevice.ConfigureOutputMode(modeIn);

                Console.WriteLine("Pfg.GetAttribute(Pfg.Attribute.OutputMode)");
                Pfg.OutputMode modeOut =
                    (Pfg.OutputMode)myDirectDevice.GetAttribute(Pfg.Attribute.OutputMode);

                // comparing the configured output mode with the attribute value returned
                // by the call of method GetAttribute
                Console.WriteLine("Result: " +
                    ((modeOut == modeIn) ? "OK!" : "Mode does not match!") +
                    " ('" + modeOut.ToString() + "' <-> '" +
                    modeIn.ToString() + "'");
            }
        }
    }
}

```

- continued on next page -

```
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
finally
{
    Console.WriteLine("Route.Cleanup()");
    route.Cleanup();

    Console.WriteLine("ResMgr.Cleanup()");
    resMgr.Cleanup();

    Console.WriteLine("Done!");
    Console.Write("Press any key to close window ... ");
    Console.ReadKey();
}
}
}
```

Source: ProgramDirectDriverCall.cs

## 4.2 Combi Test

The Combi Test program shows how to create a combined in-circuit test and functional test application in C# using the following GTSL libraries:

- Resource Manager
- Signal Routing Library
- In-Circuit Test Library

```
namespace SampleCombiTest
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;
    using RohdeSchwarz.Gtsl;

    class ProgramCombiTest
    {
        static ResMgr resMgr = new ResMgr();
        static Route route = new Route();
        static Ict ict = new Ict();
        static Pfg myDirectDevice;

        static void Main(string[] args)
        {
            string physicalIniName = "physical.ini";
            string applicationIniName = "application.ini";
            string benchName = "bench->CombiTest";
            string benchDevice = "SwitchDevice1";
            string ictProgName = "CombiTest.ict";
            int ictProgId = -1;

            Console.WriteLine("GTSL: CombiTest Sample:\r\n");

            try
            {
                SetupTests(physicalIniName, applicationIniName, benchDevice,
                    benchName, ictProgName, out ictProgId);

                RunTests(ictProgId);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                CleanUpTests();

                Console.WriteLine("Done!");
                Console.Write("Press any key to close window ... ");
                Console.ReadKey();
            }
        }
    }
}
```

- continued on next page -

```

static void SetupTests(string physicalIniName, string applicationIniName,
                      string benchDevice, string benchName,
                      string ictProgName, out int ictProgId)
{
    Console.WriteLine("ResMgr.Setup()");
    resMgr.Setup(physicalIniName, applicationIniName);

    Console.WriteLine("Route.Setup()");
    route.Setup(benchName);

    Console.WriteLine("Ict.Setup()");
    ict.Setup(benchName);

    bool compileErrors;
    Console.WriteLine("Ict.Load_Program()");
    ict.Load_Program(ictProgName, benchName, out ictProgId, out compileErrors);
    if (compileErrors)
    {
        throw new Exception("ICT: Compile errors found!");
    }

    IntPtr sessionHandle;
    Console.WriteLine("ResMgr.Get_Session_Handle()");
    resMgr.Get_Session_Handle(route.ResourceId, benchDevice, out sessionHandle);

    Console.WriteLine("Create new Pfg class instance");
    myDirectDevice = new Pfg(sessionHandle);
}

static void RunTests(int ProgId)
{
    int failCount;

    Console.WriteLine("Configure and activate waveform at PFG");
    myDirectDevice.ConfigureOutputModeChannel(Pfg.Channel.CH1, Pfg.OutputMode.Func);
    myDirectDevice.ConfigureStandardWaveform(Pfg.Channel.CH1, Pfg.Wfm.RampUp,
                                             5.0, 0.0, 1000000, 0.0);
    myDirectDevice.ConfigureOutputEnabled(Pfg.Channel.CH1, true);

    Console.WriteLine("Run ICT Program");
    ict.Run_Program(ProgId, "", "", out failCount);

    Console.WriteLine("Switch off waveform at PFG");
    myDirectDevice.ConfigureOutputEnabled(Pfg.Channel.CH1, false);
}

static void CleanUpTests()
{
    Console.WriteLine("Ict.Cleanup()");
    ict.Cleanup();

    Console.WriteLine("Route.Cleanup()");
    route.Cleanup();

    Console.WriteLine("ResMgr.Cleanup()");
    resMgr.Cleanup();
}
}
}
}

```

Source: ProgramCombiTest.cs

## 5 Installation Procedure

The installation program for the source code of the GTSL-.NET wrapper classes can be downloaded from the Rohde & Schwarz homepage:

<http://www.rohde-schwarz.com/appnote/1SE001>

The name of the installation file is `InstallerGtsIWrapper.msi`. Installation is started by right-clicking the file and selecting 'Install' from the context menu.

If GTSL is already installed on the target machine the GTSL-.NET wrapper classes will be installed to directory `<GTSL directory>\Develop\.net wrapper`.

If GTSL is not installed yet, the installation directory is `C:\Program Files (x86)\Rohde-Schwarz\GTSL\Develop\.net wrapper`.

To build the `GTSL.dll` simply open the Visual Studio solution `GtsIWrapper.sln` which is located in the installation directory and choose in the Configuration Manager whether to build a debug or release version of the DLL. After the build process the `GTSL.dll` can be found in the directory `...\net wrapper\GTSL\bin\Release (or Debug)`.

To use the GTSL-.NET wrapper classes in any project either the `GTSL.dll` can be added to the project with the menu 'Add reference...' in Visual Studio or all relevant source code files can be added to the project. The necessary source code files are located in the subdirectories `Common`, `Instruments` and `Libraries` below directory `...\net wrapper\GTSL\`.

## 6 Development Tools

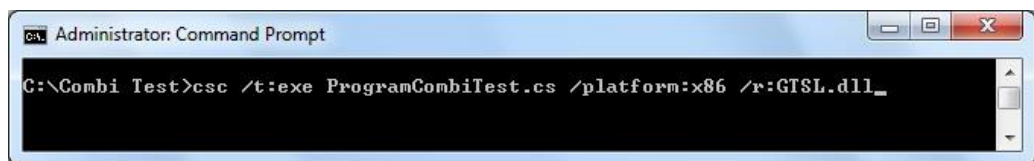
Users who do not install paid copies of Microsoft Visual Studio Professional or Enterprise Edition on their development PC still have options to create, quickly and at no additional cost, their own C# test applications that include the GTSL-.NET wrapper classes.

### 6.1 Compiler `csc.exe` from the .NET Framework SDK

The .NET framework SDK includes a C# compiler that can be launched from the command line. It is called `csc.exe` and is located in the following path: `C:\Windows\Microsoft.NET\Framework\v3.5\csc.exe`. The code to be compiled must be created in a separate text editor.

This path should be added to the Windows system path variable `Path` so that the compiler can be called from any directory on the PC.

The Combi Test programming example from section 4.2 is shown here as an example of a call of the `csc.exe` compiler:

A screenshot of a Windows Command Prompt window titled "Administrator: Command Prompt". The command prompt shows the following command being executed: `C:\Combi Test>csc /t:exe ProgramCombiTest.cs /platform:x86 /r:GTSL.dll_`. The command prompt is running in a directory named "C:\Combi Test".

This call generates the console application `ProgramCombiTest.exe`. It is a 32-bit application that references the `GTSL.dll` class library.

### 6.2 Visual Studio Community

Microsoft offers a free version of its Visual Studio development environment, called Visual Studio Community. This version can be used under the conditions published by Microsoft. The functionality corresponds to Visual Studio Professional Edition.

Visual Studio Community is intended to replace Visual Studio Express, which is also free of charge. As of March 2016, the current version of Visual Studio Express 2015 is available yet.



## 7 Resources

The installation program for the source code of the GTSL-.NET wrapper classes can be downloaded from the Rohde & Schwarz homepage:

<http://www.rohde-schwarz.com/appnote/1SE001>

A compressed file containing the GTSL installation DVD can be downloaded after registering with Rohde & Schwarz GLORIS: See

<https://gloris.rohde-schwarz.com/>

For more information about the R&S®CompactTSVP and R&S®PowerTSVP production test platforms and about the associated R&S®GTSL software package, go to:

[www.tsvp.rohde-schwarz.com](http://www.tsvp.rohde-schwarz.com)



## Rohde & Schwarz

The Rohde & Schwarz electronics group offers innovative solutions in the following business fields: test and measurement, broadcast and media, secure communications, cybersecurity, radiomonitoring and radiolocation. Founded more than 80 years ago, this independent company has an extensive sales and service network and is present in more than 70 countries.

The electronics group is among the world market leaders in its established business fields. The company is headquartered in Munich, Germany. It also has regional headquarters in Singapore and Columbia, Maryland, USA, to manage its operations in these regions.

## Regional contact

Europe, Africa, Middle East  
+49 89 4129 12345  
[customersupport@rohde-schwarz.com](mailto:customersupport@rohde-schwarz.com)

North America  
1 888 TEST RSA (1 888 837 87 72)  
[customer.support@rsa.rohde-schwarz.com](mailto:customer.support@rsa.rohde-schwarz.com)

Latin America  
+1 410 910 79 88  
[customersupport.la@rohde-schwarz.com](mailto:customersupport.la@rohde-schwarz.com)

Asia Pacific  
+65 65 13 04 88  
[customersupport.asia@rohde-schwarz.com](mailto:customersupport.asia@rohde-schwarz.com)

China  
+86 800 810 82 28 | +86 400 650 58 96  
[customersupport.china@rohde-schwarz.com](mailto:customersupport.china@rohde-schwarz.com)

## Sustainable product design

- Environmental compatibility and eco-footprint
- Energy efficiency and low emissions
- Longevity and optimized total cost of ownership



This **Choose an item** and the supplied programs may only be used subject to the conditions of use set forth in the download area of the Rohde & Schwarz website.

R&S® is a registered trademark of Rohde & Schwarz GmbH & Co. KG; Trade names are trademarks of the owners.