TeleApp #004

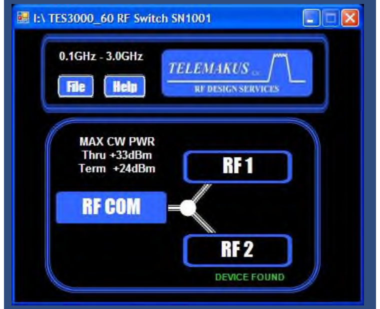
Programming Interface for Telemakus USB Devices

API Development

API Structure

API Usage
Basic Concepts

Code Examples



Telemakus USB products were originally designed as manually controlled, low cost, highly portable, measurement tools to assist RF design engineers, technicians and students with the task of performing complex RF measurements. Plug & Play USB operation and intuitive Graphical User Interfaces (GUIs) made them simple to install and easy to use without the need for developing external control software. However, soon after the initial product introduction, many users began asking for an application programming interface (API) so that these devices could be embedded into larger Automated Test Systems as well as system level products. A programming interface was added to the product a year later and first released in 2010. This tutorial guide provides a conceptual overview of the programming interface and discusses some simple examples of its use.

Background

The Telemakus USB API is best understood by first understanding the operation of the GUI. The GUI is much more than simply a user interface in this application. It establishes and manages the USB communications. It performs error checking and error handling. It coordinates the operation of graphical objects within the Windows Forms display system. And, it handles portions of the creation and destruction of the data structures required to perform all of these tasks. If the GUI didn't exist, a user would need to be able to program all of these operations within their own application just to reproduce the function of the simple GUI provided.

For the GUI to communicate with the device, it must first search through the Windows device registry and attempt to locate the device to ensure that it is actually connected to the system. Once located, the GUI sends a request to Windows to open a communication pipe to the device. If Windows can verify that all of the correct requirements are in place to be able to successfully establish the pipe, then the request is granted and the process of setting up the communications is started. This involves multiple layers of process initiation, beginning at the USB interface, by locating and launching the correct USB driver. The process then continues by setting up file streaming buffers and concludes with the construction of a socket interface in the Windows interprocess messaging system that links the GUI process to the communications process. Windows then returns a "Handle" to the GUI process that allows the GUI process access to the communications pipe for both messaging and control of the pipe. Again, a user would need to duplicate this entire process just to be able to communicate with the device.

API Development

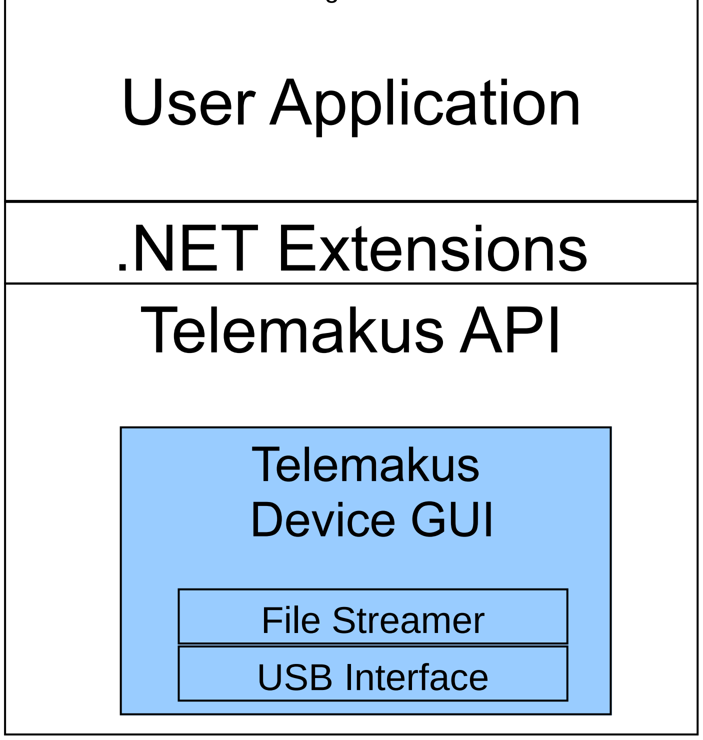
One of the most important considerations in developing the API was to insulate the developer from the necessity of learning the details of USB programming. This meant leveraging as much of the GUI code as possible for the API. A decision was made to create the API as a “wrapper” file around the existing GUI code rather than recreating the GUI code within the API. A decision was also made to create a high level programming interface for the user that would be compatible across many development tools. The platform that was selected was Microsoft’s .NET operating system extensions. This tool is supported by a wide range of software development languages and tools such as VB.NET, Visual C#, C++, LabView™, Agilent VEE™ and MATLAB™ just to name a few of the more popular ones. All of these tools support integration of .NET objects using class based structured programming techniques.

API Structure

A conceptual diagram of the API integrated with a user application is shown below in figure 1.

The advantage of this structure for the user is that it frees the user from the necessity of managing all of the layers in the code. The communication structures within the GUI are launched from the API with a simple function call to Load the driver library. The API then loads a copy of the GUI into background as a hidden object and manages the operation of the USB device through the GUI process. When the user application has finished, the user makes a function call to the API to Unload the driver library and the API closes the background GUI process and all of its sub-tasks.

Figure 1



API Usage – Basic Concepts

Software Components

There are two software components to the Telemakus API. The first component is the device driver library file. The second component is the device GUI file. These files can be found in the installation folder shown below on Windows 32 bit systems. For Windows 64 bit systems, the location varies according to your machine.

C:\Program Files\Telemakus LLC\<DeviceName>

You will be able to recognize the files by their names:

<DeviceName>_lib.dll	(library driver)
<DeviceName>.exe	(device GUI)

Where <DeviceName> represents the name for your specific Telemakus device.

These files will execute from any folder location and may be freely copied and moved to any convenient location for your application. *The only requirement is that the two files need to be kept together in the same folder.* The library driver will only look for the GUI from within the same folder where it resides and nowhere else. If it can't find the GUI file it will throw a system level error message that the target file is not found.

Most compilers will need assistance to locate these files. There are usually tools available within the compiler environment to show the compiler where to locate resource files so it can link to them at compile time. A convenient place to locate the files is in the same folder as your application code. Most compilers will automatically default to this folder to search for named resources.

Working with the Telemakus API Class Structures

A readme file is shipped along with the device that details the class structure for that particular device. It can be found in many different locations. The first location is on the device flash drive. The second is in the installation folder. A third is in the folder of the example API project file shipped with the unit. The example project is also located in the installation folder. The name of the readme file is:

<DeviceName> Driver readme.txt

The discussion that follows is a general discussion relevant to programming all Telemakus devices. The user should examine the readme file and the example project for implementation details specific to their own device.

The Device Driver Class

All Telemakus devices follow the nomenclature below for the class names of the specific device drivers:

```
<DeviceName>_lib.<DeviceName>_drv
```

This device driver class utilizes both the Windows - System.Runtime.InteropServices and the device GUI as imported resources.

Regardless of the programming language used, the first step to invoke this device class is to create an object defined as this class type and then create an instance of the object. Some examples of this step are shown below in different programming languages.

Visual Basic .NET

```
Private myAtten1 as TEA4000_7_lib.TEA4000_7_drv

myAtten1 = NEW TEA4000_lib.TEA4000_7_drv
```

C++

```
using namespace TEA4000_7_lib;

TEA4000_7_drv myAtten1;
```

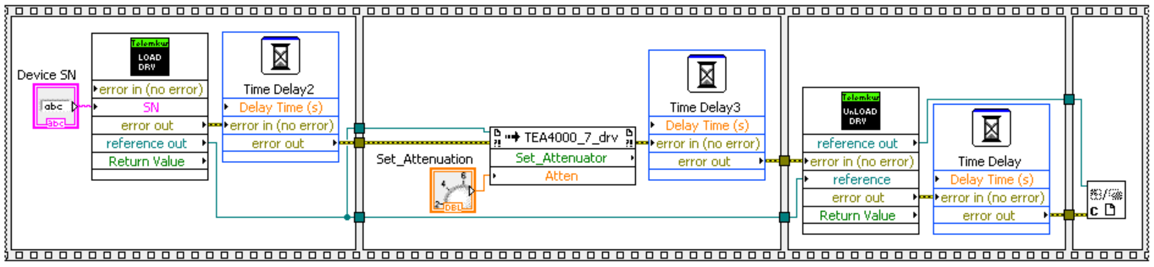
MATLAB

```
NET .addAssembly('c:\\programfiles\Telemakus LLC\TEA4000_7 Attenua-
tor\TEA4000_7_lib.dll');

my_Atten1 = TEA4000_7_lib.TEA4000_7_drv;
```

Graphical Programming languages such as LabView and VEE also support the creation of this driver class object using the tools they supply for working with .NET objects. It is best to refer to the user manuals if you are not familiar with this topic. Both Agilent and National Instruments have extensive help files on the subject. Telemakus also has several programming examples available in these languages as starter files for users. These are not shipped with the units but are available upon request through Telemakus Support.

The figure below illustrates a starter Virtual Instrument file (VI) for National Instruments Lab-View



Telemakus Support can also provide example files in other languages although these files may not be available for all devices or all languages. There is usually enough information included in each file to understand the general syntax for a given language so that a user can then apply that syntax to a specific device.

Launching the API

Creating the instance of the device driver object will automatically launch the Telemakus API.

The common practice is to define the object type early in the program, usually in the header portion, and delay the actual creation of the instance of the object until a later section of the application. This gives the developer some level of control over where and when the API is actually invoked.

Launching the API creates the data structures necessary to support the communications with the device and the interface between the user application and API. It does not open the actual pipe (communications channel) to the device. As discussed earlier, opening the pipe requires a function call to the primary “Load” method of the API. These primary methods will be discussed in a later section.

Closing the API

Disposing the device driver object removes the communications data structures from memory and closes the interface between the user application and the API.

Typically, this occurs automatically when the user application is closed. But, it is also possible to force the disposal of the device driver object under program control by invoking system level disposal methods written to support a given programming language.

The API attempts to manage the USB interface in a manner that prevents the interface from being hung under a number of different conditions including accidental disconnect of the physical device. But disposal of the device driver object, while the pipe is still open can lead to unpredictable results. It is usually bad practice to dispose of the device driver object while the pipe is still open to the device. This can result in hanging up the USB interface that the device is connected to. *Best practice is to call the primary "Unload" method of the API to close the device pipe before allowing the disposal of the device driver object. It is also a good practice to create an error handler that can "Unload" the device before allowing an application to close during an error condition.*

Invoking the Primary Device Driver Methods (Functions)

All Telemakus devices support three primary device methods (functions). The primary device methods are used to manage the USB communications with the device. They are defined below using Visual Basic syntax:

Public Function Load_drv(ByVal mySN As String) As Boolean

This function loads the background GUI driver and opens a USB communication path to the target Serial Number device. The function assigns the Serial Number argument to the driver object created so that the Serial Number is no longer required to reference the object. The Serial Number argument (mySN) is the 4 digit string from the device Serial Number located on the front label. The function returns True if the GUI driver is successfully loaded and the USB communication path is successfully opened. Otherwise, it returns False.

Public Function Unload_drv() As Boolean

This function releases the USB handles to the controlled device, and closes the inter-process messaging routines to the device. It then completes the closing process by signaling the GUI driver object to cleanup any pending transactions and prepare to be unloaded. The GUI driver object then signals Windows that it is ready for disposing and Windows completes the cleanup. This function returns True if the message is successfully sent to the GUI driver to unload. Otherwise it returns False. The actual process of unloading the GUI driver is not checked. *As a precaution, the error handler for any application using the GUI driver should unload the GUI driver before closing down that application.*

Public Function Check_Status() as Boolean

This function returns True if the USB device handles are valid and the device is responsive. It returns False if there is a problem. A GUI driver that has been successfully loaded and connected to a target device will continue to monitor that device even after that device has been disconnected from the system. If a device that has been disconnected is later re-connected to the system, the loaded driver will automatically re-establish USB handles to the device and return True when polled by an application using Check_Status (). The example program uses a Windows timer object to periodically poll the device for Check_Status().

The syntax of the definitions instruct the Visual Basic compiler to create a Windows socket interface that will allow these functions to be called from an external application and to expose that interface so that they become visible to class based reflector tools. The functions are defined to return a Boolean value as a result of a successful operation. These exposed functions may then be called from any number of different programming languages and environments using the syntax for the target language. Some examples of Load_drv() function calls are given below in different languages:

Examples of the Load_drv function

VB.NET

```
      If Atten1_lib_loaded = False Then
                SN1 = TextBox3.Text

                myAtten1 = New TEA4000_7_lib.TEA4000_7_drv
                success = myAtten1.Load_drv(SN1)
      End If
```

C++

```
      bool success = false;
      System::String^ strSN="1001"; //put SN for your device here

      TEA4000_7_lib.TEA4000_7_drv myAtten1; //create instance for
      attenuator API

      System::String^ strMsg="Type in the Serial Number of your
      TEA4000_7 device and press RETURN";

      Console::WriteLine(strMsg);
      strSN = Console::ReadLine();

      success = myAtten1.Load_drv(strSN); //create handles and open pipes
```

MATLAB

```
%This function sets up the .NET Class Object and opens a USB connection to
the device

      function Load_Atten(SN)

      NET .addAssembly('c:\\Program Files\Telemakus LLC\TEA4000_7 Attenua-
      tor\TEA4000_7_lib.dll');

      my_Atten = TEA4000_7_lib.TEA4000_7_drv;

      result = my_Atten.Load_drv(SN);
      pause(3); %Wait 3 seconds for Windows to complete the Load operation
      in case it's busy

      end
```

When the Load command executes the user may notice the GUI startup briefly and then be pushed into background where it will remain hidden until the Unload command is executed. This is normal and serves as additional confirmation that the API is active and running.

The Windows Task Manager will show that the user application is running but the Task Manager will not show the GUI driver running as a background process.

The best way to verify that the driver is in fact executing is to call the Check_Status() function. This function essentially pings the device and verifies that the communication pipe is functioning. It returns TRUE if the device is responsive. Typically, a Windows Timer object can be used to periodically poll the status of the device. An example of its use is provided below:

VB.NET

```
      Private Sub Timer1_Tick(ByVal sender As Object, ByVal e As Sys-
      tem.EventArgs) Handles Timer1.Tick
              'Use timer to monitor status of device

              If Atten1_lib_loaded Then
                  success = myAtten1.Check_Status
                  If success = True Then
                      Label3.BackColor = Color.Lime
                  Else
                      Label3.BackColor = Color.Red
                  End If
              End If

       End Sub
```

The Check_Status() function is a very useful tool for preventing an application from hanging up should a device be removed from the system while the application is running. Each layer of the communications pipe is validated before it is executed to ensure the system will not hang at the USB interface if a problem occurs. This gives the user the opportunity to trap on errors and take appropriate actions to ensure a clean re-start/recovery from the error condition.

The Unload_drv() function is critical to successfully terminating the execution of a user application. As the name implies, the purpose of this function is to cleanup the USB interface to the device and unload the background GUI driver in a manner that leaves both the Telemakus device and the USB bus ready for a trouble free re-start. This function is normally called any time a user application is preparing to close. But it should also be called any time a user application halts on an error condition. If a user application crashes for any reason, the error handler should attempt to call Unload_drv() before allowing program execution to halt. This can be handled quite easily in programming languages that allow constructs for error management. In others, it could prove to be a challenge during program development and testing.

Most programming development tools will be successful at recovering from an unexpected stoppage of the application even if the Unload_drv() is not used. Most will be able to recover everything except for resetting the USB interface and resetting the Telemakus device. In this situation, the device may report that it is still “linked to another process” even though the application was halted. *If this situation should occur, both the USB interface and the Telemakus device can be reset by physically disconnecting the Telemakus device from the system for a few seconds and* *then re-connecting the device and waiting for about 5 seconds while Windows resets the interface and re-enumerates the device.*

Conclusion

The Telemakus API was created to assist users in developing creative and cost effective RF test applications using our devices across a wide variety of programming languages and development tools. The .NET class based common programming interface simplifies the task of integrating multiple devices into larger complex systems. A user can quickly design a very robust application by following just a few simple guidelines for managing how and when the USB pipes are opened and closed for each device. Adding a level of USB management to the error handler of the application can further strengthen its robustness.

The possibilities of USB controlled RF test equipment are mainly limited by the creativity of the developer. Not only can these devices be controlled by a PC, but we have also demonstrated that our devices can be plugged into and operated from the front panel display of other Windows based test instruments. The limitations are few. The possibilities are many. Such is the promise of USB controlled instrumentation.

Telemakus Support

The Telemakus API is designed to be straight forward and easy to manage. But occasionally questions come up during the course of developing a program or project.
Telemakus Support is available to assist you with your project development and answer questions regarding the operation of the device and its API.

Our first recommendation is to always consult the example project provided. There is much information contained in the code listings that can lead to a solution to most problems. But if an answer is not so obvious, please contact us at the email address below. We are here to help.

support@telemakus.com