analytikjena
An Endress+Hauser Company

Operating manual

Biometra Library
.NET Framework API for integrating Biometra thermocyclers into a NET Framework application

Manufacturer	Analytik Jena GmbH+Co. KG Konrad-Zuse-Straße 1 07745 Jena · Germany Phone + 49 3641 77 70 Fax + 49 3641 77 92 79 Email info@analytik-jena.com
Service	Analytik Jena GmbH+Co. KG Konrad-Zuse-Straße 1 07745 Jena · Germany Phone + 49 3641 77 74 07 Fax + 49 3641 77 92 79 Email service@analytik-jena.com
General information	http://www.analytik-jena.com This operating manual is valid from the thermocycler software (firmware) version ME 2.00 – RE 2.00
Copyrights and trademarks	Microsoft and Windows are registered trademarks of Microsoft Corp. The identification with ® or TM is omitted in this operating manual.
Documentation number	34-8030-202-23
Edition	C (09/2023)
Technical documentation produced by	Analytik Jena GmbH+Co. KG

Contents

Figures

1 Basic information

1.1 Notes on this operating manual

The Biometra Library software is intended for use by qualified specialist personnel observing the operating instructions for the respective thermocycler and this operating manual for the Biometra Library software.

This operating manual provides information on how to operate the Biometra thermocyclers using the computer-aided Biometra Library software.

The compatible thermocycler models and firmware versions can be found in section 4.1.

Furthermore, the terms protocol and program are used synonymously, with both describing one or more temperature steps programmed in succession with different or identical hold times and cycles.

Symbols and signal words

The operating manual uses the following symbols and signal words to indicate hazards or instructions. Safety instructions are always placed before an action.

NOTE

Provides information on potential material or environmental damage.

1.2 Intended use

The Biometra Library software has been designed for the computer-aided operation of Biometra stand-alone thermocyclers Biometra TOne, Biometra TAdvanced and Biometra TRIO, as well as operation of the automated thermocycler Biometra TRobot II.

The intended use of the devices described in the thermocycler operating manual must be taken into account.

2 Safety instructions

2.1 General notes

For your own safety and to ensure the error-free and safe operation of the Biometra thermocycler, please read this chapter carefully before commissioning.

Besides the safety instructions in the operating manual for the Biometra thermocycler and the local safety regulations that apply to the operation of the device, the general applicable regulations regarding accident prevention, occupational health and safety, and environmental protection have to be observed and complied with.

References to potential hazards do not replace the work protection regulations which must be observed.

2.2 Requirements for the operating personnel

A Biometra thermocycler may only be operated by qualified specialist personnel instructed in the use of the device. The instruction also includes imparting the contents of this operating manual.

In addition to the safety at work instructions in the operating manual for the Biometra thermocycler, the generally applicable safety and accident prevention regulations of the respective country of operation must be observed and adhered to. The operator must ascertain the latest version of these regulations.

The operating manual must be accessible to the operating and service personnel at all times.

2.3 Behavior during emergencies

In case of danger or accidents, immediately switch off the Biometra thermocycler at the main switch. Disconnect the power plug from the power supply!

3 Function

3.1 Fields of application

The software was developed for the operation of certain Biometra thermocyclers in line with the field of application described in the device operating manual.

The software facilitates the simple and complete control of one or several thermocyclers in a network, with ongoing programs monitored and log files read out.

3.2 General description

The Biometra Library software covers the majority of the Biometra thermocycler functions and all functions of the automated Biometra TRobot II thermocycler. Three connection options are available for communication with the thermocycler: a serial RS232 interface, USB2.0 via an RS232 to USB adapter or Ethernet.

PCR programs, run log files and other files generated by the thermocycler can usually be stored in the internal memory of the thermocycler, the thermocycler controller or on the computer or server of the control software. Details can be found in the software description section.

In addition to operating the thermocycler, the Biometra Library software can also be used to exchange PCR programs between different Biometra thermocyclers in the same network. The Biometra Library software can also be used to save run log or service information files.

4 General

4.1 Field of application

The Biometra Library is a .NET Framework 4.0 DLL that can be used to control the thermocyclers connected to the end device.

If the thermocycler is to be controlled separately and not integrated into a NET Framework application, the Biometra TSuite software can be used instead.

The Biometra Library is compatible with the following thermocycler models:

- Biometra TRobot II, from firmware version ME 2.00 – RE 2.00
- Biometra TOne, Biometra TAdvanced, Biometra TRIO, from firmware versions > ME 2.20 – RE 2.20. Please contact our service department to update the firmware of existing devices.

Please refer to the Biometra TSuite Software Operating Manual for more detailed information on individual features and documentation file contents.

4.2 Scope of delivery

The Biometra Library is supplied with the Biometra TRobot II automated thermocycler on a CD-ROM or USB stick. The data medium contains the following files:

- Biometra Library
- Library documentation
- Biometra TSuite software
- Manuals (German and English) for the Biometra TRobot II thermocycler, Biometra TSuite computer software and Biometra Library

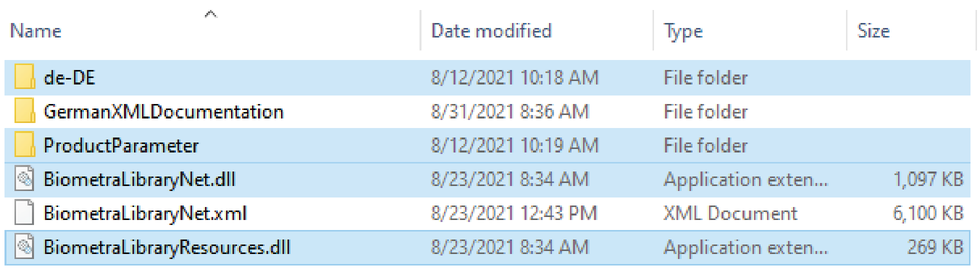
The Biometra TRobot II also comes with a license to use the protected Biometra TSuite computer software. The license is provided as a USB copy protection device (USB dongle).

4.3 Installation requirements

To use the library, at a minimum .NET Framework 4.0 must be installed on the computer. In addition, a development environment is required in which a .NET Framework application can be created, such as Microsoft Visual Studio.

4.4 Installation

The Biometra Library can be added to a .NET project in just a few steps. This sample project is a WPF app (.NET Framework). The following files and directories should first be selected and copied.

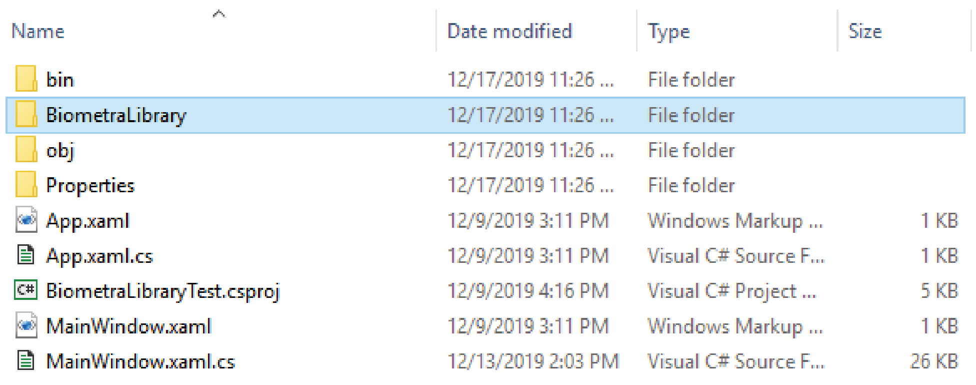


Name	Date modified	Type	Size
de-DE	8/12/2021 10:18 AM	File folder	
GermanXMLDocumentation	8/31/2021 8:36 AM	File folder	
ProductParameter	8/12/2021 10:19 AM	File folder	
BiometraLibraryNet.dll	8/23/2021 8:34 AM	Application exten...	1,097 KB
BiometraLibraryNet.xml	8/23/2021 12:43 PM	XML Document	6,100 KB
BiometraLibraryResources.dll	8/23/2021 8:34 AM	Application exten...	269 KB

Image 1: Biometra Library files and directories

Optionally, the English-language file "BiometraLibraryNet.xml" or the corresponding German-language file in the directory "GermanXMLDocumentation" can be added to the same directory as BiometraLibraryNet.dll in order to display the documentation in Visual Studio, e.g. via mouseover.

The copied files must be copied to the project directory of the application in which the Biometra Library is to be used. For a better overview, a separate subdirectory has been created for the required files and directories. The required files must now be copied into this subdirectory.



Name	Date modified	Type	Size
bin	12/17/2019 11:26 ...	File folder	
BiometraLibrary	12/17/2019 11:26 ...	File folder	
obj	12/17/2019 11:26 ...	File folder	
Properties	12/17/2019 11:26 ...	File folder	
App.xaml	12/9/2019 3:11 PM	Windows Markup ...	1 KB
App.xaml.cs	12/9/2019 3:11 PM	Visual C# Source F...	1 KB
BiometraLibraryTest.csproj	12/9/2019 4:16 PM	Visual C# Project ...	5 KB
MainWindow.xaml	12/9/2019 3:11 PM	Windows Markup ...	1 KB
MainWindow.xaml.cs	12/13/2019 2:03 PM	Visual C# Source F...	26 KB

Image 2: Project directory with subdirectory for the Biometra Library

After the files have been copied, the project (in this case "Biometra LibraryTest") can be opened and a reference to the Biometra Library can be created. To do this, select the menu item "References" in the project folder and choose the menu item "Add Reference..." in the corresponding context menu.

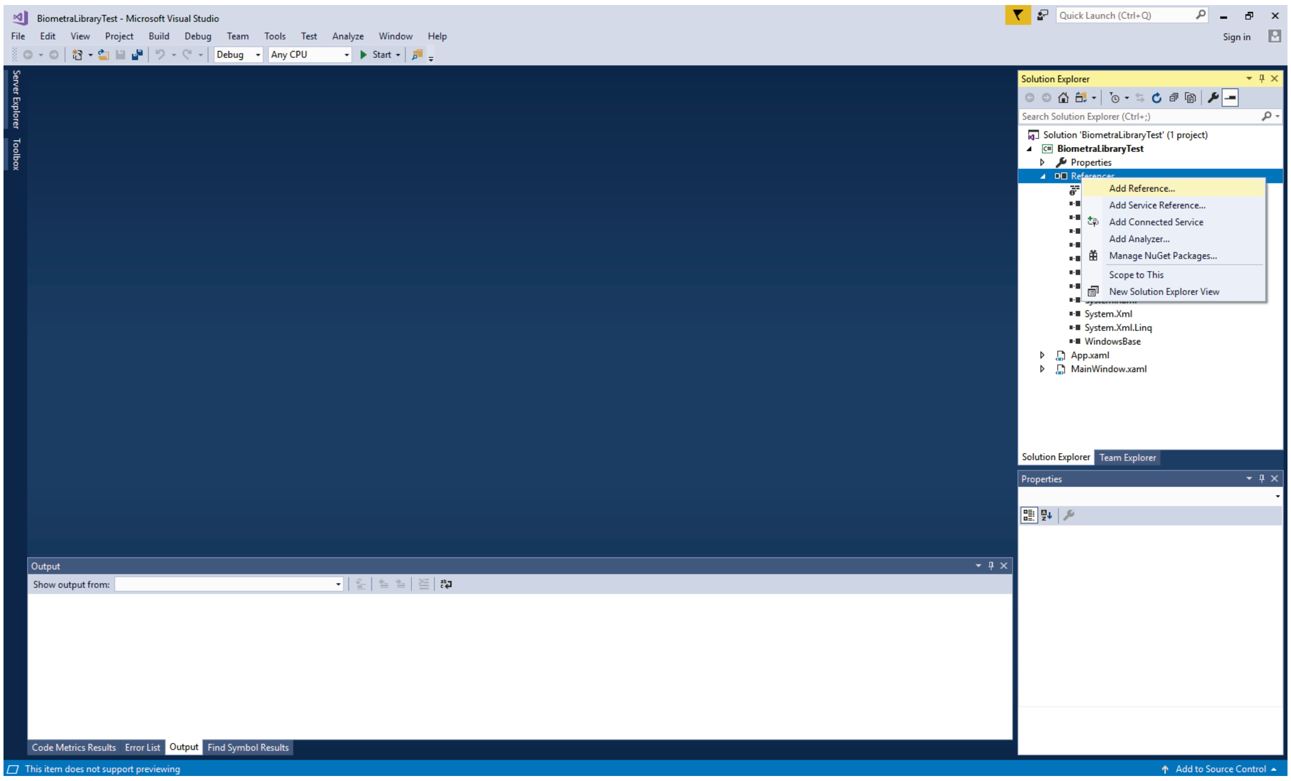


Image 3: Add a reference to the Biometra Library

In the window that opens, select "Browse". Use the "Browse..." button to select the path where the Biometra Library is located in order to create a reference to the library.

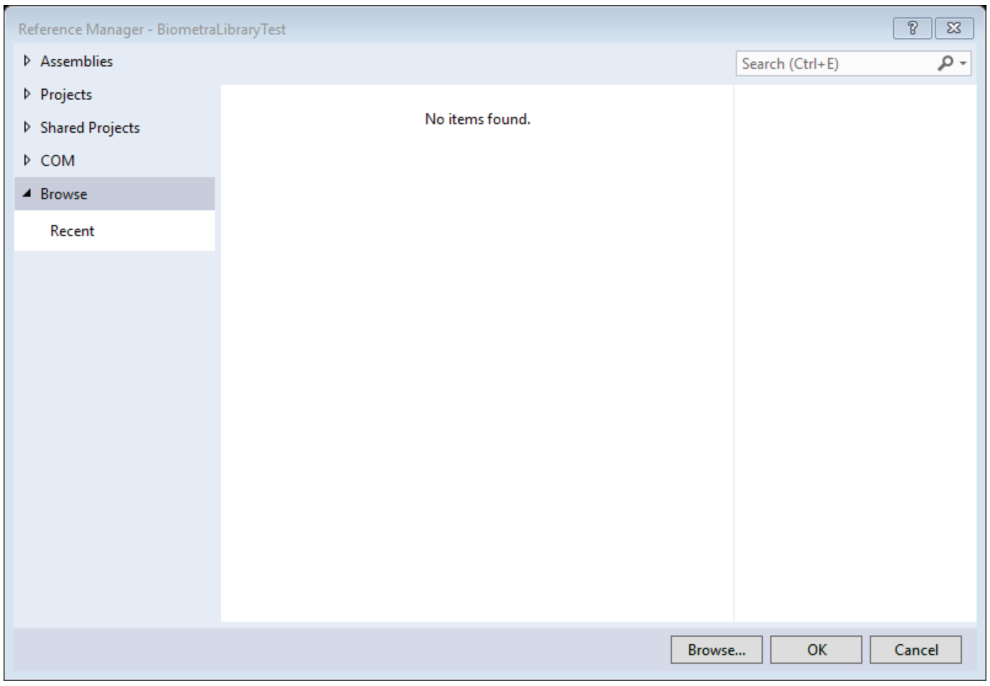
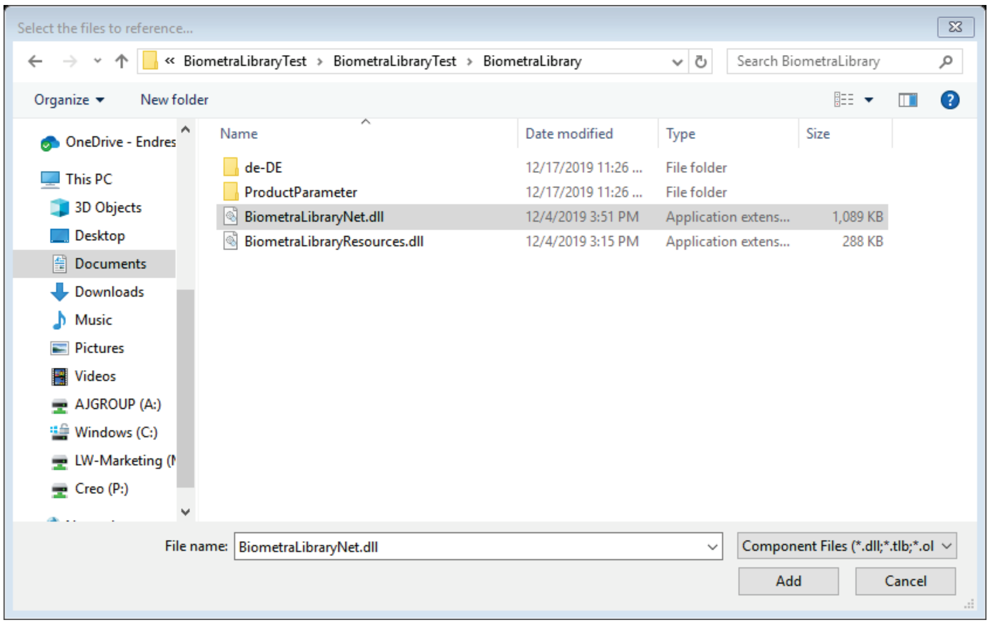


Image 4: The project's Reference Manager

In the dialog box, the file shown in Image 5 must now be selected and added.


Select the files to reference...

« BiometraLibraryTest › BiometraLibraryTest › BiometraLibrary

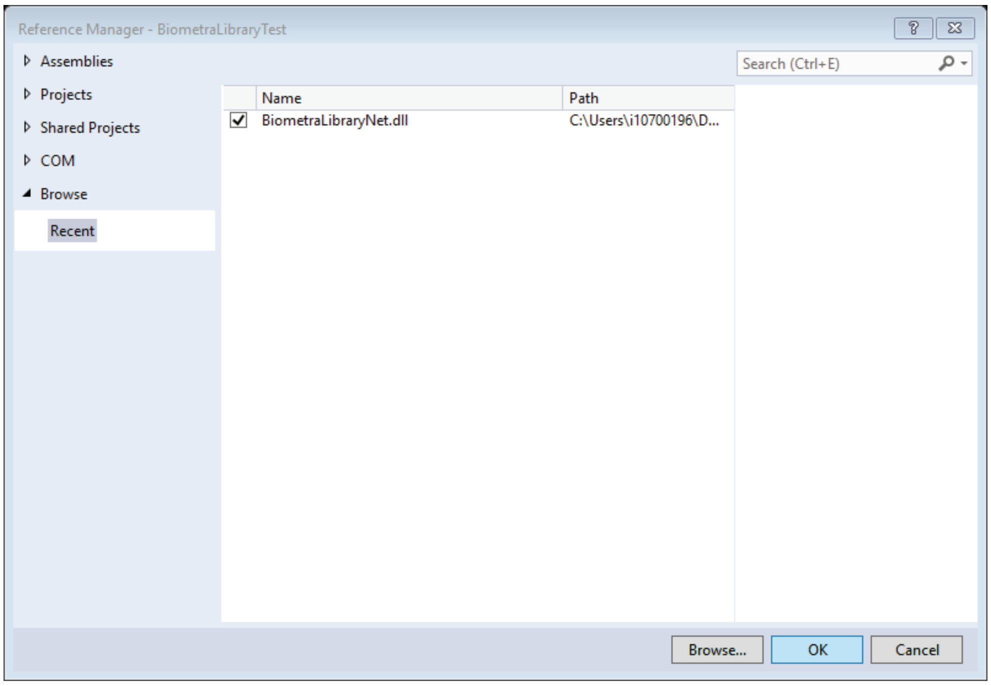
Name	Date modified	Type	Size
de-DE	12/17/2019 11:26 ...	File folder	
ProductParameter	12/17/2019 11:26 ...	File folder	
BiometraLibraryNet.dll	12/4/2019 3:51 PM	Application extens...	1,089 KB
BiometraLibraryResources.dll	12/4/2019 3:15 PM	Application extens...	288 KB

File name: BiometraLibraryNet.dll

Component Files (*.dll;*.tlb;*.ol

Image 5: Reference to the corresponding DLL

Once the reference has been created, the Reference Manager can be closed by press-ing "OK".


Reference Manager - BiometraLibraryTest

	Name	Path
✓	BiometraLibraryNet.dll	C:\Users\i10700196\D...

Image 6: Reference Manager with selected DLL

Now all public classes of the Biometra Library (see library documentation) can be used in the project.

4.5 Basic function of the DLL

The thermocyclers to be controlled can be connected to the end device on the one hand via the local network and on the other hand via serial interfaces, and controlled by the DLL. A USB<->RS232 adapter can also be used for control via a serial interface.

A UDP broadcast is used to find the thermocyclers via the local network. To be used correctly, it is necessary that broadcast queries are allowed in the local network and that the ports on which communication takes place are not blocked.

A maximum of 1000 devices can be managed by the DLL. Found devices are saved locally in a list. A device is not removed from the list of available devices until it has not responded to five consecutive broadcast queries or the list has been manually deleted.

Assignment to the IP address or to the serial interface (COM port) of the device is done according to the device description (object: DeviceDescription). This object can be used to call every communication function in order to address a specific device.

Within the DLL, an object of type CheckStateResult is used as the return value for many methods. The respective method was executed successfully if this return type returns NULL. If an existing CheckStateResult object is returned, an error occurred during processing. The object of the type CheckStateResult is a recursive object, providing information about the origin of the error. An error value with the associated description and the position where the error occurred is always returned.

The communication functions for controlling the devices are located within `Biometra Library.DeviceExtComClasses`. Namespaces there are divided into different categories, such as the `Biometra Library.DeviceExtComClasses.ProgClasses` `ProgramCmds` class. The classes for using the files of a device are located in the namespace `Biometra Library.FileClasses.FileWorkClasses.DeviceFileWorkClasses` and are divided there accordingly.

5 Training/Tutorial

The following sections explain how to configure the Biometra Library for use and how to call the main functions. How to use the Biometra Library is described using the WPF app (.NET Framework), which was already used in section 4.4. This procedure is only a recommendation. Deviating from this, other methods are also available in the different classes that can be used. For further details on usage, please refer to the library documentation.

5.1 Configuration

In order to use the Biometra Library, first the application settings must be initialized. The application settings are a configuration file that is saved by the Biometra Library in the user's "AppData" directory. To initialize the settings with default values, the following method of the static class ApplicationSettings must be called.

```
/*Load application settings. If no settings file exists yet, the file is created
and saved*/
ApplicationSettings.LoadApplicationSettings();
```

Once all desired settings have been changed within the ApplicationSettings, the following method of the static class ApplicationSettings must be called. If the settings have not been saved, the previously saved data will be loaded the next time the application is started.

```
//Save application settings
ApplicationSettings.SaveApplicationSettings();
```

5.1.1 Setting the name of the remote device

The DeviceName property of the CommunicationSettings property of the ApplicationSettings can be used to set the name of a remote device, which is assigned to the thermocycler for a logged-in user.

```
/*Set the name of the remote device from which the communication takes place.*/
ApplicationSettings.CommunicationSettings.DeviceName = new ExternalDevice-
Name("BiometraLibTest");
```

5.1.2 Defining network settings

To establish communication via a local network, it is first of all necessary to read all network adapters that are available on the end device. To do this, the following method of the static class NetworkHelperClass must be called.

```
//Read all available network adapters
AdvancedList<String> networkDeviceList = NetworkHelp-
erClass.GetAllNetworkDeviceDescriptions();
```

To configure the network settings, the first available network adapter in the list is used and passed as the first parameter when the UdpParams object is created. The second parameter is the network port on which the thermocyclers connected to the local network are listening. By default the thermocyclers use port 55555. The third parameter specifies the local broadcast port, which is used to send broadcast queries from the end device and to wait for responses from the devices.

In this example the port is initialized to 0. This means that for each broadcast search that takes place, a free port is dynamically selected on which the broadcast is to be sent. Alternatively, a static port can be defined here. The communication timeout to be used by default is set as the last parameter. This value specifies how long to wait until the reception process is completed during a device search or during direct communication to a device. A timeout value of 1500 ms is recommended.

```
//Set UDP parameters
ApplicationSettings.CommunicationSettings.NetSettings.UdpComSettings.UdpComParams =
new UdpParams(networkDeviceList[0], new NetworkPort(55555), new NetworkPort(),
EnCommunicationTimeout.TIMEOUT_1500ms);
```

It is possible to switch off the scan via broadcast:

```
// Disable UDP broadcast
ApplicationSettings.CommunicationSettings.NetSettings.UdpComSettings.UseBroadcast =
false;
```

IP addresses can be entered in the StaticIPList that are always scanned regardless of the broadcast:

```
// If broadcast is disabled, only these IP-addresses will be scanned for devices
ApplicationSettings.CommunicationSettings.NetSettings.UdpComSettings.StaticIPList =
new AdvancedList<string>() { "192.168.1.100", "192.168.1.103" };
```

Once the communication parameters have been defined, the network communication must be enabled by setting the UdpComSettings property to true. Network communication can be disabled at any time by setting the property to false. The configured communication parameters are still saved.

```
//Activate UDP communication
ApplicationSet-
tings.CommunicationSettings.NetSettings.UdpComSettings.EnableCommunication = true;
```

5.1.3 Defining the serial interface settings

To configure the serial interface settings, the corresponding property of the application settings must be configured as follows. A timeout value of 1500 ms is recommended.

```
Set settings for serial communication
ApplicationSettings.CommunicationSettings.SerialComSettings.SerialComParams = new
SerialParams(EnBaudRate.BAUDRATE_115200, EnParity.PARITY_NONE, EnData-
Bits.DATABITS_8, EnStopBits.STOPBITS_ONE, EnCommunicationTimeout.TIMEOUT_1500ms);
```

Optionally, it is possible to restrict the COM ports to be used for communication:

```
// Disable scanning all serial ports
ApplicationSettings.CommunicationSettings.SerialComSettings.AllComPorts = false;

// List of serial ports that should still be scanned for devices – in this example
COM1, COM2 and COM5
ApplicationSettings.CommunicationSettings.SerialComSettings
      .COMPortFilterListe = new AdvancedList<string>() { "COM1", "COM2", "COM5" };
```

Once the communication parameters have been defined, the serial communication must be enabled by setting the EnableCommunication property to true. Serial communication can be disabled at any time by setting the property to false. The configured communication parameters are still saved.

```
//Enable serial communication
ApplicationSettings.CommunicationSettings.SerialComSettings.EnableCommunication =
true;
```

5.1.4 Setting the default storage location

Within the application settings it is also possible to define a default location where the files called by a thermocycler are stored by default. In the example below, this is assigned as the folder path of the desktop of the user currently logged on. By default, the DefaultFolderPath is set to store the files in the user's My Documents.

```
//Set default location on desktop
ApplicationSettings.DeviceFileSettings.DefaultFolderPath = Environ-
ment.GetFolderPath(Environment.SpecialFolder.Desktop);
```

If the files should also be stored in a subfolder with the name of the running application, the SubFolderName property must be set.

```
//Set name of application as subfolder
ApplicationSettings.DeviceFileSettings.SubFolderName = "Biometra LibraryTest";
```

5.2 Finding devices

There are various methods available within the InfoCmds class to search for devices connected in the network and to the end device. This example only describes the methods that access the application settings and use the communication parameters defined there.

5.2.1 Finding all available devices

The following method is used to determine all available devices using the communication parameters of the application settings.

```
//Search all available devices
InfoCmds.GetAllComAvailableDevices(out DataSetList<AvailableDevice, NotAvailable-
Device> foundDevicesList);
```

Since the method shown is a synchronous method, the calling application is blocked until the device search is completed. The identical asynchronous method can be used as an alternative. The found devices are returned in the associated event. If no further device search is performed, the event should be removed after the search is complete so that the event is not triggered at undesired points, since this is a static event.

```
//Instance event, in which the found devices are returned
InfoCmds.GetAllComAvailableDevices_Finished += In-
foCmds_GetAllComAvailableDevices_Finished;
//Search for all available devices in a background processing
InfoCmds.GetAllComAvailableDevicesAsync();

private void InfoCmds_GetAllComAvailableDevices_Finished(object sender, DeviceCom-
EventArgs<DataSetList<AvailableDevice, NotAvailableDevice>> e)
{
        bool bProcessOkay = false;
        //Execute, if the processing was successful
        if(e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set flag
                bProcessOkay = true;
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());

        /*Remove event, so that, for example, during a page change within an
        application the event is not called at places where it is not
        is required*/
        InfoCmds.GetAllComAvailableDevicesCyclic_Finished -= In-
        foCmds_GetAllComAvailableDevices_Finished;
}
```

If the device search is to be carried out cyclically at a specified time interval, the following method can be used. In this case it is important that background processing is first stopped to terminate the device search. Furthermore, the event should be removed after the search is complete so that the event is not triggered at undesired points, since this is a static event.

```
//Instance event, in which the found devices are returned
InfoCmds.GetAllComAvailableDevicesCyclic_Finished += In-
foCmds_GetAllComAvailableDevicesCyclic_Finished;
//Search for all available devices in a background processing
InfoCmds.GetAllComAvailableDevicesCyclicAsync();

private void InfoCmds_GetAllComAvailableDevicesCyclic_Finished(object sender, De-
viceComEventArgs<DataSetList<AvailableDevice, NotAvailableDevice>> e)
{
        bool bProcessOkay = false;
        //Execute, if the processing was successful
        if(e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set flag
                bProcessOkay = true;
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());

        /*Remove event, so that, for example, during a page change within an
        application the event is not called at places where it is not required*/
        InfoCmds.GetAllComAvailableDevicesCyclic_Finished -= In-
        foCmds_GetAllComAvailableDevicesCyclic_Finished;
}
```

Various static methods are available within the DeviceCom class and all classes derived from it to manage the devices found. Some examples of the most important methods are explained below.

```
//Read descriptions from all available devices
AdvancedList<DeviceDescription> deviceList = De-
viceCom.GetSavedDeviceDescriptions();
```

As described in section 4.5, the DeviceDescription object is required to directly address the devices found. If the DeviceDescription cannot be used directly from the Available-Device record returned by the device search, the example shown above may be helpful to get the device description.

The following example can be used to read the basic device information that is temporarily stored as part of the device search.

```
//Read information about the first device from the list
CheckStateResult checkStateResult = De-
viceCom.GetInformationsByDeviceDescription(deviceList[0], out DeviceInformations
deviceInformations);
```

The following example returns the number of devices found.

```
// Read out number of found devices
int iNumOfDevices = DeviceCom.GetNumberOfScannedDevices();
```

The following method is used to delete the list of found devices.

```
// Delete all found devices
DeviceCom.RemoveAllSavedDevices();
```

A single device can be removed using the following method.

```
//Delete selected device
bool bResult = DeviceCom.RemoveDeviceFromDeviceList(deviceList[0]);
```

Furthermore, various static events are available within the DeviceCom class and all classes derived from it, which can be instantiated if required. Details of the events and other available methods can be found in the library documentation.

5.3 User management

Since most of the device functions require a corresponding user on the thermocycler and this user must be logged in, a description of user management is given in the sections below. This includes the procedure for logging a user on and off on the device. They also explain how a user is created, edited or deleted. In addition, they show how various information about users can be retrieved. Other methods that can be used are contained in the LoginOutCmds and UserCmds classes. The description can be found in the library documentation of the Biometra Library.

5.3.1 Reading available users

Before a user can be logged on to a device, the list of all users must first be retrieved. The first entry from the list of devices from section 5.2.1 is used as the device. The following method of the UserCmds class can be used to retrieve the list.

```
try
{
        //Create object for communication
        using (UserCmds userCmds = new Us-
        erCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Retrieve all available users
                CheckStateResult checkStateResult = us-
                erCmds.GetUserList(deviceList[0], out AdvancedList<UserDataset> us-
                erList);
        }
/*Catch exception, because an exception is triggered when the device has returned
an error message*/
} catch(Exception ex) { MessageBox.Show(ex.Message); }
```

This method is a synchronous method. If the process is to be run in the background, the following method can be used.

```
//Create object for communication
UserCmds userCmds = new UserCmds(ApplicationSettings.CommunicationSettings, device-
List[0]);
//Instance event in which the list is returned
userCmds.GetUserList_Finished += UserCmds_GetUserList_Finished;
//Start the process to retrieve the available users
userCmds.GetUserListAsync(deviceList[0]);

//Event in which the available users are returned
private void UserCmds_GetUserList_Finished(object sender, DeviceComEven-
tArgs<AdvancedList<UserDataset>> e)
```

```
{
        bool bProcessOkay = false;
        //Execute, if the processing was successful
        if(e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set Flag
                bProcessOkay = true;
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());
}
```

5.3.2 Logging a user on and off

After retrieving the users available on the device, a user can be selected from the corresponding list and logged on to the device. The "Admin" user with the initials "ADM" and the default password "Admin" is available on each device. The "Admin" user is used for the following examples. The following method can be used to log on the user.

```
try
{
        //Create object for communication
        using (LoginOutCmds loginOutCmds = new
        LoginOutCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Register user ADM
                CheckStateResult checkStateResult =
                loginOutCmds.LoginUser(deviceList[0], new UserInitials("ADM"), new
                UserPassword("Admin"));
        }
         /*Catch exception, because an exception is triggered when the device has
        returned an error message.
         In this case, an exception can be made if a user is already logged in*/
}catch(Exception ex) { MessageBox.Show(ex.Message); }
```

The following method is used to log a user off the device.

```
try
{
        //Create object for communication
        using (LoginOutCmds loginOutCmds = new
        LoginOutCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Log off user
                CheckStateResult checkStateResult =
                loginOutCmds.LogoutUser(deviceList[0]);
        }
/*Catch exception, because an exception is triggered when the device has returned
an error message.
In this case an exception can be made if no user is logged on*/
}catch(Exception ex) { MessageBox.Show(ex.Message); }
```

Various static methods are available within the LoginOutCmds class and all classes derived from it to manage the logged-on users. Some examples of the most important methods are explained below.

```
//Read the number of users logged in
int iNumOfUser = LoginOutCmds.GetNumOfLoggedInUsers();
```

Using the example shown above, it is possible to read the number of users logged on. The following example shows how to check whether a user is logged on to the device passed as a parameter.

```
/*Check whether a user has already been logged in via the Biometra Library on the
selected device*/
CheckStateResult checkStateResult =
LoginOutCmds.CheckIfUserLoggedIn(deviceList[0]);
```

The following example can be used to read the record of the logged on user for further use (e.g. checking whether permissions exist).

```
//Read data record of the logged in user
CheckStateResult checkStateResult = LoginOutCmds.GetUserDataSetByDeviceDescription
(deviceList[0], out UserDataset userDataSet);
```

Furthermore, various static events are available within the LoginOutCmds class and all classes derived from it, which can be instantiated if required. Details of the events and other available methods can be found in the library documentation.

5.3.3 Creating a user

The following example shows how to create a user on a device. Since individual permissions are adjusted in the example, a user from the administrator group must first be logged on and user management must be enabled on the device. The process for creating a user may differ slightly depending on the permissions of the logged-on user.

```
//Create new user data set
UserDataset userDataset = new UserDataset
{
        //Set the initials of the user (required parameter)
        Initials = new UserInitials("TES"),
        //Specify the user's name (required parameter);
        Name = new UserName("TestUser"),
        //Set the language of the user (optional, default is English)
        Language = EnLanguage.LANGUAGE_GERMAN,
        //Set the user's password (optional)
        Password = new UserPassword("TestPass"),
        //Define the user's authorizations (based on the authorization group of a
        restricted user)
        Authorizations = new UserAuthoriza-
        tions(EnUserAuthorizationTemplates.USERTEMPLATE_LIMITED),
};
//As the user is not allowed to read other programs, the individual authorization
is set to false
userDataset.Authorizations.ReadOtherProgs = false;
try
{
        //Create communication object
        using (UserCmds userCmds = new Us-
        erCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
        //Create user
        CheckStateResult checkStateResult = userCmds.CreateUser(deviceList[0],
        userDataset);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*

catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.3.4 Editing a user

The following example shows how a user existing on the device can be edited. To do this, the user created in section 5.3.3 is retrieved from the device and edited. The process for editing a user may differ slightly depending on the permissions of the logged-on user.

```
try
{
        //Create communication object
        using (UserCmds userCmds = new Us-
        erCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Retrieve data record of the user "TES"
                CheckStateResult checkStateResult = us-
                erCmds.GetUserDataset(deviceList[0], new UserInitials("TES"), out
                UserDataset userDataset);
                //Execute, if the record could be retrieved
                if (checkStateResult == null)
                {
                        //Switch user language to English
                        userDataset.Language = EnLanguage.LANGUAGE_ENGLISH;
                        //Change name of the user
                        userDataset.Name = new UserName("Test");
                        //Disable password of the user
                        userDataset.Password = new UserPass-
                        word(EnUserPasswordState.USERPASSWORD_DISABLED);
                        //Change the user's authorization group to a general user
                        userDataset.Authorizations = new UserAuthoriza-
                        tions(EnUserAuthorizationTemplates.USERTEMPLATE_GENERAL);
                        /*Edit user. The editing process is carried out using the
                        initials contained in the data record.
                        The initials of a user cannot be changed. If the initials
                        in the data set are changed and a corresponding user exists
                        on the associated instrument, this user is edited*/
                        checkStateResult = userCmds.EditUser(deviceList[0],
                        userDataset);
                }
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.3.5 Delete user

The following example can be used to delete a user. To do this, the user created in section 5.3.3 is deleted from the device.

```
try
{
        //Create communication object
        using (UserCmds userCmds = new Us-
        erCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Delete user "TES"
                CheckStateResult checkStateResult = us-
                erCmds.DeleteUser(deviceList[0], new UserInitials("TES"));
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.4 Device configuration

The following sections describe how to retrieve and configure the various device settings. They also describe how to reset the device to factory settings, how to retrieve and save a service information file (SINF file), and how to create and restore a backup. Other methods that can be used are contained in the SettingsCmds class. The description can be found in the library documentation of the Biometra Library.

5.4.1 Reading settings

The following example can be used to retrieve the current settings from the device. The settings are retrieved dynamically depending on the device type. With the TRobot

II the parameters for the motorized lid are retrieved in addition to the normal device settings.

```
try
{
        //Create communication object
        using (SettingsCmds settingsCmds = new Setting-
        sCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                ///Recall current settings from the device
                CheckStateResult checkStateResult = setting-
                sCmds.GetCurrentDeviceSettings(deviceList[0], out DeviceSettings
                deviceSettings);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.4.2 Configuring settings

To configure the settings, it is a good idea to first retrieve the current device settings as described in section 5.4.1. Once the settings have been retrieved, they can be edited and transferred back to the device. In this example all settings are changed. However, only those settings that are required need to be amended.

```
try
{
        //Create communication object
        using (SettingsCmds settingsCmds = new Setting-
        sCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Recall current settings from the device
                CheckStateResult checkStateResult = setting-
                sCmds.GetCurrentDeviceSettings(deviceList[0], out DeviceSettings
                deviceSettings);
                //Execute, if the settings have been successfully recalled
                if (checkStateResult == null)
                {
                        //Activate automatic user logoff after 45 minutes
                        deviceSettings.AutoLogoutSettings = new AutoLogoutModeSet-
                        tings(EnAutoLogoutState.AUTOLOGOUT_ENABLED, new AutoL-
                        ogoutTime(45));
                        //Activate beeper
                        deviceSettings.BeeperState = EnBeeperState.BEEPER_ENABLED;
                        //Define device location
                        deviceSettings.DevLocation = new DeviceLocation("TestLab");
                        //Change device name
                        deviceSettings.DevName = new DeviceName("TestDevice");
                        /*Set the opening mode of the lid so that the inserted
                        plate is automatically ejected*/
                        deviceSettings.LidOpenMode = EnMot-
                        LidOpenMode.OPENMODE_EJECT;
                        //Attempt to generate a contact pressure of 10 kg
                        CheckStateResult innerResult = MotLidPres-
                        sure.TryCreate(10.0, out MotLidPressure motLidPressure);
                        //Execute, when the contact pressure has been generated
                        if (innerResult == null)
                        {
                                //Set the contact pressure to 10 kg
                                deviceSettings.LidPressure = motLidPressure;
                        }
                        //Adjust network port. Attention: If the network port is
                        changed, the application software must also be set to the
                        corresponding device port
                        deviceSettings.NetPort = new NetworkPort(55556);
                        /*Set IP address statically. You must ensure that the ter-
                        minal device from which the IP address is changed is also
                        in the same address range, otherwise
                        the device can no longer be found. By default, this setting
                        is DHCP*/
                        deviceSettings.NetworkConfig = new NetworkConfigura-
                        tion(IPAddress.Parse("192.168.0.10"), IPAd-
                        dress.Parse("255.255.255.0"));
                        //Activate user management
                        deviceSettings.UserManagementState = EnUserManagement-
                        State.USERMANAGEMENT_ENABLED;
                        //Read current date/current time from PC and create Device-
                        DateTime object
                        innerResult = DeviceDateTime.TryCreate(DateTime.Now, out
                        DeviceDateTime deviceDateTime);
                        Execute if the object could be created
                        if (innerResult == null)
                        {
                                //Assign current date / current time
                                deviceSettings.CurDateTime = deviceDateTime;
                        }
                        //Create reference to the device
                        DeviceDescription deviceDescription = deviceList[0];
                        //Transfer settings to the device. The DeviceDescription is
                        passed as ref parameter, because if the device name is
                        changed, the passed object will also be adjusted
                        checkStateResult = settingsCmds.SetDeviceSettings(ref de-
                        viceDescription, deviceSettings);
                }
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.4.3 Resetting to factory settings

The following section describes how to reset the device to factory settings.

```
try
{
        //Create communication object
        using (SettingsCmds settingsCmds = new Setting-
        sCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Set device back to factory settings
                CheckStateResult checkStateResult = setting-
                sCmds.SetFactoryReset(deviceList[0]);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

Since resetting to factory settings is a lengthy process and the example above blocks further processing until the process is complete, the following asynchronous method can be used as an alternative.

```
//Create communication object
SettingsCmds settingsCmds = new Setting-
sCmds(ApplicationSettings.CommunicationSettings, deviceList[0]);
//Instance event in which the result of the operation is reported
settingsCmds.SetFactoryReset_Finished += SettingsCmds_SetFactoryReset_Finished;
//Start the procedure for resetting to factory settings
settingsCmds.SetFactoryResetAsync(deviceList[0]);

private void SettingsCmds_SetFactoryReset_Finished(object sender, DeviceComEven-
tArgs<object> e)
{
        bool bProcessOkay = false;
        //Execute, if the processing was successful
        if(e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set Flag
                bProcessOkay = true;
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());
}
```

5.4.4 Retrieving and saving a service info file

The service info file of the device is required to transmit any errors to the Analytik Jena/Biometra service. The following example can be used to retrieve the file from the device.

```
try
{
        //Create communication object
        using (SinfFileWorker sinfFileWorker = new SinfFileWork-
        er(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                /* Retrieve SINF file from the device and create it under the de-
                fault path named "TestSinf". If the parameter "TestSinf" is not
                transferred or is empty, a file name is automatically generated
                which corresponds to the format in which the files are also saved
                from the thermal cycler to the USB stick*/
                CheckStateResult checkStateResult = sinfFileWork-
                er.LoadAndSaveSinfFile(deviceList[0], ApplicationSet-
                tings.DeviceFileSettings.GetServiceInfoFolderPath(deviceList[0]),
                "TestSinf");
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

Since retrieving and saving the service info file is a lengthy process and the example above blocks further processing until the process is complete, the following asynchronous method can be used as an alternative.

```
//Create communication object
SinfFileWorker sinfFileWorker = new SinfFileWork-
er(ApplicationSettings.CommunicationSettings, deviceList[0]);
//Instantiate event
sinfFileWorker.LoadAndSaveSinf_Finished += SinfFileWorker_LoadAndSaveSinf_Finished;
/*Retrieve SINF file from the device and create it under the default path named
"TestSinf". If the "TestSinf" parameter is not passed, a file name is automatically
generated which corresponds to the format in which the files are saved from the
thermal cycler to the USB stick*/
sinfFileWorker.LoadAndSaveSinfFileAsync(deviceList[0], ApplicationSet-
tings.DeviceFileSettings.GetServiceInfoFolderPath(deviceList[0]), "TestSinf");

private void SinfFileWorker_LoadAndSaveSinf_Finished(object sender, DeviceFile-
WorkEventArgs<object> e)
{
        bool bProcessOkay = false;
        //Execute, if the processing was successful
        If (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set flag
                bProcessOkay = true;
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());
}
```

5.4.5 Creating and saving a backup

The following example can be used to create and save a backup of a device.

```
try
{
        //Create communication object
        using (ImageFileWorker.imageFileWorker = new ImageFileWork-
        er(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                /*Create a backup of the device and save it under the default path
                with the name "TestBackup". If the "TestBackup" parameter is not
                transferred or is empty, a file name is automatically generated
                which corresponds to the format in which the files are also saved
                from the thermal cycler to the USB stick*/
                CheckStateResult checkStateResult = imageFileWork-
                er.LoadImgFromDeviceAndSave(deviceList[0], ApplicationSet-
                tings.DeviceFileSettings.GetImageFolderPath(deviceList[0]),
                "TestBackup");
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

Since creating and saving a backup is a lengthy process and the example above blocks further processing until the process is complete, the following asynchronous method can be used as an alternative.

```
//Create communication object
ImageFileWorker imageFileWorker = new ImageFileWork-
er(ApplicationSettings.CommunicationSettings, deviceList[0]);
//Instantiate event
imageFileWorker.LoadImgFromDeviceAndSave_Finished += LoadImgFromDevice-
AndSave_Finished;
/*Create a backup of the device and save it under the default path with the name
"TestBackup". If the "TestBackup" parameter is not transferred or is empty, a file
name is automatically generated which corresponds to the format in which the files
are also saved from the thermal cycler to the USB stick*/
imageFileWorker.LoadAndSaveSinfFileAsync(deviceList[0],           ApplicationSet-
tings.DeviceFileSettings.GetImageFolderPath(deviceList[0]), "TestBackup");

private void ImageFileWorker_LoadImgFromDeviceAndSave_Finished(object sender, De-
viceFileWorkEventArgs<object> e)
{
        bool bProcessOkay = false;
        //Execute, if the processing was successful
        If (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set flag
                bProcessOkay = true;
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());
}
```

5.4.6 Restoring a backup to the device

The following example can be used to restore a retrieved and saved backup to a device. In this example, the backup retrieved and saved in section 5.4.5 is transferred back to the device.

```
try
{
        //Create communication object
        using (ImageFileWorker.imageFileWorker = new ImageFileWork-
        er(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                /*Read backup with the name TestBackup from the default path and
                transfer it to the device.*/
                CheckStateResult checkStateResult = imageFileWork-
                er.ReadSavedImageAndTransferToDevice(deviceList[0],
                Path.Combine(ApplicationSettings.DeviceFileSettings.GetImageFolderP
                ath(deviceList[0]), "TestBackup.TXT"));
        }
}
/*Catch an exception, because an exception is triggered when the device has re-
turned an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

Since restoring a backup is a lengthy process and the example above blocks further processing until the process is complete, the following asynchronous method can be used as an alternative.

```
//Create communication object
ImageFileWorker imageFileWorker = new ImageFileWork-
er(ApplicationSettings.CommunicationSettings, deviceList[0]);
//Instance event
imageFileWorker.ReadSavedImgAndTransferToDevice_Finished += ImageFileWork-
er_ReadSavedImgAndTransferToDevice_Finished;
/*Read backup with the name TestBackup from the default path and transfer it to the
device.*/
imageFileWorker.ReadSavedImageAndTransferToDeviceAsync(deviceList[0],
Path.Combine(ApplicationSettings.DeviceFileSettings.GetImageFolderPath(deviceList[0
]), "TestBackup.TXT"));

private void ImageFileWorker_ReadSavedImgAndTransferToDevice_Finished(object send-
er, DeviceFileWorkEventArgs<object> e)
{
        bool bProcessOkay = false;
        //Execute, if processing was successful
        If (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set flag
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());
}
```

5.5 Program management

The following sections describe how to create PCR programs that can be run on the device. They also describe how these programs are managed and executed. In addition to the basic methods, there are other methods in the classes PcrProgram, ProgramCmds, ProgramFileWorker and BlockCmds that can be used. The description can be found in the library documentation of the Biometra Library.

5.5.1 Retrieving the program overview from the device

To start a program on a device for example, it is first necessary to read all available programs of the user who wants to start the program. The following example can be used for this purpose.

```
try
{
        //Create communication object
        using (ProgramCmds programCmds = new Pro-
        gramCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Read all available programs of the user "ADM"
                CheckStateResult checkStateResult = pro-
                gramCmds.GetProgramOverview(deviceList[0],new UserInitials("ADM"),
                out DataSetList<ProgramInfos> programList);
        }
}
/* Catch exception, because an exception is triggered when the device has returned
an error message.*/
```

```
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

Since the process can take a long time depending on the number of available programs, the asynchronous method can also be used as an alternative. The procedure is shown in the following example.

```
//Create communication object
ProgramCmds programCmds = new Pro-
gramCmds(ApplicationSettings.CommunicationSettings, deviceList[0]);
//Instantiate return event
programCmds.GetProgramOverview_Finished += ProgramCmds_GetProgramOverview_Finished;
//Retrieve program list in the background
programCmds.GetProgramOverviewAsync(deviceList[0], new UserInitials("ADM"));

private void ProgramCmds_GetProgramOverview_Finished(object sender, DeviceComEven-
tArgs<DataSetList<ProgramInfos>> e)
{
        bool bProcessOkay = false;
        //Execute, if processing was successful
        if (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set flag
                bProcessOkay = true;
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());
}
```

5.5.2 Creating a program

The class PcrProgram must be used to create a program. This class contains all necessary properties and methods of a PCR program. The following example shows how to create a program. This program is a temperature program with one step.

```
//Create an object to read all available templates, on the basis of which a program
is to be created
ProgramFileWorker programFileWorker = new ProgramFileWorker();
//Load all available templates
CheckStateResult checkStateResult = programFileWork-
er.ReadAllProgramTemplateInfosToShow(out DataSetList<ProgramInfosToShow, String>
progTemplateList);
if(checkStateResult != null)
{
        return;
}
//Create new program object
PcrProgram pcrProgram = new PcrProgram();
//Load program template for the block type of the TRobot. In this case an empty
program template was selected
checkStateResult = pcrPro-
gram.ImportProgramTemplate(progTemplateList.DataList[0].CompleteFilePath, new
BlockType(EnBlockTypeNumber.BLOCK_400_GRADIENT_96));
if(checkStateResult != null)
{
        return;
}
//Set name of the program
pcrProgram.ProgramEditInfo.ProgName = new ProgramName("TestProg");
//Set the user directory of the program
pcrProgram.ProgramEditInfo.UserDirectory = new UserInitials("ADM");
//Set storage date
pcrProgram.ProgramEditInfo.SaveDate = DateTime.Now;
//Activate preheating of heating lid and set the temperature to 98 °C
pcrProgram.ProgramHeadInfo.LidSetting = new LidSettings(new LidTemperature("98 °C",
pcrProgram.ProgramHeadInfo.BlockTyp.BlockTypData), true);
//Read next free program number
checkStateResult = pcrProgram.GetNextProgStepNumber(out ProgStepNumber
nextProgStepNumber);
if (checkStateResult != null)
```

```
{
        return;
}
//Create new program step
ProgramStep programStep = new Program-
Step(pcrProgram.ProgramHeadInfo.BlockTyp.BlockTypData, nextProgStepNumber);
//Set step temperature to 95 °C
programStep.SetProgStepTemperature(new ProgStepTemperature("95,0 °C"));
//Set the holding time of the program step to 2 hours, 5 minutes and 10 seconds
checkStateResult = programStep.SetHoldTime(new ProgStepHoldTime(new TimeSpan(2, 5,
10)));
if (checkStateResult != null)
{
        return;
}
//Add program step to the list of program steps
checkStateResult = pcrProgram.AddProgramStep(programStep);
if (checkStateResult != null)
{
        return;
}
```

Once the program has been created, it can be saved locally, transferred directly to the device or exported. Saving the program is simply a matter of calling the Save method of the PcrProgram class.

```
//Save the program in the standard directory with the name TestProg
checkStateResult = pcrPro-
gram.Save(ApplicationSettings.DeviceFileSettings.GetProgramFolderPath(deviceList[0]
), "TestProg");
```

The following example can be used to export the program.

```
//Export program as CSV file named TestProgramExport to the default directory
CheckStateResult checkStateResult = pcrPro-
gram.Export(ApplicationSettings.DeviceFileSettings.GetProgramFolderPath(deviceList[
0]), "TestProgramExport", EnDeviceFileExportTypes.FILETYPE_CSV);
```

The available export file types can be read as follows.

```
//Read available export types
AdvancedList<EnDeviceFileExportTypes> exportTypes = PcrPro-
gram.SUPPORTED_EXPORT_FILE_TYPES_P;
```

The following example can be used to transfer the created program to the device.

```
try
{
        //Create communication object
        using (ProgramCmds programCmds = new Pro-
        gramCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Send created program to the device.
                CheckStateResult checkStateResult = pro-
                gramCmds.SendProgramToDevice(deviceList[0], pcrProgram);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.5.3 Edit program

This section describes how to read and edit a program stored locally or on the device. The program created in section 5.5.2 is read and edited. The following example shows how the locally stored program is read and edited.

```
//Create new program
PcrProgram pcrProgram = new PcrProgram();
```

```
//Import saved program
checkStateResult = pcrPro-
gram.Import(Path.Combine(ApplicationSettings.DeviceFileSettings.GetProgramFolderPat
h(deviceList[0]), "TestProg.TXT");
/*Read out all contained program steps that can be displayed. An update event is 
instantiated, so that if the step has changed, a dependent graphic can also be 
adjusted*/
AdvancedList<ProgStepToShow> progStepToShow = pcrPro-
gram.GetProgStepToShowList(ProgramStepUpdateEvent);
//Read program step data
ProgramStep programStep = progStepToShow[0].ProgStepDataToShow.Clone();
//Change normal temperature step to a standard gradient
programStep.SetProgStepTemperature(new ProgStepGrad-
LineTemp(pcrProgram.ProgramHeadInfo.BlockTyp.BlockTypData, "82,5 °C"), new 
ProgStepGradLineTemp(pcrProgram.ProgramHeadInfo.BlockTyp.BlockTypData, "87,5 °C"), 
pcrProgram.ProgramHeadInfo.BlockTyp.BlockTypData);
//Update program step
checkStateResult = pcrProgram.UpdateProgramStep(programStep);
//Run when the step has been updated
if(checkStateResult == null)
{
        //Overwrite previous program
        checkStateResult = pcrPro-
        gram.Save(ApplicationSettings.DeviceFileSettings.GetProgramFolderPath(devic
        eList[0]), "TestProg");
}

private void ProgramStepUpdateEvent(object sender, ProgStepUpdateEventArgs e)
{

}/*private void ProgramStepUpdateEvent(object sender, ProgStepUpdateEventArgs e)*/
```

The following example can be used to edit a program stored on the device.

```
try
{
        //Create communication object
        using (ProgramCmds programCmds = new Pro-
        gramCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Retrieve the user's program overview
                CheckStateResult checkStateResult = pro-
                gramCmds.GetProgramOverview(deviceList[0], new UserInitials("ADM"), 
                out DataSetList<ProgramInfos> programList);
                if(checkStateResult != null)
                {
                        return;
                }
                ProgramInfos programInfos = programList.DataList[0];
                //Read program from device
                checkStateResult = programCmds.GetUserProgram(deviceList[0], pro-
                gramInfos, out PcrProgram pcrProgram);
                if(checkStateResult != null)
                {
                        return;
                }
                //Read program step list
                AdvancedList<ProgStepToShow> progStepToShow = pcrPro-
                gram.GetProgStepToShowList(ProgramStepUpdateEvent);
                //Read program step data
                ProgramStep programStep = progStep-
                ToShow[0].ProgStepDataToShow.Clone();
                //Change normal temperature step to a standard gradient
                programStep.SetProgStepTemperature(new ProgStepGrad-
                LineTemp(pcrProgram.ProgramHeadInfo.BlockTyp.BlockTypData, "82,5 
                °C"), new ProgStepGrad-
                LineTemp(pcrProgram.ProgramHeadInfo.BlockTyp.BlockTypData, "87,5 
                °C"), pcrProgram.ProgramHeadInfo.BlockTyp.BlockTypData);
                //Update program step
                checkStateResult = pcrProgram.UpdateProgramStep(programStep);
                //Run when the step has been updated
                if(checkStateResult == null)
                {
                        //Create program information that can be displayed
```

```
                                ProgramInfosToShow.TryCreate(deviceList[0], programInfos,
                                out ProgramInfosToShow programInfosToShow);
                                //Overwrite previous program
                                checkStateResult = programCmds.SaveProgram(deviceList[0],
                                ref pcrProgram, programInfosToShow);
                        }
                }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.5.4 Copying a program

To copy a program, various options are available in the classes ProgramCmds and ProgramFileWorker, some of which are described within this section. For a description of the other options, please refer to the library documentation of the Biometra Library. The following example shows how a program stored on the device can be copied within the same user directory.

```
try
{
        //Create communication object
        using (ProgramCmds programCmds = new Pro-
        gramCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Create initials of the admin user
                UserInitials adminUser = new UserInitials("ADM");
                /*Copy the created program with the number 1 from the directory of
                the ADM to the directory of the ADM with the number 2*/
                CheckStateResult checkStateResult = pro-
                gramCmds.CopyProgram(deviceList[0], adminUser, new ProgramNumber(1,
                EnProgramType.TYPE_PROGRAM), adminUser, new ProgramNumber(2, EnPro-
                gramType.TYPE_PROGRAM));
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

Copying the program to another user directory on the device is simply a matter of changing the "targetDirectory" parameter. The following example shows how a program can be retrieved from the device and stored locally.

```
try
{
        //Create communication object
        using (ProgramFileWorker programFileWorker = new ProgramFileWork-
        er(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                /*Retrieve program number 1 from the ADM directory and store it lo-
                cally in the default path of the device*/
                CheckStateResult checkStateResult = programFileWork-
                er.LoadUserProgFromDeviceAndSave(deviceList[0], ApplicationSet-
                tings.DeviceFileSettings.GetProgramFolderPath(deviceList[0]), new
                UserInitials("ADM"), new ProgramNumber(1, EnPro-
                gramType.TYPE_PROGRAM));
                }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

The following example can be used to transfer a program stored locally to the device.

```
try
{
        //Create communication object
        using (ProgramFileWorker programFileWorker = new ProgramFileWork-
        er(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                /*Read in stored program and transfer it to the device in the di-
                rectory of the ADM*/
                CheckStateResult checkStateResult = programFileWork-
                er.ReadSavedProgAndTransferToDevice(deviceList[0],
                Path.Combine(ApplicationSettings.DeviceFileSettings.GetProgramFolde
                rPath(deviceList[0]), "TestProg.TXT"), new UserInitials("ADM"));
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.5.5 Deleting a program

To delete a program, various options are available in the classes ProgramCmds and ProgramFileWorker, some of which are described within this section. For a description of the other options, please refer to the library documentation of the Biometra Library.

The following example shows how a program stored on the device can be deleted from the user directory.

```
try
{
        //Create communication object
        using (ProgramCmds programCmds = new Pro-
        gramCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Delete program number 1 from the ADM directory
                CheckStateResult checkStateResult = pro-
                gramCmds.DeleteProgram(deviceList[0], new UserInitials("ADM"), new
                ProgramNumber(1, EnProgramType.TYPE_PROGRAM));
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

The following example can be used to delete all programs of a user.

```
try
{
        //Create communication object
        using (ProgramCmds programCmds = new Pro-
        gramCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Delete program number 1 from the ADM directory
                CheckStateResult checkStateResult = pro-
                gramCmds.DeleteAllPrograms(deviceList[0], new UserInitials("ADM"),
                out DataSetList<ProgramInfos> notDeletedPrograms);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

Since the process may take longer depending on the number of programs that exist, it is recommended to use the corresponding asynchronous method.

```
//Create communication object
ProgramCmds programCmds = new Pro-
gramCmds(ApplicationSettings.CommunicationSettings, deviceList[0]);
//Instance event, which is triggered after the deletion process is completed
programCmds.DeleteAllPrograms_Finished += ProgramCmds_DeleteAllPrograms_Finished;
//Start procedure to delete all programs
programCmds.DeleteAllProgramsAsync(deviceList[0], new UserInitials("ADM"));

private void ProgramCmds_DeleteAllPrograms_Finished(object sender, DeviceComEven-
tArgs<DataSetList<ProgramInfos>> e)
{
        bool bProcessOkay = false;
        //Execute, if processing was successful
        if (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set flag
                bProcessOkay = true;
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());
}
```

5.5.6 Creating and starting an incubation program

The IncubationProg class is available for creating an incubation program. The following example shows how to use this class.

```
//Create new incubation program
IncubationProg incubationProg = new IncubationProg();
//Set the temperature of the incubation program to 37 °C
incubationProg.IncubationTemp = new ProgStepTemperature(37.0);
//Set settings of the heating lid
incubationProg.LidTempSettings = new LidSettings(new LidTemperature(new Block-
Type(EnBlockTypeNumber.BLOCK_400_GRADIENT_96).BlockTypData), true);
```

The following example can be used to start the created incubation program on the device.

```
try
{
        //Create communication object
        using (BlockCmds blockCmds = new
        BlockCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Start the created incubation program on block 1. An extended run
                log file is recorded for the incubation program
                CheckStateResult checkStateResult =
                blockCmds.StartProgramOnBlock(deviceList[0], incubationProg, new
                BlockNumber(1), true);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

The example shows starting to log to an advanced run log file. If no advanced run log file is required, the last parameter of the StartProgramOnBlock method must not be passed or the value must be changed to false.

5.5.7 Starting the program

The following example describes how to start a program from a user directory on a block of a device. Within this method, logging to an advanced run log file can be started as an option.

```
try
{
        //Create communication object
        using (BlockCmds blockCmds = new
        BlockCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                /*The recording of an extended run log file is started. If you do
                not want this to happen, you must pass false or prevent the method
                from being overloaded without the parameter for starting the ex-
                tended run log file*/
                CheckStateResult checkStateResult =
                blockCmds.StartProgramOnBlock(deviceList[0], new
                UserInitials("ADM"), new ProgramNumber(1, EnPro-
                gramType.TYPE_PROGRAM), new BlockNumber(1), true);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.5.8 Stop program

The following example can be used to stop a running program on the block of a device.

```
try
{
        //Create communication object
        using (BlockCmds blockCmds = new
        BlockCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Stop currently running program on block 1 of the device
                CheckStateResult checkStateResult =
                blockCmds.StopProgramOnBlock(deviceList[0], new BlockNumber(1));
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.5.9 Pause program

The following example can be used to pause a running program on the block of a device.

```
try
{
        //Create communication object
        using (BlockCmds blockCmds = new
        BlockCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Pause the currently running program on block 1 of the device
                CheckStateResult checkStateResult =
                blockCmds.HoldProgramOnBlock(deviceList[0], new BlockNumber(1));
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.5.10 Continuing the program

The paused program on a block can be continued using the following example.

```
try
{
        //Create communication object
        using (BlockCmds blockCmds = new
        BlockCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Continue currently paused program on block 1 of the device
                CheckStateResult checkStateResult =
                blockCmds.ContProgramOnBlock(deviceList[0], new BlockNumber(1));
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.5.11 Skipping a program step

The following example can be used to skip a program step.

```
try
{
        //Create communication object
        using (BlockCmds blockCmds = new
        BlockCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                /*Skip currently running program step to block 1 of the unit*/
                CheckStateResult checkStateResult =
                blockCmds.SkipProgramOnBlock(deviceList[0], new BlockNumber(1));
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.6 Operating the Biometra TRobot II lid

The following sections describe how to open and close the lid of the Biometra TRobot II.

5.6.1 Opening the lid

The following example shows how to open the lid of the Biometra TRobot II. The command to open the lid can be sent if the lid status has an error, the lid is not yet initialized, or is closed.

```
try
{
        //Create communication object
        using (BlockCmds blockCmds = new
        BlockCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                /*Open the cover of the first block. The opening mode used in the
                opening mode used in the instrument settings is used*/
                CheckStateResult checkStateResult =
                blockCmds.OpenMotLid(deviceList[0], new BlockNumber(1));
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.6.2 Closing the lid

The following example shows how to close the lid of the Biometra TRobot II. The command to close the lid can only be sent if the lid status is open.

```
try
{
        //Create communication object
        using (BlockCmds blockCmds = new
        BlockCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                /*Close the cover of the first block. The contact pressure set in
                the device settings is used*/
                CheckStateResult checkStateResult =
                blockCmds.CloseMotLid(deviceList[0], new BlockNumber(1));
        }
}
```

```
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.7 Live data from the device

The following sections describe the options for retrieving and managing live data from the device. The live data of the device includes the current block parameters, the advanced run log file, and the block monitoring. Other data can be retrieved from the device using the classes InfoCmds and TcdaCmds.

5.7.1 Current block parameters

Using the current block parameters, it is possible to control the devices depending on the respective status values. The following example shows how the current block parameters are retrieved from the device.

```
try
{
        //Create communication object
        using (TcdaCmds tcdaCmds = new TcdaC-
        mds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Query block parameters for all existing blocks
                CheckStateResult checkStateResult = tcdaC-
                mds.GetBlockParamsDataSet(deviceList[0], out DataSet-
                List<BlockParams, BlockNumber> blockParamList);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

It is also possible to query the block parameters in a background process (also cyclically). The corresponding example is shown below.

```
//Create communication object
TcdaCmds tcdaCmds = new TcdaCmds(ApplicationSettings.CommunicationSettings, device-
List[0]);
//Instance event
tcdaCmds.GetAllBlockParamsDataSet_Finished += TcdaC-
mds_GetAllBlockParamsDataSet_Finished;
/*Call up block parameters cyclically. If the block parameters are to be retrieved
once, no interval must be transferred.*/
tcdaCmds.GetBlockParamsDataSetAsync(deviceList[0], TcdaC-
mds.BLOCK_PARAMS_SCAN_INTERVAL_P);

private void TcdaCmds_GetAllBlockParamsDataSet_Finished(object sender, DeviceComEv-
entArgs<DataSetList<BlockParams, BlockNumber>> e)
{
        if (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Put out all block parameters
                foreach(BlockParams blockParams in
                e.Result.ProcessResultData.DataList)
                {
                        MessageBox.Show(blockParams.ToString());
                }
        }
}
```

When using cyclical background processing, it must be ensured that the process is also terminated. This requires the following call.

```
//Cancel background processing
tcdaCmds.CancelTransferProcess();
```

5.7.2 Extended run log file

As described in section 5.5.6 and 5.5.7, when starting a program, it is possible to specify that an advanced run log file should be written for the program run. The following examples describe how the current logging can be managed. The following example can be used to save all completed run log files after logging is complete.

```
//Instance event, that is triggered after the recording of extended run log files
is completed
AdvancedRunLogFileRecording.AdvancedRunLogFileCompleted += AdvancedRunLogFileRe-
cording_AdvancedRunLogFileCompleted;

private void AdvancedRunLogFileRecording_AdvancedRunLogFileCompleted(object sender,
AdvancedRunLogFileCompletedEventArgs e)
{
        //Read all completed recordings
        AdvancedList<AdvancedRunLogRecordInfo> runLogRecordInfos = AdvancedRunLog-
        FileRecording.GetAllCompletedRecordingInfos();
        //Run through all completed recordings of the extended runlog file
        foreach (AdvancedRunLogRecordInfo recordInfo in runLogRecordInfos)
        {
                //Read the corresponding extended RunLogFiles. Execute, if the file
                could be read out
                if (AdvancedRunLogFileRecording.GetAdvancedRunLogFile(recordInfo,
                out AdvancedRunLogFile advancedRunLogFile))
                {
                        /*Save the extended RunLogFile encrypted in the default
                        path associated with the device. If the run log files are
                        not to be saved encrypted, no AESCryptoInfo object may be
                        passed. In this example a default name is created for each
                        extended Run-Logfile*/
                        if (advancedRunLog-
                        File.Save(ApplicationSettings.DeviceFileSettings.GetAdvance
                        dRunLogFolderPath(recordInfo.DeviceDescriptionInfo), new
                        AESCryptoInfo("Test", EnAESKeySize.KEYSIZE_128)) == null)
                        {
                                /*Delete recorded extended runlog file so that fur-
                                ther recordings can be started*/
                                AdvancedRunLogFileRecord-
                                ing.DeleteCompletedRecordings(recordInfo);
                        }
                }
        }
}
```

The following example can be used to read the number of recordings that can still be started.

```
//Retrieve the number of recordings that can still be started
int iNumOfRecordings = AdvancedRunLogFileRecording.GetNumOfFreeRecordings();
```

The following example can be used to read the currently running recordings (devices) from the corresponding list.

```
//Read list of currently running recordings
AdvancedList<AdvancedRunLogRecordInfo> recordList = AdvancedRunLogFileRecord-
ing.GetAllRecordingInfos();
```

The following example can be used to stop recording the advanced run log files.

```
//Stop recording
AdvancedRunLogFileRecording.StopAdvancedRunLogFileRecording();
```

Other methods for managing advanced run log files are available within the static class AdvancedRunLogFileRecording. For a description, please refer to the library documentation of the Biometra Library.

5.7.3 Block monitoring

Block monitoring can be used to monitor a total of 10 selected blocks from different devices. Block monitoring is used to report certain status changes. Only status changes that are not caused by the Biometra Library's own instance are reported. The following example shows how an AlarmDevice is created and how block monitoring is started. Before the alarm monitoring is started, it is necessary to instantiate the corresponding event in which the status changes are reported.

```
//Instantiate event for the status messages
AlarmManagement.AlarmManagementMessageReceive += AlarmManage-
ment_AlarmManagementMessageReceive;
```

The following example can be used to start block monitoring.

```
//Read information for the device
DeviceCom.GetInformationsByDeviceDescription(deviceList[0], out DeviceInformations
deviceInformations);
//Create AlarmDevice
AlarmDevice alarmDevice = new AlarmDevice(deviceList[0], deviceInfor-
mations.NumOfBlocks, new AdvancedList<BlockNumber>() { new BlockNumber(1) });
//Load application settings
ApplicationSettings.LoadApplicationSettings();
//Add AlarmDevice to the list of devices to be monitored
ApplicationSettings.AlarmManageSettings.AlarmDeviceList.Add(alarmDevice);
//Save application settings
ApplicationSettings.SaveApplicationSettings();
//Start block monitoring and monitor all devices from the list
AlarmManage-
ment.SetMonitoringList(ApplicationSettings.AlarmManageSettings.AlarmDeviceList);
```

All cumulative messages are reported in the following event.

```
private void AlarmManagement_AlarmManagementMessageReceive(object sender, AlarmMan-
agementMessageEventArgs e)
{
        //Display all messages that have occurred
        foreach(String stMessage in e.NotificationList)
        {
                MessageBox.Show(stMessage);
        }
}
```

The following method can be used to read the number of monitoring events that can still be started.

```
//Retrieve the number of monitoring operations that can still be started
int iNumOfFreeMonitorings = AlarmManagement.GetNumOfFreeMonitorings();
```

The following method can be used to delete all messages that have occurred. A maximum of 100 messages per monitored block is stored. A total of 10 blocks can be monitored. When the maximum number of messages is reached, the oldest message is automatically removed.

```
//Delete all messages that have occurred
AlarmManagement.DeleteAlarmMessages();
```

The following example can be used to stop block monitoring.

```
//Stop monitoring
AlarmManagement.StopAlarmManagement();
```

Other methods in connection with block monitoring are available within the static class AlarmManagement. For a description, please refer to the library documentation of the Biometra Library.

5.8 Advanced self-test

The device offers the possibility to start an advanced self-test. The status indicating whether a self-test is active or not can be found within the BlockParams object, retrieved in the examples in section 5.7.1. The following sections show how to start and stop a self-test.

5.8.1 Starting a self-test

The following example can be used to start an advanced self-test.

```
try
{
        //Create communication object
        using (SelftestProtCmds selftestProtCmds = new Self-
        estProtCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Start extended self-test
                CheckStateResult checkStateResult = selft-
                estProtCmds.StartExtendedSelftest(deviceList[0]);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.8.2 Stopping a self-test

The following example can be used to stop an advanced self-test.

```
Try
{
        //Create communication object
        using (SelftestProtCmds selftestProtCmds = new Self-
        estProtCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Stop extended self-test
                CheckStateResult checkStateResult = selft-
                estProtCmds.StopExtendedSelftest(deviceList[0]);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

5.9 Documentation

Various documentation options are available on the device, which can be retrieved, stored and viewed using the Biometra Library. It is also possible to read and view the stored advanced run log files using the Biometra Library. The following sections describe how the individual documentations can be managed.

Please refer to the Biometra TSuite Software Operating Manual for more detailed information on the data contained in the various documentation files.

5.9.1 The run log file

Key program run data run is stored in the run log file:

- Name of the run log file
- Log time and date
- Start/end time
- Date
- Name of the company
- Software version
- User initials
- Cycler type
- Serial number
- Block type
- Serial number of the block
- Number of the block
- Messages

In connection with the run log file of a device, various options are available to retrieve and manage the corresponding data from the device. An example of the different options is described in the following sections.

5.9.1.1 Retrieving an overview of the run log files available on the device

To obtain information about the run log files available on the device, it is first necessary to retrieve an overview of all available run log files from the device. The RunLogInfos contains the respective number of the run log file, which can be retrieved from the device. The procedure is described in the following example.

```
try
{
        //Create communication object
        using (RunLogCmds runLogCmds = new RunLog-
        Cmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Retrieve an overview of the available run log files from
                the device
                CheckStateResult checkStateResult = runLog-
                Cmds.GetRunLogFileOverview(deviceList[0], out DataSet-
                List<RunLogInfos> runLogOverview);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

Depending on how many run log files are on the device, it may be wise to use the following asynchronous method so that the current processing is not blocked for too long.

```
//Create communication object
RunLogCmds runLogCmds = new RunLogCmds(ApplicationSettings.CommunicationSettings,
deviceList[0]);
//Instance event in which the list is returned
runLogCmds.GetRunLogFileOverview_Finished += RunLog-
Cmds_GetRunLogFileOverview_Finished;
//Retrieve overview from device
runLogCmds.GetRunLogFileOverviewAsync(deviceList[0]);

private void RunLogCmds_GetRunLogFileOverview_Finished(object sender, DeviceComEv-
entArgs<DataSetList<RunLogInfos>> e)
{
        //Execute, if processing was successful
        if (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Run through all RunLogInfos
                foreach (RunLogInfos runLogInfos in
                e.Result.ProcessResultData.DataList)
                {
                        //Put out information
                        MessageBox.Show(runLogInfos.ToString());
                }
        }
}
```

5.9.1.2 Retrieving the run log file from the device, managing and processing it

After the information about the available run log file has been retrieved from the device, the corresponding run log file can be retrieved and further managed and processed using the respective run log file number. The procedure is described in the following example.

```
try
{
        //Create communication object
        using (RunLogCmds runLogCmds = new RunLog-
        Cmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                /*Retrieve Run log file with the number 1 from the device. It is
                necessary to transfer the corresponding encryption information so
                that the run log file can be read correctly*/
                CheckStateResult checkStateResult = runLog-
                Cmds.GetRunLogFile(deviceList[0], new RunLogFileNumber(1), Ad-
                vancedBiometra LibraryData.PPRT_SEC_LOCK_P, out RunLogFile
                runLogFile);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

The run log file retrieved from the device can now be used for further management/processing. Firstly, the retrieved data can be displayed in a corresponding interface. For a description of the individual properties of the RunLogFile object, please refer to the Biometra Library library documentation. The following method can be used to save the retrieved run log file.

```
/*Save the run log file with the name "TestRunLog" encrypted in the standard direc-
tory with the password "TestPass". If the file name TestRunLog is not transferred,
an automatically generated file name is used. Furthermore, it is not necessary to
encrypt the file. The corresponding Save method can be used for this*/
CheckStateResult checkStateResult = runLog-
File.Save(ApplicationSettings.DeviceFileSettings.GetRunLogFolderPath(deviceList[0])
, "TestRunLog", new AESCryptoInfo("TestPass", EnAESKeySize.KEYSIZE_128));
```

The following example can be used to export the retrieved run log file.

```
//Export Run log file as CSV file named TestRunLogExport to the default directory
CheckStateResult checkStateResult = runLog-
File.Export(ApplicationSettings.DeviceFileSettings.GetRunLogFolderPath(deviceList[0
]), "TestRunLogExport", EnDeviceFileExportTypes.FILETYPE_CSV);
```

The available export file types can be read as follows.

```
//Read available export types
AdvancedList<EnDeviceFileExportTypes> exportTypes = RunLog-
File.SUPPORTED_EXPORT_FILE_TYPES_P;
```

The following example can be used to re-read a saved run log file.

```
//Create new RunLog object
RunLogFile runLogFile = new RunLogFile();
//Read stored run log file
CheckStateResult checkStateResult = runLog-
File.Import(Path.Combine(ApplicationSettings.DeviceFileSettings.GetRunLogFolderPath
(deviceList[0]), "TestRunLog.TXT");
```

To retrieve more than one run log file from the device, different methods are available within the RunLogCmds and RunLogFileWorker classes. The following section describes how the RunLogCmds class can be used to retrieve all run log files available on the device.

```
try
{
        //Create communication object
        using (RunLogCmds runLogCmds = new RunLog-
        Cmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Retrieve all available run log files from the device
                CheckStateResult checkStateResult = runLog-
                Cmds.GetAllRunLogFiles(deviceList[0], AdvancedBiometra LibraryDa-
                ta.PPRT_SEC_LOCK_P, out DataSetList<RunLogFile, RunLogInfos>
                resultList);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

The class RunLogFileWorker can be used to retrieve several run log files from the device and save them directly. The following example describes how to retrieve and save all available run log files.

```
try
{
        //Create communication object
        using (RunLogFileWorker runLogFileWorker = new RunLogFileWork-
        er(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Retrieve all available run log files and save them with the orig-
                inal encryption
                CheckStateResult checkStateResult = runLogFileWork-
                er.LoadAndSaveAllRunLogFiles(deviceList[0], ApplicationSet-
                tings.DeviceFileSettings.GetRunLogFolderPath(deviceList[0]),
                AdvancedBiometra LibraryData.ADV_PPRT_SEC_LOCK_P, out DataSet-
                List<RunLogInfos> notGettedFiles);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

As an alternative to the synchronous method, an asynchronous method is also available so that the current processing is not blocked while the retrieval and storage process takes place.

```
//Create communication object
RunLogFileWorker runLogFileWorker = new RunLogFileWork-
er(ApplicationSettings.CommunicationSettings, deviceList[0]);
//Instantiate return event
runLogFileWorker.LoadAndSaveAllRunLogFiles_Finished += RunLogFileWork-
er_LoadAndSaveAllRunLogFiles_Finished;
//Retrieve and save all existing run log files from the device
runLogFileWorker.LoadAndSaveAllRunLogFilesAsync(deviceList[0], ApplicationSet-
tings.DeviceFileSettings.GetRunLogFolderPath(deviceList[0]), AdvancedBiometra Li-
braryData.ADV_PPRT_SEC_LOCK_P);

private void RunLogFileWorker_LoadAndSaveAllRunLogFiles_Finished(object sender,
DeviceFileWorkEventArgs<DataSetList<RunLogInfos>> e)
{
        bool bProcessOkay = false;
        //Execute, if processing was successful
        if (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set flag
                bProcessOkay = true;
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());
}
```

In addition, further methods are available in connection with the run log files.

5.9.2 Advanced run log file

In addition to the data contained in the run log file, the advanced run log file continuously stores temperatures of the program run. Due to the sheer volume of this data, it is not saved to the thermocycler, but to the local storage location (generally the computer).

To create advanced run log files, you can use the examples described in section 5.7.2. The advanced run log files can be read from the corresponding storage location for further processing/management using the following example.

```
try
{
        //Create object to be read in
        using (AdvancedRunLogFileWorker runLogFileWorker = new AdvancedRunLogFile-
        Worker())
        {
                //Read in all stored extended run log files
                CheckStateResult checkStateResult = runLogFileWork-
                er.ReadAllSavedAdvancedRunLogInfosToShow(ApplicationSettings.Device
                FileSettings.GetAdvancedRunLogFolderPath(deviceList[0]), new AES-
                CryptoInfo("Test", EnAESKeySize.KEYSIZE_128), out DataSet-
                List<AdvancedRunLogInfosToShow, String> runLogInfos);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

As an alternative to the synchronous method, an asynchronous method is also available so that the current processing is not blocked while the read process takes place. This example shows how the advanced run log files can be read and exported.

```
//Create object to be read in
AdvancedRunLogFileWorker runLogFileWorker = new AdvancedRunLogFileWorker();
//Instantiate return event
advancedRunLogFileWorker.ReadAllSavedAdvancedRunLogInfosToShow_Finished += Ad-
vancedRunLogFileWorker_ReadAllSavedAdvancedRunLogInfosToShow_Finished;
//Read in information about all stored extended run log files
runLogFileWork-
er.ReadAllSavedAdvancedRunLogInfosToShowAsync(ApplicationSettings.DeviceFileSetting
s.GetAdvancedRunLogFolderPath(deviceList[0]), new AESCryptoInfo("Test", EnAES-
KeySize.KEYSIZE_128));

private void AdvancedRunLogFileWork-
er_ReadAllSavedAdvancedRunLogInfosToShow_Finished(object sender, DeviceFile-
WorkEventArgs<DataSetList<AdvancedRunLogInfosToShow, string>> e)
{
        //Execute, if processing was successful
        if (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Scroll through all information
                foreach(AdvancedRunLogInfosToShow advancedRunLogInfosToShow in
                e.Result.ProcessResultData.DataList)
                {
                        //Create new object
                        AdvancedRunLogFile advancedRunLogFile = new AdvancedRunLog-
                        File();
                        //Read extended run log file
                        CheckStateResult checkStateResult = advancedRunLog-
                        File.Import(advancedRunLogInfosToShow.CompleteFilePath, new
                        AESCryptoInfo("Test", EnAESKeySize.KEYSIZE_128));
                        //Execute when the file has been read in
                        if (checkStateResult == null)
                        {
                                //Export extended run log file
                                checkStateResult = advancedRunLog-
                                File.Export(ApplicationSettings.DeviceFileSettings.
                                GetAdvancedRunLogFolderPath(deviceList[0]), ad-
                                vancedRunLogInfosToShow.ExportFileDescription,
                                EnDeviceFileExportTypes.FILETYPE_CSV);
                        }
                }
        }
}
```

5.9.3 Error log file

The last 30 errors that occurred are stored on the device. The following example can be used to retrieve the available errors. The class ErrorLogCmds also contains other methods relating to error log files.

```
try
{
        //Create communication object
        using (ErrorLogCmds errorLogCmds = new ErrorLog-
        Cmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Retrieve all existing errors from the device
                CheckStateResult checkStateResult = errorLog-
                Cmds.GetAllErrorLogFiles(deviceList[0], out DataSet-
                List<LogFileData, LogFileNumber> errorList);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

Since the process of retrieving the error log files can take longer depending on the number of errors, use the corresponding asynchronous method to run the process in a background job. The following example can be used to do this. This example also shows how the progress can be reported during the retrieval process.

```
//Create communication object
ErrorLogCmds errorLogCmds = new ErrorLog-
Cmds(ApplicationSettings.CommunicationSettings, deviceList[0]);
//Instance event that is triggered after the error is retrieved
errorLogCmds.GetAllErrorLogFiles_Finished += ErrorLog-
Cmds_GetAllErrorLogFiles_Finished;
//Instance the event that is triggered to update the progress of the error log file
errorLogCmds.GetAllErrorLogFilesProgressUpdate += ErrorLog-
Cmds_GetAllErrorLogFilesProgressUpdate;
//Start the procedure to retrieve the available errors
errorLogCmds.GetAllErrorLogFilesAsync(deviceList[0]);

private void ErrorLogCmds_GetAllErrorLogFilesProgressUpdate(object sender, Process-
ProgressUpdateEventArgs e)
{
        //Output approximate remaining time
        MessageBox.Show(e.CurrentProgress.ProcessRemainingTime.ToString());
        //Show the current progress
        MessageBox.Show(e.CurrentProgress.Duration.ToString());
        //Output current percentage of progress
        Message-
        Box.Show(e.CurrentProgress.ProgressPercentage.CurrentPercentage.ToString())
        ;
}

private void ErrorLogCmds_GetAllErrorLogFiles_Finished(object sender, DeviceComEv-
entArgs<DataSetList<LogFileData, LogFileNumber>> e)
{
        //Execute, if processing was successful
        if (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Run through all errors
                foreach (LogFileData logFileData in
                e.Result.ProcessResultData.DataList)
                {
                        //Put out error data
                        MessageBox.Show(logFileData.ToString());
                }
        }
}
```

5.9.4 Self-test file

At regular intervals, every Biometra thermocycler will automatically recommend carrying out a detailed self-test. Although it is not absolutely necessary to perform the test, it is highly recommended to follow the prompt and to let the device check itself.

In order to obtain reproducible and correct results, certain conditions must be observed. Please refer to the manual of the corresponding thermocycler.

During the detailed self-test, the thermocycler checks various functions and components:

Test	Function
Cooler	Incubates the sample block to 4°C and checks if the temperature in the block is reached and can be held for a longer time.
Thermal tracking	Checks the synchronicity of the control circuits and whether they work together in a coordinated way.
Heating/cooling rate	Checks whether the specified average heating and cooling rate is reached.
Refrigeration	Checks whether the heat sink and the fans are working properly together.
Gradient[1]	Checks if the sample block reaches the set gradient temperatures.
Heated lid	Tests if the heated lid reaches the set temperature and can hold it for a longer time.
Regulation	Tests if the sample block is controlled properly.

[1] only for devices with gradient function

The result of every test is listed as either "Passed" or "Error".

After an advanced self-test has been performed on the device, the associated log can be retrieved and further managed and processed. The procedure is described in the following example.

```
try
{
        //Create communication object
        using (SelftestProtCmds selftestCmds = new Selft-
        estProtCmds(ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                /*Retrieve the self-test log from the device. It is necessary to
                transfer the corresponding encryption information so that the pro-
                tocol can be read correctly*/
                CheckStateResult checkStateResult = selft-
                estProtCmds.GetAllSelftestProtocols(deviceList[0], AdvancedBiometra
                LibraryData.EXT_SEC_LOCK_P, out AllSelftestData selftestData);
}
```

```
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

Since the process of retrieving the self-test log can take longer depending on the number of tests, it is a good idea to use the corresponding asynchronous method to run the process in a background job. The following example can be used to do this.

```
//Create communication object
SelftestProtCmds selftestProtCmds = new Selft-
estProtCmds(ApplicationSettings.CommunicationSettings, deviceList[0]);
//Instantiate return event
selftestProtCmds.GetAllSelftestProtocols_Finished += Selft-
estProtCmds_GetAllSelftestProtocols_Finished;
//Retrieve self test logs
selftestProtCmds.GetAllSelftestProtocolsAsync(deviceList[0], AdvancedBiometra Li-
braryData.EXT_SEC_LOCK_P);

private void SelftestProtCmds_GetAllSelftestProtocols_Finished(object sender, De-
viceComEventArgs<AllSelftestData> e)
{
        bool bProcessOkay = false;
        //Execute, if processing was successful
        if (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set flag
                bProcessOkay = true;
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());
}
```

The advanced self-test log retrieved from the device can now be used for further management or processing. The retrieved data can be displayed in a corresponding interface. For a description of the individual properties of the AllSelftestData object, please refer to the Biometra Library library documentation. The following method can be used to save the retrieved log.

```
/*Save the self-test protocol with the name "TestSelftest" encrypted in the stand-
ard directory with the password "TestPass". If the file name TestSelftest is not
transfered, an automatically generated file name will be used. Furthermore, it is
not necessary to encrypt the file. The corresponding Save method can be used for
this*/
CheckStateResult checkStateResult = selft-
estData.Save(ApplicationSettings.DeviceFileSettings.GetExtSelftestFolderPath(device
List[0]), "TestSelftest", new AESCryptoInfo("TestPass", EnAESKeySize.KEYSIZE_128));
```

The following example can be used to export the retrieved log.

```
//Export self-test log as CSV file named TestSelftestExport to the default directo-
ry
CheckStateResult checkStateResult = selftestData.Export(ApplicationSettings.
GetExtSelftestFolderPath.GetExtSelftestFolderPath(deviceList[0]), "TestSelftestEx-
port", EnDeviceFileExportTypes.FILETYPE_CSV);
```

The available export file types can be read as follows.

```
//Read in stored self-test protocol
AdvancedList<EnDeviceFileExportTypes> exportTypes = AllSelft-
estData.SUPPORTED_EXPORT_FILE_TYPES_P;
```

The following example can be used to re-read a saved self-test log.

```
//Create new self-test object
AllSelftestData selftestData = new AllSelftestData();
//Read available export types
CheckStateResult checkStateResult = selft-
estData.Import(Path.Combine(ApplicationSettings.DeviceFileSettings.GetExtSelftestFo
lderPath(deviceList[0]), "TestSelftest.TXT");
```

The ExtSelftestFileWorker class can be used to retrieve the self-test log from the device and save it directly. The procedure to do this is described in the following example.

```
try
{
        //Create communication object
        using (ExtSelftestFileWorker extSelftestFileWorker = new ExtSelftestFile-
        Worker (ApplicationSettings.CommunicationSettings, deviceList[0]))
        {
                //Retrieve all available self-test logs and save them with the
                original encryption
                CheckStateResult checkStateResult = extSelftestFileWork-
                er.LoadAndSaveExtSelftestFile(deviceList[0], ApplicationSet-
                tings.DeviceFileSettings.GetExtSelftestFolderPath(deviceList[0]),
                AdvancedBiometra LibraryData.EXT_SEC_LOCK_P);
        }
}
/*Catch exception, because an exception is triggered when the device has returned
an error message.*/
catch (Exception ex) { MessageBox.Show(ex.Message); }
```

As an alternative to the synchronous method, an asynchronous method is also available so that the current processing is not blocked while the retrieval and storage process takes place.

```
//Create communication object
ExtSelftestFileWorker extSelftestFileWorker = new ExtSelftestFileWork-
er(ApplicationSettings.CommunicationSettings, deviceList[0]);
//Instantiate return event
extSelftestFileWorker.LoadAndSaveExtSelftestFile_Finished += ExtSelftestFileWork-
er_LoadAndSaveExtSelftestFile_Finished;
//Retrieve and save the self-test protocol from the device
extSelftestFileWorker.LoadAndSaveExtSelftestFileAsync(deviceList[0], Application-
Settings.DeviceFileSettings.GetExtSelftestFolderPath(deviceList[0]), Advanced-
Biometra LibraryData.EXT_SEC_LOCK_P);

private void ExtSelftestFileWorker_LoadAndSaveExtSelftestFile_Finished(object send-
er, DeviceFileWorkEventArgs<object> e)
{
        bool bProcessOkay = false;
        //Execute, if processing was successful
        if (e.BackgroundCompletedState == EnBackgroundCompleted-
        State.COMPLETEDSTATE_OK)
        {
                //Set flag
                bProcessOkay = true;
        }
        //Put out value of the flag
        MessageBox.Show(bProcessOkay.ToString());
}
```

In addition, further methods are available in connection with the self-test logs.

6 Service

If there are any problems with the device or software, please contact the Service department or your local Analytik Jena dealer. For the address of the Analytik Jena Service department, please refer to the inside front cover of this operating manual. Please observe the return information (see chapter "Returning a device" p. 47) if you wish to return a device to Biometra.

6.1 Returning a device

WARNING

Risk of damage to health due to improper decontamination! Perform a professional decontamination before returning the device to Analytik Jena.

NOTE

Biometra/Analytik Jena must refuse acceptance of contaminated devices. The sender may be liable for any damage caused by inadequate decontamination of the device.

- Please clean all device components from biologically hazardous, chemical and radioactive contamination.

Download the Declaration of Decontamination as an editable PDF document in German or English from the Internet:

https://www.analytik-jena.com/fileadmin/content/service/customer/Declaration_of_decontamination_en_01.pdf

Complete the form and attach the signed decontamination declaration to the outside of the shipment.

- Only use the original packaging for the shipment and insert the transport lock. If the original packaging is no longer available, please contact Biometra GmbH or your local dealer.
- Please attach the warning note "CAUTION! SENSITIVE ELECTRONIC DEVICE!" to the packaging.
- Please include a sheet containing the following data:
  - Name and address of the sender
  - Name and telephone number of a contact for inquiries
  - A detailed description of the fault, the precise conditions and situations under which the fault occurs
  - If you need to send in your device, the service department will assign you a RAN which you will receive over the phone. This number must be attached to the outside of the packaging so that it is clearly visible. **Devices submitted without a RAN will not be accepted!**