inheco

Software Integration Guide

Teleshake (95) AC USB

Part No.: 7100180 | 7100181 | 7100182 | 7100183

Doc ID: 901779-000

08/2024

Company information

INHECO Industrial Heating and Cooling GmbH

Fraunhoferstr. 11

82152 Martinsried

Germany

Telephone sales:	+49 89 899593 120
Telephone Techhotline:	+49 89 899593 121
Fax:	+49 89 899593 149
E-mail - Sales:	sales@inheco.com
E-mail - Technical – Hotline:	techhotline@inheco.com
Website:	**www.inheco.com**
Technical Support & Trouble Shooting Instructions:	www.inheco.com/tech-support.html

INHECO Industrial Heating and Cooling GmbH reserves the right to modify their products for quality improvement. Please note that such modifications may not be documented in this manual.
This manual and the information herein have been assembled with due diligence.
INHECO Industrial Heating and Cooling GmbH does not assume liability for any misprints or cases of damage resulting from misprints in this manual. If there are any uncertainties, please feel free to contact sales@inheco.com.
The brand and product names within this manual are registered trademarks and belong to the respective titleholders.

Table of Contents

Company information 2
1 Introduction 5
1.1 Overview 5
1.2 Product Description 5
2 System Requirements 6
2.1 Hardware Requirements 6
2.2 Software Requirements 6
3 Installation Guide 7
4 Getting Started 9
4.1 Thread Safety Restrictions (Important Note) 9
4.2 Initial Setup 9
4.3 Preparing the Codebase for Device Communication 9
4.4 Finding Connected Devices 10
5 API Integration Strategies 11
5.1 Recommended Integration Approach 11
5.2 Teleshake (95) AC USB Library Package 13
5.3 Teleshake AC USB Adapter (Adapter DLL) 14
6 Advanced Integration Examples 17
6.1 Setting Up and Searching for Devices 17
6.2 Retrieving Device Information 17
6.3 Executing Commands and Handling Responses 17
6.4 Managing Device Operations 18
6.5 Performing a Shaking Test 18
6.6 Reading and Setting Temperature 18
6.7 Comprehensive Integration Example Code 19
6.8 ExecutionService Class Documentation 23
7 Additional Tools for Device Management and Development 33
7.1 Device Manager 33
7.2 Command Line Tool 34
8 Troubleshooting 35
8.1 Device Connectivity Issues 35
8.2 Firmware Command Execution Failures 35
8.3 Unexpected Device Behavior 35
8.4 Additional Support 35
9 Support and Resources 36

10 Legal and Licensing 37
11 Glossary 38

1 Introduction

1.1 Overview

This document describes the integration of the Teleshake (95) AC USB supporting libraries into third-party custom applications. It is intended to provide developers with the necessary information and guidelines to seamlessly incorporate the full functionality of the device within a broader automation ecosystem.

By following the instructions outlined in this document, developers can ensure compatibility and performance of the device within their custom solutions.

1.2 Product Description

The Teleshake series, encompassing both the Teleshake AC USB (for shaking) and Teleshake 95 AC USB (for shaking and heating), represents a state-of-the-art solution for enhancing liquid handling processes in life science, including molecular biology, biochemistry, and clinical chemistry and beyond. These devices are designed to improve lab efficiency and precision through precise heating and shaking capabilities.

Operational control is achieved through independent software, making these units versatile for integration into robotic platforms and LabAutomation systems.

2 System Requirements

2.1 Hardware Requirements

To ensure seamless integration and optimal performance of the devices, the following hardware requirements must be met:

USB Data Connection

A standard USB port is required for direct connection between the devices and the host computer. This connection facilitates data transfer and device operation control.

Power Supply

For operating the features of active clamping, shaking, and heating (optional), the device requires a connection to a 24 Vdc power supply unit, that can deliver up to 8 A for the full featured device configuration.

Operating System

The device is compatible with Windows operating systems. While it has been extensively tested on Windows 10, it is expected to function correctly on other versions of Windows as well. Users are encouraged to verify compatibility with their specific Windows OS version.

Computer System

A PC equipped with at least an Intel i5 processor (or its equivalent), 8GB of RAM, and sufficient hard drive space for software installation and data storage is required. The system must also support the necessary software environment for the device's operation.

2.2 Software Requirements

To integrate and operate the device within your automation setup, the following software requirements must be met:

Operating System Compatibility

The device software libraries have been developed and tested specifically for use with Windows 10 and later versions of the Microsoft Windows operating system. While the libraries may function on other versions of Windows, optimal performance and compatibility are guaranteed only for Windows 10 and newer.

.NET Framework

The supporting software libraries for the devices are currently built on the Microsoft .NET 8.0 framework. It is essential that a host system also provides .NET 8.0 to ensure full functionality.

Driver Software

The latest USB driver compatible with the device must be installed on the system to facilitate communication between the device and the PC. All “Teleshake (95) AC USB” devices comply with the “Device Class Definition for HID” (Human Interface Device) by the USB-IF (USB Implementers Forum). Therefore, HID devices do not rely on the additional installation of a communications driver within the operating system.

Cross-Platform Support

The software libraries provided are primarily developed for Windows environments and the x64 processor architecture. While the .NET 8.0 framework has cross-platform capabilities, the specific integration and performance on Linux or other non-Windows operating systems have not been validated. If integration outside of Windows or other processor architectures is planned, please contact Inheco for further assistance.

3 Installation Guide

To incorporate the provided software libraries into your project, two principal methods are provided to ensure a targeted integration process:

1. DLL_Package_Installer Execution

Utilize the "DLL_Package_Installer" for an automated setup. Select the recommended application directory or opt for a custom directory of your choice. This method efficiently organizes the necessary components in the selected location. Once installed, reference these DLLs within your project directly from the installation directory.

2. Direct File Placement

If you prefer a more hands-on approach or need to integrate the DLLs into a specific project structure, you can directly place the provided DLL files into a custom folder on your local folder. Reference the DLLs from this folder in your application. This method offers the flexibility to align with various project configurations.

Specific DLLs

Our integration package includes essential DLLs tailored for device functionality of the Teleshake (95) AC USB product family:

- Inheco.HidApi
- Inheco.HidDevice.Shared
- Inheco.TeleshakeAcUsb
- Inheco.TeleshakeAcUsb.Adapter

These DLLs, delivered as part of the provided package, are crucial for the full utilization of the device features.

Utilizing NuGet for Third-Party Libraries

To streamline the integration process and ensure you are working with the latest versions of necessary third-party libraries, it is recommended to install the following .NET DLLs through the “NuGet Package Manager”:

- Castle.Core
- DryIoc.dll
- log4net
- Moq
- Newtonsoft.Json

Installation via “NuGet” not only simplifies the update process but also maintains the integrity of your project dependencies. This method is demonstrated within the sample integration application for clarity and ease of use.

Processor Architecture Consideration

Include a processor architecture-specific folder, such as "win-X64," to house the "hidapi.dll" for your own setup. We’ll provide a “hidapi.dll” for each processor architecture, but always with the same name to provide cross-platform independency.

Please contact Inheco if you have any questions regarding a “hidapi.dll” for an alternative processor architecture other than the x64 platform.

Version Information through Sample Application

To assist developers in ensuring they are utilizing the correct versions of the libraries installed via Microsoft's "NuGet Package Manager", the sample integration application includes detailed information on the versions of the "Castle.Core", "DryIoc.dll", "log4net", "Moq", and "Newtonsoft.Json" DLLs. This feature aids in maintaining compatibility and leveraging the latest functionalities offered by these libraries. By referencing the sample application, developers can seamlessly verify and match the library versions required for optimal integration with the device.

4 Getting Started

This section provides a straightforward guide for integrating any of the Teleshake (95) AC USB device variants with a simple example focused on detecting the device. This will help new users to quickly set up their development environment and confirm their device's connectivity.

4.1 Thread Safety Restrictions (Important Note)

The provided supporting libraries are following an open architecture. This leads to the result, that there are no “thread safety restrictions” implemented. Therefore, the integrator is responsible to implement its own thread-safe code. It is recommended to combine a write operation with a corresponding read operation, before starting a new write operation.

4.2 Initial Setup

Before start, it is necessary to ensure that the device-specific DLLs are installed in the directory of your choice. This guide uses the following installation directory as an example:

```
const string INSTALLATION_DIRECTORY = @"C:\Program Files\Inheco.TeleshakeAcUsb.Win-x64.Library";
```

4.3 Preparing the Codebase for Device Communication

Reference Necessary DLLs

In your .NET project, reference the “Inheco.TeleshakeAcUsb.Adapter” namespace to interact with the device. Additionally, use “System.Runtime.InteropServices” to work with native libraries:

```
using Inheco.TeleshakeAcUsb.Adapter;
using System.Runtime.InteropServices;
```

Set DLL Directory

Use the “SetDllDirectory” method to specify the directory where the device-specific DLLs are located. This ensures that the application can locate and use these DLLs at runtime:

```
[DllImport("kernel32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
public static extern bool SetDllDirectory(string lpPathName);
```

4.4 Finding Connected Devices

The main objective in this step is to initialize the device context and search for connected devices of the Teleshake (95) AC USB product family.

Initialize Device Adapter

Create an instance of the “Context” class from the “Inheco.TeleshakeAcUsb.Adapter” namespace. This class serves as the central point for managing device interactions:

```
var adapter = new Context();
```

Set Installation Directory

Direct the application to the directory where the DLLs are installed. This is crucial for the application to function properly:

```
SetDllDirectory(INSTALLATION_DIRECTORY);
```

Setup Adapter and Search for Devices

Call the “Setup” method to initialize the adapter's internal components. Then, subscribe to the “MessageHandler” event to receive and print messages. Finally, use “SearchForDevicesAsync” to asynchronously search for connected devices:

```
adapter.Setup();
adapter.HidDeviceManager.MessageHandler += (sender, e) => Console.WriteLine(e.Message);
await adapter.HidDeviceManager.SearchForDevicesAsync();
```

This code snippet demonstrates the most basic setup required to detect a device connected to your system. It provides a quick way to verify device connectivity and ensures that your development environment is correctly configured.

5 API Integration Strategies

This chapter outlines two recommended approaches for integrating the Teleshake (95) AC USB device into your applications, utilizing the provided .NET Standard 2.0 library package. These approaches are designed to accommodate different integration preferences and requirements, ensuring flexibility in how the device's functionalities are accessed and utilized within your software solutions.

5.1 Recommended Integration Approach

There are two primary paths for integrating the Teleshake (95) AC USB device libraries into a custom solution or project (Figure 1).

Path A: Using the “Inheco.TeleshakeAcUsb.Adapter.dll”

This approach leverages the “Setup” method (as demonstrated in the provided code sample) and encapsulates the Teleshake AC USB DLLs, including an Inversion of Control (IoC) container. It offers a simplified integration process by abstracting the complexities involved in managing device communication and dependencies.

Path B: Direct Reference to “Inheco.TeleshakeAcUsb.dll”

For developers preferring or requiring direct control over the services used within the “Context” class's source code, this path involves directly referencing the “Inheco.TeleshakeAcUsb.dll”. Integration under this approach requires registering the services used by the “Context” class in your own IoC container.

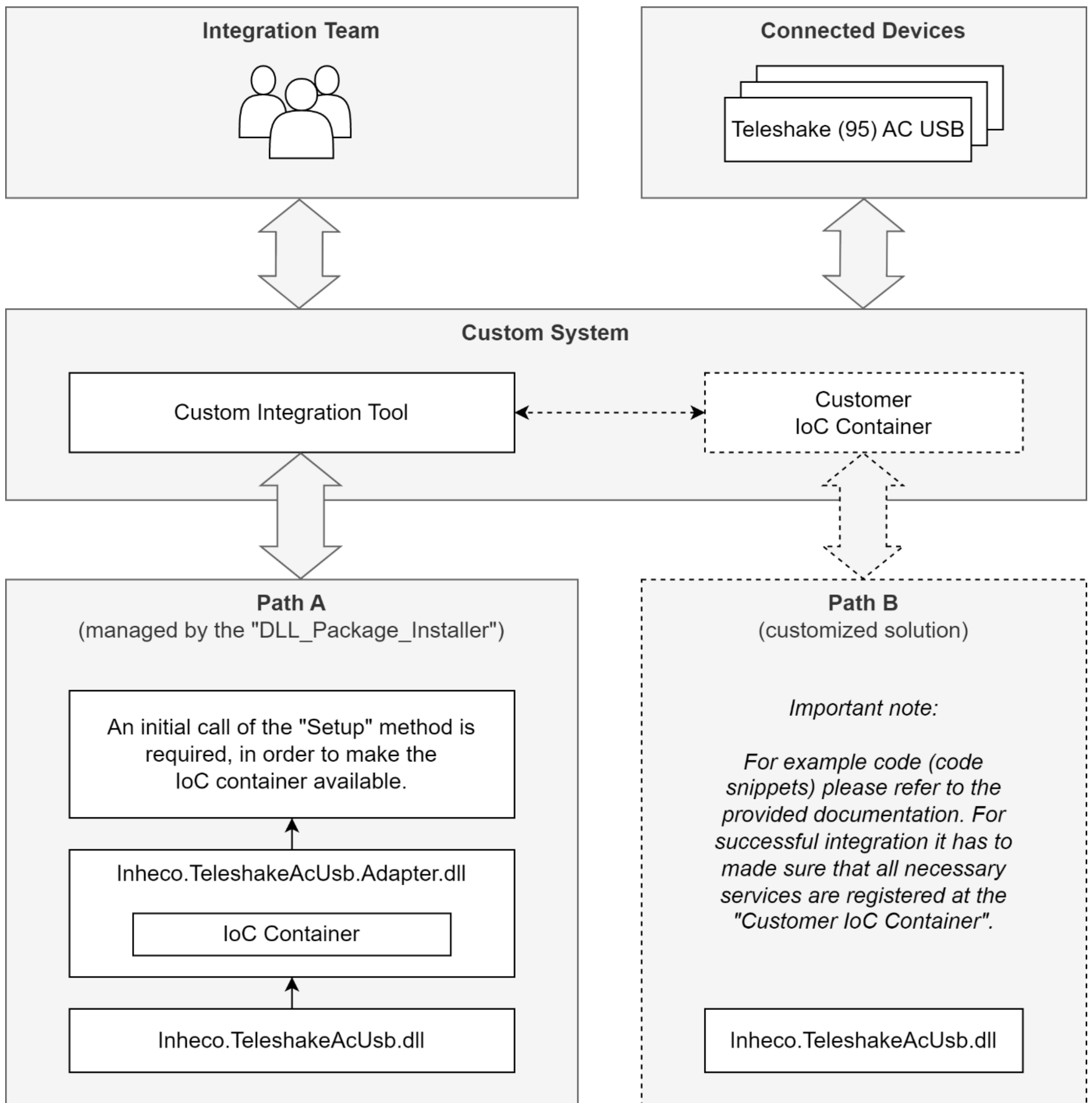

Figure 1: Abstracted block diagram to visualize two ways of possible device integration into a custom system.

5.2 Teleshake (95) AC USB Library Package

The Teleshake (95) AC USB Library Package comprises a set of .NET Standard 2.0 libraries essential for interacting with the device. These include:

Core and Wrapper DLLs

- Inheco.TeleshakeAcUsb.dll (Core DLL)
- Inheco.HidApi.dll, Inheco.HidDevice.Shared.dll (Wrapper DLLs for the native hidapi.dll)

Dependent Libraries

The package also contains dependencies that vary based on compilation factors such as operating system, processor architecture, and whether the environment is 32-bit or 64-bit. Notably, the native component of the package for different platforms must be stored under specific directories like “win-X64” or “win-X86” for Windows, “linux-X86” or “linux-X64” (and potentially ARM directories) for Linux, each containing the respective “hidapi.dll” or “libhidapi-libusb.so.0”. This organization addresses the need for different native DLLs due to naming conflicts and architectural differences.

Please note: All DLLs within this package are mandatory for integration, and no files or folders should be removed to ensure full functionality. Therefore, please make sure that you have all files available before debugging your own solution.

Customer Firmware Command Set

The complete set of commands available for the Teleshake (95) AC USB device family is documented in the “Customer Firmware Command Set” (CFWCS). The CFWCS serves as a comprehensive guide to the commands supported by the device's firmware, facilitating custom operations and advanced device management. Please find more information about the supporting documents in chapter 9.

5.3 Teleshake AC USB Adapter (Adapter DLL)

The “Teleshake AC USB Adapter”, a .NET Standard 2.0 library, includes an IoC Container that is initiated by calling the “Setup(bool useFileLogger=false)” method of the “Context” class. Upon initialization, you can access all interfaces provided by the “Context” class or, more broadly, all interfaces offered by the “Inheco.TeleshakeAcUsb.dll”. This is achieved by resolving services through the “Container” interface, as shown:

```
IScriptingService scriptingService = IoCContainer.Resolve<IScriptingService>();
```

The “Context” class encapsulates the device's functionalities and manages communication with the device. It initializes the Inversion of Control (IoC) Container, configures services, and facilitates the execution of commands to the device. This example highlights the initialization of essential services, registration of device-specific commands, and execution of those commands, showcasing the flexibility and power of our integration library.

Code Example: Teleshake (95) AC USB Adapter Context Class

```
using DryIoc;
using Inheco.HidApi;
using Inheco.HidApi.Services;
using Inheco.HidApi.Services.Search;
using Inheco.HidDevice.Shared;
using Inheco.HidDevice.Shared.Domain;
using Inheco.TeleshakeAcUsb.CommandSet;
using Inheco.TeleshakeAcUsb.Communication;
using Inheco.TeleshakeAcUsb.CompositeCommands;
using Inheco.TeleshakeAcUsb.Services;
using log4net;
using log4net.Config;
using Moq;
using System.Collections.Generic;
using System.IO;
using System.Reflection;
using System.Threading;
using System.Threading.Tasks;

namespace Inheco.TeleshakeAcUsb.Adapter
{
    public class Context
    {
        // Logger for diagnostic messages
        private ILog _logger;
        // Manages HID device connections
        private IHidDeviceManager _hidDeviceManager;
        // Factory for creating HID commands
        private IHidCommandFactory _hidCommandFactory;
        // Service for command definitions
        private ICommandDefinitionService _commandDefinitionService;
        // Service for transforming messages
        private IMessageTransformationService _messageTransformationService;
        // Service for file operations
        private IFileService _fileService;

        // Inversion of Control Container
        public Container IoCContainer { get; private set; }

        // List of connected USB devices
        public IList<IUsbDevice> Devices => _hidDeviceManager.Devices;

        // Directory where the executing assembly is located
        public string WorkingDirectory => Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);

        // Map of command definitions
        public            IDictionary<string,           CommandDefinition>           CommandMap           =>
_commandDefinitionService.CommandDefinitions;

        // Indicates if file logging is active
```

```
        public bool IsFileLoggerActive { get; private set; }

        // Access to the HID command factory
        public IHidCommandFactory CommandFactory => _hidCommandFactory;

        // Access to the message transformation service
        public IMessageTransformationService MessageTransformationService => _messageTransformationService;

        // Access to the file service
        public IFileService FileService => _fileService;

        // Access to the HID device manager
        public IHidDeviceManager HidDeviceManager => _hidDeviceManager;

        // Setup method initializes IoC container and services
        public void Setup(bool useFileLogger=false)
        {
            IsFileLoggerActive = useFileLogger;
            IoCContainer = new Container();
            ConfigureServices(IoCContainer);
            InitializeServices();
        }

        // Executes a command asynchronously on a device
        public async Task<IResponse> ExecuteCommandAsync(IUsbDevice device, string command, string parameters)
        {
            var request = _hidCommandFactory.GetRequest(command, parameters);
            var cts = new CancellationTokenSource();
            cts.CancelAfter(3000); // Timeout for command execution
            byte[]   responseResult   =   await   _hidDeviceManager.ResponseByRequest(device.Path,   request,
cts.Token);
            return new Response(responseResult);
        }

        // Synchronous wrapper for executing a command
        public IResponse ExecuteCommand(IUsbDevice device, string command, string parameters)
        {
            return ExecuteCommandAsync(device, command, parameters).Result;
        }

        // Initializes services after IoC container configuration
        private void InitializeServices()
        {
            _hidDeviceManager = IoCContainer.Resolve<IHidDeviceManager>();
            _hidDeviceManager.SearchForDevices();
            _hidCommandFactory = IoCContainer.Resolve<IHidCommandFactory>();
            _commandDefinitionService = IoCContainer.Resolve<ICommandDefinitionService>();
            _messageTransformationService = IoCContainer.Resolve<IMessageTransformationService>();
            _fileService = IoCContainer.Resolve<IFileService>();

            // Register message handlers for logging
            _hidDeviceManager.MessageHandler += (s, e) => _logger.Debug(e.Message);
            _hidCommandFactory.MessageHandler += (s, e) => _logger.Debug(e.Message);
            IoCContainer.Resolve<IFileService>().MessageHandler += (s, e) => _logger.Debug(e.Message);
            IoCContainer.Resolve<IScriptService>().MessageHandler += (s, e) => _logger.Debug(e.Message);
        }

// Configures services within the IoC container
        private void ConfigureServices(Container iocContainer)
        {
            _logger = IsFileLoggerActive ? GetLogger() : new Mock<ILog>().Object;

            iocContainer.RegisterInstance(_logger);
            _logger.Info("Inheco.Bootstrapper started");
            // Register all necessary services and their implementations
            iocContainer.Register<ICommandDefinitionService, CommandDefinitionService>(Reuse.Singleton);
            iocContainer.Register<IHidCommandFactory, HidCommandFactory>(Reuse.Singleton);
            // Additional service registrations omitted for brevity
        }

        // Configures and returns a logger
        public ILog GetLogger()
        {
```

```
            XmlConfigurator.Configure(new FileInfo("log4net.config"));
            return LogManager.GetLogger(GetType().Assembly, "Root");
        }
    }
}
```

Understanding the Code

IoC Container Initialization: The “Setup” method initializes the IoC Container, which is central to managing dependencies and service lifetimes within the application. It allows for flexible, decoupled design patterns that simplify maintenance and testing.

Service Registration: The “Service Registration” demonstrates how various services related to device management, command processing, and response handling are registered within the IoC Container. This step is crucial for ensuring that all components of the application can access the functionalities they require.

Executing Commands: In the section “Executing Commands” the execution of commands on the device by preparing a request, sending it, and processing the response is presented. This operation exemplifies interaction with the device, allowing for a wide range of tasks to be performed, from querying device status to executing specific actions like shaking or temperature control (if available).

Logging and Diagnostics: Using “log4net” for logging purposes illustrates how to incorporate robust logging mechanisms into your application, facilitating debugging and monitoring of the integration process.

6 Advanced Integration Examples

This chapter delves into advanced integration examples of the device, illustrating the device's comprehensive functionalities. Through a command-line application, various device operations can be explored, providing a practical guide to leveraging the library's full potential. The complete code is provided after the description of single parts.

6.1 Setting Up and Searching for Devices

```
// Initialize the adapter and set the DLL directory to ensure the application can find and use the Teleshake
DLLs.
var adapter = new Context();
SetDllDirectory(INSTALLATION_DIRECTORY);

// Begin device search and handle any messages during the process.
adapter.Setup(useFileLogger: true);
adapter.HidDeviceManager.MessageHandler += (sender, e) => Console.WriteLine(e.Message);
await adapter.HidDeviceManager.SearchForDevicesAsync();
```

Description: Initializes the device communication context, sets the library path (please insert your specific string), searches for connected devices of the Teleshake (95) AC USB family, and printing messages to the console.

6.2 Retrieving Device Information

```
// Example: Retrieving the device's serial number.
var device = adapter.HidDeviceManager.Devices.First();
Console.WriteLine($"Device Serial Number: {device.ShortName}");
```

Description: Basic demonstration of fetching and displaying the serial number of (only) the first found device.

6.3 Executing Commands and Handling Responses

In the code examples for integrating the Teleshake (95) AC USB devices, there are two demonstrated methods for executing commands: using the “ExecutionService” and the “CommandFactory”.

ExecutionService: This method provides a higher-level, more abstracted way to interact with the device, simplifying command execution. By resolving the “IExecutionService“ from the IoC container, you can call methods directly related to device operations, such as getting the device's article number. This approach encapsulates the command details, offering a straightforward interface for common tasks. A detailed documentation of the “ExecutionService” class is provided in chapter 6.8.

CommandFactory: This method offers a lower-level, more granular control over command execution. By creating a specific request with the “CommandFactory” and executing it through the device management application, you gain the flexibility to craft and send custom commands to the device. This approach is useful for operations that require specific command parameters or when you need to handle the device's response directly.

The full set of commands available for the “CommandFactory” can be found in the “Customer Firmware Command Set” (CFWCS). Also refer to the supporting documents in chapter 9.

Each method has its use cases, with the “ExecutionService” being ideal for standard operations and the “CommandFactory” for custom command scenarios.

Using the **ExecutionService**:

```
// Getting the Article Number using ExecutionService
Console.WriteLine("Getting Device Article Number using ExecutionService...");
var executionService = adapter.IoCContainer.Resolve<IExecutionService>();
string articleNumber = await executionService.GetArticleNumber(device, default);
Console.WriteLine($"Device Article Number: {articleNumber}");
```

Description: Shows a high-level approach where the “ExecutionService” directly provides a method to get the device's article number, abstracting the underlying command details.

Using the **CommandFactory**:

```
// Getting the Article Number using CommandFactory
var cmd = "RAN";
Console.WriteLine("Getting Device Article Number using CommandFactory...");
var request = adapter.CommandFactory.GetRequest(cmd, NiceType.Empty);
var response = await adapter.ExecuteCommandAsync(device, cmd, "");
Console.WriteLine($"Response:                    {adapter.MessageTransformationService.BeautifyResponse(new
Response(response.Bytes), true)}");
```

6.4 Managing Device Operations

```
// Example: Checking and changing the clamp status.
var clampStatus = await executionService.GetClampStatus(device, new CancellationTokenSource(3000).Token);
Console.WriteLine($"Clamp Status: {clampStatus}");
```

Description: Details checking the current status of the device's “Active Clamping” mechanism and provides an example of how to open or close it based on the current state.

6.5 Performing a Shaking Test

```
// Initiating a shaking test by setting the target rotation speed.
await executionService.SetTargetRotation(device, 400, new CancellationTokenSource(1000).Token);
Console.WriteLine("Starting Rotation...");
```

Description: Initiates a test to set the device's shaking speed, showcasing the command execution for starting the rotation.

6.6 Reading and Setting Temperature

```
// Example: Reading the current temperature and setting a new target temperature.
var currentTemperature = await executionService.GetCurrentTemperature(device, default);
Console.WriteLine($"Current Temperature: {currentTemperature}");
await      executionService.SetTargetTemperature(device,      currentTemperature      +      5.0,      new
CancellationTokenSource(1000).Token);
```

Description: Illustrates how to read the current temperature from the device (Teleshake 95 AC USB only) and set a new target temperature, demonstrating the device's temperature control capabilities, here by adding a temperature step of 5.0 K to the current temperature.

6.7 Comprehensive Integration Example Code

Within this chapter a comprehensive example code is provided illustrating the full process of integrating the device, from setting up the environment and connecting to the device to executing various functionalities using both “Execution Service” and “Command Factory” methods.

```
using System.Runtime.InteropServices;
using DryIoc;
using Inheco.TeleshakeAcUsb.Adapter;
using Inheco.TeleshakeAcUsb.Communication;
using Inheco.TeleshakeAcUsb.Services;

namespace Inheco.TeleshakeAcUsb.Integration.CmdLine;

internal class Program
{
    const string INSTALLATION_DIRECTORY = @"C:\Temp\Inheco.TeleshakeAcUsb.Win-x64.Library";
    [DllImport("kernel32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
    [return: MarshalAs(UnmanagedType.Bool)]

    public static extern bool SetDllDirectory(string lpPathName);

    static void Assert(Func<bool> condition, string message)
    {
        if (condition())
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine($"{message}: passed");
        }
        else
        {
            // If the condition is false, print the message in red
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine($"{message}: failed");
        }
        // Reset the console color to its default
        Console.ResetColor();
    }

    static async Task Main(string[] args)
    {
        try
        {
            Console.WriteLine($"{DateTime.Now} Inheco.TeleshakeAcUsb.CmdLine.App started\n");

            // Instantiate adapter for interacting with the HID devices.
            var adapter = new Context();
            Console.WriteLine($"Working Directory -> '{adapter.WorkingDirectory}'\n");

            // Set the directory defined by INSTALLATION_DIRECTORY as the search path for DLLs,
            // ensuring that any subsequent DLL imports look in this directory
            SetDllDirectory(INSTALLATION_DIRECTORY);

            Console.WriteLine("Initializing IoC-Container...\n");
            adapter.Setup(useFileLogger: true);

            // Subscribe to the MessageHandler event of the HidDeviceManager within the adapter.
            // Whenever a message event occurs, it prints the message to the console.
            adapter.HidDeviceManager.MessageHandler += (sender, e) => Console.WriteLine(e.Message); ;

            Console.WriteLine("Searching for Teleshake AC USB  device...\n");

            // search for HID devices
            await adapter.HidDeviceManager.SearchForDevicesAsync();

            // Assert that at least one device has been found
            if (adapter.HidDeviceManager.Devices.Count == 0)
            {
```

```
                Console.WriteLine("No devices found. Exiting application.");
                return;
            }
            Assert(() => adapter.HidDeviceManager.Devices.Count > 0, "At least one device has been found");
            Console.WriteLine($"{adapter.HidDeviceManager.Devices.Count} devices found.\n");

            // Device Serial Number
            Console.WriteLine("Getting Device Serial Number...");
            var device = adapter.HidDeviceManager.Devices.First();
            Assert(() => !string.IsNullOrEmpty(device.ShortName), "Device Serial Number should not be empty");
            Console.WriteLine($"Device Serial Number: {device.ShortName}\n");

            // Device Article Number
            Console.WriteLine("Getting Device Article Number using ExecutionService...");
            var executionService = adapter.IoCContainer.Resolve<IExecutionService>();
            string articleNumber = await executionService.GetArticleNumber(device, default);
            Assert(() => !string.IsNullOrEmpty(articleNumber), "Device Article Number should not be empty");
            Console.WriteLine($"Device Article Number: {articleNumber}");

            var cmd = "RAN";
            Console.WriteLine("Getting Device Article Number using CommandFactory...");
            var request = adapter.CommandFactory.GetRequest(cmd, NiceType.Empty);
            Console.WriteLine($"Request:       {adapter.MessageTransformationService.BeautifyRequest(request,
true)}");
            var response = await adapter.ExecuteCommandAsync(device, cmd, "");
            Console.WriteLine($"Response:          {adapter.MessageTransformationService.BeautifyResponse(new
Response(response.Bytes), true)}\n");

            // Firmware Version
            cmd = "RFV";
            Console.WriteLine("Getting Firmware Version using CommandFactory...");
            request = adapter.CommandFactory.GetRequest(cmd, NiceType.Empty);
            Console.WriteLine($"Request:       {adapter.MessageTransformationService.BeautifyRequest(request,
true)}");
            response = await adapter.ExecuteCommandAsync(device, cmd, "");
            var res = adapter.MessageTransformationService.MakeNiceResponse(response);
            Assert(() => res.Count > 0, "Response should not be empty");
            for (int i = 0; i < res.Count; i++)
            {
                Console.WriteLine($"Response [{i}]: {res[i].Value}");
            }

            // Clamp Status
            Console.WriteLine("\nGetting Clamp Status...");
            var        clampStatus        =        await        executionService.GetClampStatus(device,        new
CancellationTokenSource(3000).Token);
            Assert(() => clampStatus == Device.ClampStatus.Closed || clampStatus == Device.ClampStatus.Open ||
clampStatus == Device.ClampStatus.Error,
                "Clamp Status is Open or Closed");
            Console.WriteLine($"Clamp Status: {clampStatus}");
            if (clampStatus == Device.ClampStatus.Open)
            {
                Console.WriteLine("Closing Clamps...");
                var        ccResponse        =        await        executionService.CloseClamps(device,        new
CancellationTokenSource(3000).Token);
                await Task.Delay(3000);
                clampStatus         =         await         executionService.GetClampStatus(device,         new
CancellationTokenSource(3000).Token);
                Assert(() => clampStatus == Device.ClampStatus.Closed, "Clamp Status is Closed");

                Console.WriteLine("Opening Clamps again...");
                ccResponse          =          await          executionService.OpenClamps(device,          new
CancellationTokenSource(3000).Token);
                await Task.Delay(3000);
                clampStatus         =         await         executionService.GetClampStatus(device,         new
CancellationTokenSource(3000).Token);
                Assert(() => clampStatus == Device.ClampStatus.Open, "Clamp Status is Open");
            }
            else
            {
                Console.WriteLine("Opening Clamps...");
                var        ocResponse        =        await        executionService.OpenClamps(device,        new
CancellationTokenSource(3000).Token);
```

```
            await Task.Delay(3000);
            clampStatus = await executionService.GetClampStatus(device, new
CancellationTokenSource(3000).Token);
            Assert(() => clampStatus == Device.ClampStatus.Open, "Clamp Status is Open");

            Console.WriteLine("Closing Clamps again...");
            ocResponse = await executionService.CloseClamps(device, new
CancellationTokenSource(3000).Token);
            await Task.Delay(3000);
            clampStatus = await executionService.GetClampStatus(device, new
CancellationTokenSource(3000).Token);
            Assert(() => clampStatus == Device.ClampStatus.Closed, "Clamp Status is Closed");
        }

        Console.WriteLine("\nPress any key to start Shaking test.");
        Console.ReadKey();
        // Shaking
        // close clamps if open
        clampStatus = await executionService.GetClampStatus(device, new
CancellationTokenSource(3000).Token);
        if (clampStatus == Device.ClampStatus.Open)
        {
            await executionService.CloseClamps(device, new CancellationTokenSource(3000).Token);
            await Task.Delay(3000);
        }

        Console.WriteLine("Setting Target Rotation to 400 RPM...");
        response = await executionService.SetTargetRotation(device, 400, new
CancellationTokenSource(1000).Token);
        res = adapter.MessageTransformationService.MakeNiceResponse(response);
        Assert(() => res.Count > 0, "Response should not be empty");
        Console.WriteLine($"Set Target Rotation Response: {res.First().Value}");

        Console.WriteLine("Starting Rotation...");
        response = await executionService.StartRotation(device, new CancellationTokenSource(3000).Token);
        res = adapter.MessageTransformationService.MakeNiceResponse(response);
        Assert(() => res.Count > 0, "Response should not be empty");
        Console.WriteLine($"Start Rotation Response: {res.First().Value}");
        Console.WriteLine($"Waiting for 180s");
        await Task.Delay(5000);

        cmd = "RSR";
        Console.WriteLine("Reading Rotation Status...");
        request = adapter.CommandFactory.GetRequest(cmd, NiceType.Empty);
        response = await adapter.ExecuteCommandAsync(device, cmd, "");
        res = adapter.MessageTransformationService.MakeNiceResponse(response);
        Assert(() => res.Count > 0, "Response should not be empty");
        Console.WriteLine($"Rotation Speed: {res.First().Value}");
        response = await executionService.StopRotation(device, new CancellationTokenSource(1000).Token);
        res = adapter.MessageTransformationService.MakeNiceResponse(response);
        Assert(() => res.Count > 0, "Response should not be empty");
        Console.WriteLine($"Stop Rotation Response: {res.First().Value}");
        await Task.Delay(10000);
        Console.WriteLine("Opening Clamps...");
        await executionService.OpenClamps(device, new CancellationTokenSource(3000).Token);
        await Task.Delay(3000);
        clampStatus = await executionService.GetClampStatus(device, new
CancellationTokenSource(3000).Token);
        Assert(() => clampStatus == Device.ClampStatus.Open, "Clamp Status is Open");

        // Temperature
        await executionService.CloseClamps(device, new CancellationTokenSource(3000).Token);
        await Task.Delay(3000);
        cmd = "RAT";
        Console.WriteLine("Reading Current Temperature...");
        request = adapter.CommandFactory.GetRequest(cmd, NiceType.Empty);
        response = await adapter.ExecuteCommandAsync(device, cmd, "");
        res = adapter.MessageTransformationService.MakeNiceResponse(response);
        Assert(() => res.Count > 0, "Response should not be empty");
        var roomTemperature = (float)res[1].Value;
        for (int i = 0; i < res.Count; i++)
        {
            Console.WriteLine($"Response [{i}]: {res[i].Value}");
```

```
            }

            Console.WriteLine($"Setting Target Temperature to {roomTemperature + 5}...");
            await     executionService.SetTargetTemperature(device,     roomTemperature     +     5.0,     new
CancellationTokenSource(1000).Token);
            response = await executionService.StartHeating(device, new CancellationTokenSource(1000).Token);
            res = adapter.MessageTransformationService.MakeNiceResponse(response);
            Assert(() => res.Count > 0, "Response should not be empty");

            Console.WriteLine($"Waiting for 180s");
            await Task.Delay(5000);
            Console.WriteLine("Reading Current Temperature...");
            request = adapter.CommandFactory.GetRequest(cmd, NiceType.Empty);
            response = await adapter.ExecuteCommandAsync(device, cmd, "");
            res = adapter.MessageTransformationService.MakeNiceResponse(response);
            var newTemperature = (float)res[1].Value;
            Assert(() => newTemperature >= roomTemperature + 5, "Response should not be empty");
            for (int i = 0; i < res.Count; i++)
            {
                Console.WriteLine($"Response [{i}]: {res[i].Value}");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex);
        }

        Assert(() => true, "\nAll tests");
        Console.WriteLine("\nPress any key to exit.");
        Console.ReadKey();
    }
}
```

6.8 ExecutionService Class Documentation

The “IExecutionService” interface defines a set of asynchronous tasks for executing commands and managing states of USB devices in a high-level manner. It is designed to handle various operational commands, such as executing specific actions, managing device status, and retrieving device information. Please note that the low-level commands are documented in the “Customer Firmware Command Set” (CFWCS).

The “ExecutionService” class offers a concrete implementation of the “IExecutionService” interface and can be resolved using “Inheco.TeleshakeAcUsb.Adapter” as shown in the example above.

Here is a list of methods implemented by the “ExecutionService” class, along with brief descriptions and parameters:

ExecuteAsync

Executes a command asynchronously on a specified USB device.

```
Task<IResponse> ExecuteAsync(IUsbDevice device, string cmd, string parameters, CancellationToken token, int
delayInMilliseconds = 0);

Task<IResponse> ExecuteAsync(IUsbDevice device, string cmd, byte[] parameters, CancellationToken token, int
delayInMilliseconds = 0);
```

Parameter	Description
IUsbDevice device	The target USB device.
string cmd	The command to execute.
string parameters byte[] parameters	The parameters for the command, in string or byte array format.
CancellationToken token	Token for cancelling the task.
int delayInMilliseconds (optional)	The delay before executing the command, in milliseconds.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object containing the response from the device.

GetPowerSupplyValue

Returns the current supply voltage value measured by the USB device. Before clamping, shaking, or heating make sure to verify that the power supply value is within the nominal range of 24 V ± 10%. If this is not the case, then the USB device might not function properly.

```
Task<float> GetPowerSupplyValue(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	The power supply value measured by the USB Device. This value needs to be around 24V ± 10% for the device to function properly.

HasPowerSupplyAsync

Checks if the device is connected to a proper DC power supply with a voltage level of +24 V (±10%) and a maximum current of 6.3 A.

```
Task<bool> HasPowerSupplyAsync(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	Returns ‘true’ if the device is connected to a proper DC power supply with a voltage level of +24 V (±10%) and a maximum current of 6.3 A.

ToggleClampStatus

Toggles the clamp status of a specified USB device asynchronously.

```
Task<ClampStatus> ToggleClampStatus(IUsbDevice device, bool shouldClose, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
bool shouldClose	A boolean indicating whether the clamp should be closed (true) or opened (false).
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields a “ClampStatus” indicating the result of the operation.

GetClampStatus

Retrieves the current clamp status of a specified USB device.

```
Task<ClampStatus> GetClampStatus(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields a “ClampStatus” indicating the current status of the clamp.

SetTargetTemperature

Sets the target temperature (in °C) of a specified USB device.

```
Task SetTargetTemperature(IUsbDevice device, double targetTemperature, CancellationToken token);
Task SetTargetTemperature(IUsbDevice device, double targetTemperature, double boostDelta, int boostIntervalInSeconds, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
double targetTemperature	The target temperature to set.
CancellationToken token	Token for cancelling the task.
double boostDelta	Represents the temporary increase in the target temperature, above the initially set target temperature. It is the difference between the temporary boosted target temperature and the original target temperature. For example, if the target temperature is set to 20°C and “boostDelta” is 5 K, then the device will initially aim to reach 25°C before settling back to 20°C.
int boostIntervalInSeconds	Specifies the duration for which the boosted target temperature (original target temperature + “boostDelta”) should be maintained before returning to the original target temperature. It is measured in seconds. For instance, if “boostIntervalInSeconds” is set to 300, the device will maintain the boosted temperature for 5 minutes (= 300 seconds) before reverting to the target temperature.

StartHeating

Starts the active heating process of a specified USB device.

```
Task<IResponse> StartHeating(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

StopHeating

Stops the active heating process of a specified USB device.

```
Task<IResponse> StopHeating(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

SetTargetRotation

Sets the target rotation speed for a specified USB device. This method configures the device to rotate at the specified speed.

```
Task<IResponse> SetTargetRotation(IUsbDevice device, int targetRotation, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
int targetRotation	The target rotation speed.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

StartRotation

Initiates the rotation of a specified USB device. This method starts the device's rotation at the previously set target rotation speed.

```
Task<IResponse> StartRotation(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

StopRotation

Stops the rotation of a specified USB device. This method halts any ongoing rotation of the device, bringing it to a standstill with respect to a deceleration rate.

```
Task<IResponse> StopRotation(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

GetFirmwareVersion

Retrieves the firmware version of a specified USB device.

```
Task<String> GetFirmwareVersion(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that yields the firmware version as a string.

GetArticleNumber

Retrieves the article number of a specified USB device.

```
Task<String> GetArticleNumber(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that yields the article number as a string.

CloseClamps

Closes the clamps of a specified USB device.

```
Task<IResponse> CloseClamps(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

OpenClamps

Opens the clamps of a specified USB device.

```
Task<IResponse> OpenClamps(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

GetErrorDetails

Retrieves detailed error information from a specified USB device. The full error list can be viewed in the “Customer Firmware Command Set”. Please find more information about the supporting documents in chapter 9 Support and Resources.

```
Task<string> GetErrorDetails(IUsbDevice device, string inputError, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
string inputError	The error input for which details are requested.
CancellationToken token	Token for cancelling the task.
Returns	A task that yields a detailed error message as a string.

GetDiagnosticCounter

Retrieves the diagnostic counter value from a specified USB device.

```
Task<string> GetDiagnosticCounter(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

IsHeaterEnabled

Checks if heater is enabled.

```
Task<bool?> IsHeaterEnabled (IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	- True if heater is enabled - False if heater is not enabled - Null it the status could not be retrieved due to an error

IsShakerEnabled

Checks if shaker is enabled.

```
Task<bool?> IsShakerEnabled (IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	- True if shaker is enabled - False if shaker is not enabled - Null it the status could not be retrieved due to an error

IsPositionerEnabled

Checks if the positioning functionality (angle in milli°) is enabled or disabled.

```
Task<bool?> IsPositionerEnabled (IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	True if positioner is enabled False if positioner is not enabled Null it the status could not be retrieved due to an error

StartPositioner

Enables or disables the position control. The clamps need to be closed during positioning and the shaker needs to be inactive. The device will try to reach and hold the entered position. This feature is intended to be used for pipetting operations.

```
Task<IResponse> StartPositioner(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

SetTargetPosition

Sets the positioning angle, in milli°.

```
Task<IResponse> SetTargetPosition(IUsbDevice device, int targetPosition, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

GetPosition

Reports the actual position in milli°.

```
Task<IResponse> GetPosition(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	Clamp Activations Counter Value.

StopPositioner

Stops the positioner of a specified USB device. This method halts any ongoing positioning of the device.

```
Task<IResponse> StopPositioner (IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

GetTemperature1

Reports the actual temperature value recorded by the control sensor. Note: This sensor provides the feedback value for the closed-loop temperature control.

```
Task<IResponse> GetTemperature1(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

GetTemperature2

Reports the actual temperature value recorded by the monitoring sensor. This can be used to verify that there are no major deviations between the control and the optional monitoring temperature sensor.

```
Task<IResponse> GetTemperature2(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

SetRotationDirection

Sets the shaking direction, clockwise or anti-clockwise.

```
Task<IResponse> SetRotationDirection(IUsbDevice device, bool ClockwiseDirection, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
Bool clockwiseDirection	Is clockwise direction.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

GetRotationSpeed

Reports the shaker revolutions per minute.

```
Task<IResponse> GetRotationSpeed(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

GetMotorCounter

Reports the on time of the shaker motor in milliseconds.

```
Task<string> GetMotorCounter(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	Motor Runtime Counter Value.

GetHeatingCounter

Reports the on time of the heating element in milliseconds.

```
Task<string> GetMotorCounter(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	Heater Runtime Counter Value.

GetClampCounter

Reports the total number of clamp activities.

```
Task<string> GetClampCounter(IUsbDevice device, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
CancellationToken token	Token for cancelling the task.
Returns	Clamp Activations Counter Value.

SetTargetPosition

Sets the target position in milli° of the shaker unit for closed loop control.

```
Task<IResponse> SetTargetPosition(IUsbDevice device, int targetPosition, CancellationToken token);
```

Parameter	Description
IUsbDevice device	The target USB device.
Int targetPosition	Target position.
CancellationToken token	Token for cancelling the task.
Returns	A task that represents the asynchronous operation and yields an “IResponse” object.

7 Additional Tools for Device Management and Development

This section introduces additional tools for device management and development with the Teleshake (95) AC USB. It covers the “Device Manager” for intuitive device interactions and the Command Line Tool for efficient firmware command execution, each offering unique functionalities to cater to different integration and development requirements.

7.1 Device Manager

The “Device Manager” for the Teleshake (95) AC USB is a Windows-based application designed to facilitate device discovery and management in a Plug 'n Play mode, allowing for seamless detection and integration of devices (Figure 2). It offers a user-friendly graphical interface for monitoring and adjusting device functionalities such as shaking, clamp operation, and heating (if selected). For detailed operation instructions and advanced features, please refer to the comprehensive documentation provided with the application. This reference material provides in-depth information on utilizing the “Device Manager” to its full potential, ensuring efficient device management and development support.

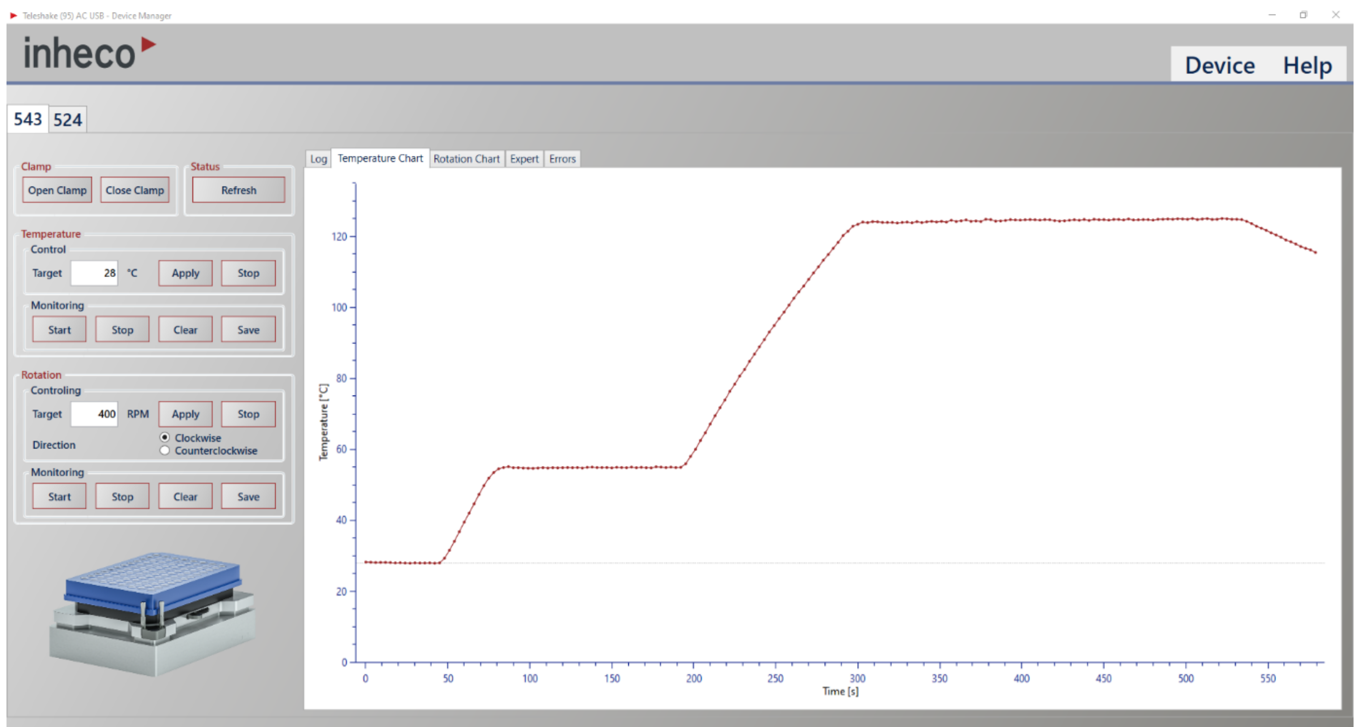

Figure 2: The application “Teleshake (95) AC USB Device Manager” enables visual operation of the device, along with logging, monitoring, process data storage and error reporting.

7.2 Command Line Tool

The “Command Line Tool for Firmware Commands Execution” is a compact, command-line executable included in the "Adapter Package”, designed for rapid integration without the need for a graphical user interface (Figure 3). This tool, which references the “Inheco.TeleshakeAcUsb.Adapter.dll”, can also be utilized to debug the integration into alternative development tools, e.g. in a NI LabVIEW project structure (32-bit only). The firmware commands and their details are extensively documented in the “Customer Firmware Command Set” (CFWCS), providing a robust resource for developers looking to leverage this tool for device control and integration (please find additional information in chapter 9).

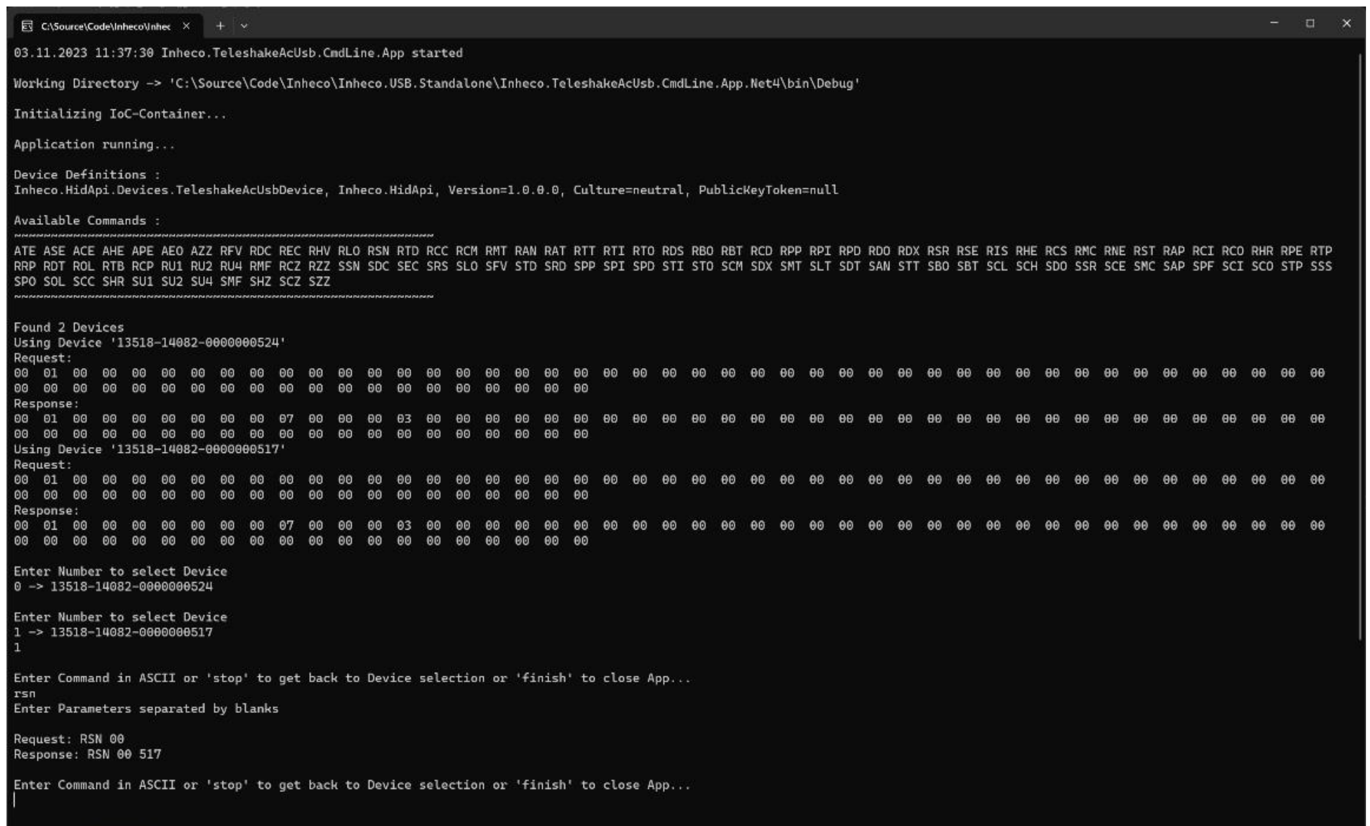

Figure 3: Demonstration of the “Command Line Tool for Firmware Commands Execution" for testing multiple devices (in this instance: two) using commands from the “Customer Firmware Command Set”

8 Troubleshooting

8.1 Device Connectivity Issues

Problem: Device not recognized by the host computer.

Solution: Ensure USB cables are securely connected, and the device is powered on. Check if the device appears in the system's device manager. If not, try reconnecting or testing with another USB port.

Problem: Clamping, shaking, and heating is not operating as expected after a command has been transmitted.

Solution: For operating the features of active clamping, shaking, and heating (optional), the device requires a connection to a 24 Vdc power supply unit, that can deliver up to 8 A per device.

8.2 Firmware Command Execution Failures

Problem: Commands return unexpected errors or no response.

Solution: Verify command syntax against the “Customer Firmware Command Set” (CFWCS) documentation. Ensure the latest device firmware is installed and compatible to the CFWCS version that is used. For detailed error descriptions, refer to the CFWCS.

8.3 Unexpected Device Behavior

Problem: The device behaves unpredictably or does not follow command specifications.

Solution: Power cycle the device and retest. If the issue persists, review the device logs for error messages. For specific error handling, consult the device manual.

8.4 Additional Support

In terms of issues that are not covered by this document for or more complex questions, please do not hesitate to contact the manufacturer. Also, please provide detailed information about your device, serial number, physical setup, the configuration that is leading to an issue, and any error messages received.

9 Support and Resources

If additional information is required, please consider accessing the "Inheco Customer Area" at https://www.inheco.com first. You can register for free through the menu button for immediate access. Once signed in, a comprehensive resource library is available at your fingertips, including FAQs, user manuals, software updates, and troubleshooting guides.

User Manuals: Access detailed manuals for your products, command sets for firmware, and extensive documentation, including guides for related accessories.

Software: Obtain the latest support software for your products, explore compatible program libraries (.dll), and discover demo software for various Inheco products.

Troubleshooting Form: Employ our Troubleshooting Form to gather product data that assists you in self-diagnosing issues or enhances our support to you.

Knowledge Base: Investigate well-documented solutions and answers that have aided other clients.

Error Code List: Consult our comprehensive list of error codes to uncover potential causes and actionable solutions for issues.

Product Registration: Register your Inheco product to activate your 2-Year Global Warranty. This also ensures you receive timely updates, relevant news, and essential information regarding your products.

10 Legal and Licensing

Software License Agreement

All software that is developed and published by Inheco is based on Inheco’s “End User License Agreement” (EULA). In case you do not agree with the license agreement, please contact Inheco to discuss an additional custom solution.

Third-Party Software Notices

Within the installation folder of each application software that contains third-party software licenses, a copy of the specific license is included.

11 Glossary

Teleshake Device: A specialized laboratory equipment used for mixing samples with both shaking and temperature control capabilities.

Execution Service: A service layer in software that executes defined tasks, commands, or functions in a managed way.

Command Factory: A design pattern used to encapsulate the creation logic of objects, allowing for the dynamic creation of different objects based on specific conditions or configurations.

HID (Human Interface Device): In the context of this integration, it refers to the communication protocol used for interacting with the Teleshake device.

IoC (Inversion of Control) Container: A design pattern used in software development where custom-written portions of a program receive their dependencies from external sources rather than creating them internally.

Context Class: The central class in the Teleshake library that manages device communication, service instantiation, and execution of commands.

Customer Firmware Command Set: The set of commands supported by the device's firmware, allowing for interaction and control over its functionalities.

Execution Command: A specific instruction sent to the Teleshake device to perform an operation, such as shaking or setting temperature.

Message Handler: A component or method designed to process incoming messages or events, commonly used in event-driven architectures.

Command Definition: Specifications or configurations that describe how commands should be structured and processed by the system.

Asynchronous Task: A method of executing operations concurrently with the main application thread, improving responsiveness and performance.

Cancellation Token: Used in asynchronous programming to provide the ability to cancel a running task or operation.