**WinKit 6 Communications and Scripting**

**Version 605.4**

**Updated October 29, 2012**

Contents

[1. Introduction 4](#_Toc334122098)

[2. Terminology 5](#_Toc334122099)

[3. WinKit 6 Communication Layers 6](#_Toc334122100)

[3.1 CPS features and architecture 6](#_Toc334122101)

[3.2 Scripting features 8](#_Toc334122102)

[3.3 Content of Communication Libraries: 8](#_Toc334122103)

[4. CPS API 9](#_Toc334122104)

[4.1 Communication Parameters 9](#_Toc334122105)

[4.2 Base Communication Parameters and Connection Modifiers 11](#_Toc334122106)

[4.3 Creating connection to device 12](#_Toc334122107)

[4.4 Device state (Device information) 13](#_Toc334122108)

[4.5 Creating connection to secondary processors 14](#_Toc334122109)

[4.6 CPS functions 16](#_Toc334122110)

[4.6.1 Device context management 18](#_Toc334122111)

[4.6.2 Authentication functions 21](#_Toc334122112)

[4.6.3 Memory Manipulation Functions 23](#_Toc334122113)

[4.6.4 File System functions 27](#_Toc334122114)

[4.6.5 Maintenance functions 35](#_Toc334122115)

[4.6.6 Serial port management 39](#_Toc334122116)

[4.6.7 Direct execution of DNP requests 39](#_Toc334122117)

[4.7 Errors handling 40](#_Toc334122118)

[5. Scripting API 40](#_Toc334122119)

[5.1 Basic Information 40](#_Toc334122120)

[5.2 WinKit 6 Cmdlets and Objects 41](#_Toc334122121)

[5.3 Cmdlets 42](#_Toc334122122)

[5.3.1. Get-Device 42](#_Toc334122123)

[5.3.2. Get-DataProvider 43](#_Toc334122124)

[5.3.3.Update-Firmware 43](#_Toc334122125)

[5.3.4.Save-Memory 44](#_Toc334122126)

[5.3.5. Restore-Memory 45](#_Toc334122127)

[5.4 Objects 45](#_Toc334122128)

[5.4.1. DeviceWrapper 45](#_Toc334122129)

[5.4.2.DataProviderWrapper 48](#_Toc334122130)

[5.5 IntelliLINK Scripts 49](#_Toc334122131)

[5.5.1 IntelliLINK Built-In Scripts 49](#_Toc334122132)

[5.5.2 User-authored Scripts 49](#_Toc334122133)

[5.5.3 IntelliLINK Built-In PowerShell Host (constrains and specialities) 50](#_Toc334122134)

[6. CPS Server 52](#_Toc334122135)

[7. CPS Console 53](#_Toc334122136)

[8. Appendix 1. Usage Basics 54](#_Toc334122137)

[7.1 Logging 54](#_Toc334122138)

[7.1.1 Configuring logging programmatically 54](#_Toc334122139)

[7.1.2 Configuring logging with an application config file 56](#_Toc334122140)

[7.2 IAsyncRequest interface 57](#_Toc334122141)

[7.3 Virtual Memory Manipulation Interface 59](#_Toc334122142)

[9. Appendix 2. Base Protocol Services 61](#_Toc334122143)

[8.1 Overview 61](#_Toc334122144)

[8.2 Main features 62](#_Toc334122145)

[8.3 Using BPS library 63](#_Toc334122146)

[8.3.1 Configuring client application 63](#_Toc334122147)

[8.3.2 Creating a device connection 63](#_Toc334122148)

[8.3.3 Communication parameters 64](#_Toc334122149)

[8.3.4 Communication Channels 66](#_Toc334122150)

[8.3.5 Connections. 66](#_Toc334122151)

[8.4 Important limitations of BPS 68](#_Toc334122152)

[8.5 Program example 69](#_Toc334122153)

[8.6 Working over DNP 70](#_Toc334122154)

[8.6.1 DNP Objects 70](#_Toc334122155)

[8.6.2 DNP Object Sets 70](#_Toc334122156)

[10. Appendix 3. Scripts localization 73](#_Toc334122157)

# Introduction

#### Applicable Software

This document describes methods of using WinKit 6 communication libraries and PowerShell script engine for communicating with devices and also describes these libraries’ architecture, basic underlying principles and algorithms. For questions regarding the applicability of information in this document to future releases, please contact S&C Electric Company.

#### About This Manual

This manual contains the following sections:

#### Terminology

Describes terms and definitions used.

#### WinKit 6 Communication Layers

Provides a brief description of the hierarchy of the WK6 communication libraries and the main functionality of these libraries.

#### CPS Overview

Describes the basic information about the CPS .

#### CSP Client API

Explains how to create a CPS based application and use the CPS API to work with devices.

#### Scripting API

Explains how to use the CPS and AppLevel API from PowerShell scripts to work with devices.

#### CPS Console

Describes an application for CPS server management.

#### Appendixes

Describes basic WK6 patterns and provides more details about communication libraries.

#### Document Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Ver** | **Date** | **By** | **Change Summary** |
| 600.58 Rev 1 | 01/28/12 | V. Ostapchuk | Initial Version |

# Terminology

#### DNP master

The device or program that monitors and manages other DNP device(s), such as SCADA master stations.

#### DNP slave (DNP device)

The device or program that is managed and represents its state by DNP protocol. The device states and available operations are described in a special XML file called DNP3 Device profile.

#### ICP device

The device that is managed and represents its state by ICP protocol.

#### Channel

A channel represents a particular physical link between a local computer and a remote device. There are three types of channels: serial, TCP/IP, UDP/IP. More channel types may be added in the future.

#### Communication parameters

A parameter set for establishing a connection with a device. The whole protocol parameter set can be divided into protocol parameters and channel parameters. For example for a DNP device connected over serial port, communication parameters are DNP parameters (local DNP address, peer DNP address) and serial port parameters (name, baud rate)

#### Communication Unit (Processor)

Some devices have several units (processors) on-board and it is possible to work with any of these processors independently. Such devices can be represented as two (or more) independent

devices with different communication parameters.

#### BPS

Base Protocol Services library. This library implements specific protocols (DNP, ICP).

#### CPS (CPS API)

Control Platform Services library. This library abstracts a certain functionality of the control platform (represented primarily by the EOS) in a protocol-neutral way. This functionality also is called CPS API.

#### AppLevel

This is an API to be used by applications to interact with controls (and other sources of data) in terms of named and typed objects.

#### CPS Client

A program or a .NET library using the CPS API for device management.

# WinKit 6 Communication Layers

The data/communications architecture of WinKit 6 has the following three layers (from lowest to highest) and a scripting engine for the access and management of the devices:

* BPS (Base Protocol Services). This layer implements specific protocols ( for example, DNP). BPS functions allow you to perform DNP communications to any DNP device (master or slave, by any manufacturer). BPS libraries can be used to write any DNP-based software (same for other supported protocols).
* CPS (Control Platform Services). This layer abstracts certain functionality of the control platform (represented primarily by the EOS) *in a protocol-neutral way*. This level is the basis for implementing maintenance functionality. Example: to exchange files with Compact Flash, you use the same CPS API, regardless of which protocol is being used.
* Application Layer Services (AppLevel). This layer provides an API to be used by applications to interact with controls (and other sources of data) in terms of named and typed objects. The API can be accessed locally, as a DLL function or remotely, over WCF (which is what HDE does). The API does not depend on the BPS-level protocol or the control platform.
* Scripting. This is WK6 extension for PowerShell to allow writing scripts for managing the devices faster and easier. Special IDE is not required for writing scripts.

Applications use Application Layer Services. Any platform-specific maintenance functionality is implemented using CPS functions. BPS functions are not normally used directly except for writing additional protocol-specific software (a DNP test set would be an example of that).

From the communication point of view the most important components are CPS library and scripting.

## 3.1 CPS features and architecture

The CPS abstracts certain functionality of the control platform (represented primarily by the EOS) in a ***protocol-neutral way***.

Main purposes of CPS (Control Platform Services) are:

* Provide simultaneous access to one physical device from several clients. The clients may be different applications (processes) running on the same or different computers, or different tasks being executed in the context of one process. The main problems that are solved by CPS are sharing of communications resource (serial port or socket) in windows environment (only one process can correctly work with the resource) and multiplexing requests from different clients to one physical device, creating an illusion for clients that they communicate directly to the device without even knowing that other clients exist (in a sense of proxy functionality).
* Provide unified abstract layer of functionality for common tasks such as reading/writing of virtual memory, compact flash-files access, querying device attributes (ident, revision, capabilities) uniformly and independently (to some extent) of device platform and communications protocol being used.

The CPS uses BPS for communication with devices and allows to establish a connection by the following protocols:

* DNP ,
* ICP

over the following physical channels:

* serial port,
* TCP,
* UDP.

The CPS provides the following functional:

* querying the device configuration (attributes);
* access to physical or virtual memory of the device;
* access to internal file system of the device;
* mechanism for the device firmware upgrading;
* support for the device authentication;
* the device data/time setting;
* protocol depended functions (Read CLASS0, Select/Operate); etc.

These functions can be used for the implementation of monitoring or maintenance functionality.

CPS has a client/server architecture based on WCF. This architecture makes it possible to provide several client applications with simultaneous access to a single device. For communicating with devices CPS uses BPS.

**Common Communication schema:**

WCF

DNP

…

Client Application 1 (using CPS Client)

Client Application N (using CPS Client)

Local CPS Server (using BPS)

Device (IntelliRupter)

**User Host**

The distributives including CPS library (for example IntelliLINK 6) contain simple CPS server implementation named CpsServer.exe and this CPS server can work as Windows service.

## 3.2 Scripting features

The scripting provides a set of the most of CPS functions, such as:

* file system functions;
* read/write virtual memory;
* authentication;
* maintenance functions(ResetDevice, UpdateFirmware);

and some of AppLevel functions:

* read/write named object;
* save/restore predefined object set (setpoint groups)

## 3.3 Content of Communication Libraries:

The Communication Libraries are implemented as a set of .NET assemblies listed in the table below. A .NET program using CPS(BPS) should reference these assemblies.

|  |  |  |
| --- | --- | --- |
| **Component** | **Library** | **Description** |
| **BPS** | SandC.WinKit6.Communications.BPS.dll | Base protocol services |
| SandC.WinKit6.Communications.Dnp.dll | Implementation of DNP protocol |
| **CPS** | SandC.WinKit6.Communications.CPS.Client.dll | Common API definitions for CPS and Remote client to Communication Server |
| SandC.WinKit6.Communications.CPS.Engine.dll | CPS server implementation |
| SandC.WinKit6.Common.dll | General definitions used throughout WinKit6, e.g. logging, virtual memory objects, etc. |
| **Scripting** | SandC.WinKit6.Scripting.dll | PowerShell module that contains cmdlets. |

The assemblies have accompanying documentation (.chm) files.

|  |  |
| --- | --- |
| BPS.ClassLibrary.chm | Base Protocol Services documentation, including DNP |
| CPS.Client.ClassLibrary.chm | Control Platform Services documentation; both CPS client side and Scripting. |

# CPS API

The CPS abstracts certain functionality of the control platform (represented primarily by the EOS) in a ***protocol-neutral way***.

All requests to a physical device are potentially long since they require communications to the device and thus are implemented using IAsyncRequest pattern (see Appendix 1). That is, they always have both synchronous and asynchronous variants of the request. In the examples below we will discuss (for brevity) only synchronous requests.

## 4.1 Communication Parameters

To establish a connection to a device, the user should specify a set of communication parameters. A whole set of communication parameters needed to establish a connection to a device is represented by ICommParameters interface:

public interface ICommParameters

{

IProtocolParameters ProtocolParameters { get; }

IChannelParameters ChannelParameters { get; }

Processor TargetProcessor { get; }

int Retries { get; }

int Timeout { get; } /// if value is 0 then default value will be used.

}

IProtocolParameters is a base interface for several particular protocol parameter classes:

public interface IDnpProtocolParameters : IProtocolParameters

{

bool Master { get; } /// Should be true.

ushort LocalDnpAddress { get; } /// Address of a local/master DNP device.

ushort PeerDnpAddress { get; } /// Address of a peer/slave DNP device.

}

public interface IIcpProtocolParameters : IProtocolParameters

{

ushort Address { get; } /// ICP Address of a device.

}

IChannelParameters is a base class for several channel parameter classes:

public interface IChannelParameters

{

ChannelType ChannelType { get; }

}

public interface ISerialChannelParameters : IChannelParameters

{

string PortName { get; }

int BaudRate { get; }

}

public interface INetworkChannelParameters : IChannelParameters

{

IPEndPoint Local { get; }

IPEndPoint Remote { get; }

}

The classes below implement these interfaces.

|  |  |
| --- | --- |
| **Class** | **Description** |
| DnpProtocolParameters | Represents DNP communication parameters. It implements IDnpProtocolParameters. |
| IcpProtocolParameters | Represents ICP communication parameters. It implements IIcpProtocolParameters. |
| SerialChannelParameters | Represents parameters for communication using a serial port. It implements ISerialChannelParameters. |
| TcpChannelParameters | Represents parameters for communication using a TCP/IP. It implements INetworkChannelParameters. |
| UdpChannelParameters | Represents parameters for communication using a UDP/IP. It implements INetworkChannelParameters. |
| CommParameters | Complete communication parameters. It implements ICommParameters. This class sets the following defaults:  Timeout = 500,  Retries = 0,  TragetProcessor = MCU |

**Examples:**

* Creating parameters for connection to a DNP slave device over UDP.

CommParameters dnpConnectionParams = new CommParameters() {

ChannelParameters = new UdpChannelParameters(

IPAddress.Parse("10.4.0.17"), 20000,

IPAddress.Parse("10.8.0.239"), 20000),

ProtocolParameters = new DnpProtocolParameters(**true**, 2, 65532),

Timeout = 2000,

Retries = 0

};

* Creating parameters for connection to ICP device over serial port.

CommBaseParameters dnpSlaveParams = new CommBaseParameters() {

ChannelParameters = new SerialChannelParameters("COM1", 9600),

ProtocolParameters = new IcpProtocolParameters(0x1ff0),

Timeout = 2000,

Retries = 0

};

By default a connection is established with the main processor(MCU). But you can set other processor manually using TargetProcessor property.

* Creating parameters for connection to DSP over DNP/TCP.

CommParameters mcuParams = new CommParameters() {

ChannelParameters = new TcpChannelParameters(

IPAddress.Parse("10.8.0.239"), 20000),

ProtocolParameters = new DnpProtocolParameters(true, 2, 65532),

Timeout = 2000,

Retries = 0,

TargetProcessor = Processor.DSP

};

## 4.2 Base Communication Parameters and Connection Modifiers

All the communication parameters fall into two groups: *base parameters*, which define the identity of the device, and *connection modifiers*, which set various options of the connection. The difference between the two is important: changing a single base parameter means connecting (generally) to a different device, while a connection to one and the same device may be made using various values of connection modifiers.

Base parameters are:

* ChannelParameters;
* ProtocolParameters;
* Retries;
* Timeout;
* TargetProcessor;

Connection modifiers are:

* BaudRate for serial connections;
* Timeouts;
* Retries;

When a connection is created and there is no previously created connection with the same base parameters, the connection is created and connection modifiers are used to establish communications with the device.

When a connection with some communication parameters is created and there already exists a connection with the same *base parameters,* the library does not create a new connection but uses the existing one instead. Connection modifiers are not applied in this case.

If particular values of connection modifiers are needed for a particular application, the user should not rely on specifying connection modifiers in initial connection establishment. Instead, she should create a connection and then check the values of connection modifiers and set them as needed.

## 4.3 Creating connection to device

The main interface through which a user gets access to CPS services is the **CpsClient** class (in SandC.WinKit6.Communications.CPS namespace). This class allows creating an instance of IDevice interface. This instance represents a connection to server through which various requests related to a physical device with configured communication parameters may be executed.

IDevice may be created through CpsClient.CreateDevice method overloads. In all variants of the method, one of the parameters is ICommParameters which specifies communication parameters to be used for connecting a server to device. Various creating methods vary in specifying how the connection of the client to server is established.

When IDevice instance is initialized, it checks connection to the server and if the server is not running, it attempts to start the server using the supplied server configuration information.

|  |  |
| --- | --- |
| Creation method | Description |
| CreateDevice(commParams) | Standard named pipe URI is used for connecting to CPS server. Server information is taken from the registry, as set by the server service installer. |
| CreateDevice(pipeName, commParams) | pipeName is used for connecting to CPS server. Server information is taken from the registry, as set by the server service installer. |
| CreateDevice(commParams, cpsServiceName) | Standard named pipe URI is used for connecting to CPS server and if the server is not running then windows service with the name cpsServiceName will be attempted to start (only if there is administrator permission). |
| CreateDevice(pipeName, commParams, cpsServiceName) | pipeName is used for connecting to CPS server and if the server is not running then windows service with the name cpsServiceName will be attempted to start (only if there is administrator permission). |
| CreateDevice(commParams, cpsServerApp, execArgs) | Standard named pipe URI is used for connecting to CPS server and if the server is not running then cpsServerApp with execArgs will be executed. |
| CreateDevice(pipeName, commParams, cpsServerApp, execArgs) | pipeName is used for connecting to CPS server and if the server is not running then cpsServerApp with execArgs will be executed. |

**Example**

* Creating a connection to MCU over DNP/UDP.

CommParameters mcuParams = new CommParameters() {

ChannelParameters = new UdpChannelParameters(

IPAddress.Parse("10.4.0.17"), 20000,

IPAddress.Parse("10.8.0.239"), 20000),

ProtocolParameters = new DnpProtocolParameters(true, 2, 65532),

Timeout = 2000,

Retries = 0,

Processor = Processor.MCU

};

IDevice mcu = CpsClient.CreateDevice(mcuParams);

* Creating a connection to DSP over ICP/Serial.

CommParameters dspParams = new CommParameters() {

ChannelParameters = new SerialChannelParameters("COM1", 9600)

IPAddress.Parse("10.4.0.17"), 20000,

IPAddress.Parse("10.8.0.239"), 20000),

ProtocolParameters = new IcpProtocolParameters(0x1ff1),

Timeout = 2000,

Retries = 0,

Processor = Processor.DSP

};

IDevice dsp = CpsClient.CreateDevice(dspParams);

After creation the IDevice instance must be initialized in the first request. This request connects a client to WCF server and gives parameters for communication with a device to the server and waits when connection to the device is established.

IDeviceRequest request = mcu.CreateRequest();

CpsError error = request.Init();

if (error != null)

{

Console.WriteLine("Initialization failed: {0}", res.Message);

}

Any IDevice instance is also IVirmMemProvider instance and it can be used for creating IAppDataProvider instances from AppLevel.

## 4.4 Device state (Device information)

The IDevice instance contains the following base properties.

|  |  |
| --- | --- |
| **Property** | **Description** |
| IsInitialized | Indicating whether the connection to device is initialized (established). If it is true, the Info property contains valid information and all requests may be performed. If it is false, only IDeviceRequest.Init request may be executed. |
| Info | Provides basic information about this device and its capabilities. This property is available only if this device is initialized (established). |

When connection is established the Info property represents actual information about the device.

|  |  |
| --- | --- |
| Property | Description |
| Processor | current(connected) processor. |
| Series | device series (e.g., Series\_2XX, Series\_Spirit\_MCU, etc.) |
| OSVersion | EOS version |
| Mode | represents device mode (application or maintenance). |
| Capabilities | Capabilities of the device. This is a bit mask containing the following:   * ICP * DNP * Virtual Memory * Application defined protocol * Compact Flash * Authentication * Multiple processors |
| AuthVersion | authentication version supported by application |
| Processors | information about all processors of the device |
| LastResetStatus | information from 0x200-3 addresses |
| CreationTime | timestamp -when this info was updated last time |
| AppIdent | application ident. |
| AppVersion | application version. |
| HmiVersion | computed from AppVersion according to CCP Platform conventions. |

If Capabilities property contains only DNP capability then device is not S&C device and most CPS functionality will be inaccessible.

The extended device information can be queered using ReadInfo request. See 4.6.5.

## 4.5 Creating connection to secondary processors

Some controller architectures include an internal routing of communication requests to various controller subsystems. Particular details of this routing may be different for different architectures.

For the CCP platform, there are MCU and (possibly multiple) DSP processors inside the controller.

For the ICP protocol, addressing a particular processor inside the controller is achieved by using a dedicated ICP address. If DSP is a single processor this address is 1FF0. For the DNP protocol, addressing the ICP controller when the physical connection is made to the MCU board is achieved by using a special virtual address mapping scheme. The TargetProcessor parameter, applicable only to the DNP protocol connections, is used to designate which of the processors inside the controller should handle the request.

A common task is to “switch” a connection from MCU to DSP, or from DSP to MCU, inside the same controller. E.g., this is used when downloading firmware to the controller.

When communication parameters for the initial connection are given, and connection has been established, connection parameters for another processor in the same controller may be obtained by creating the instance of CommParameters class and passing the needed processor as constructor argument. The obtained CommParameters class may be used to create a new IDevice for the processor.

The helper function CreateDevice on IDevice interface creates an instance of IDevice for the specified processor.

**Examples:**

* Creating connection to IntelliRupter

CommParameters ir1params = new CommBaseParameters() {

ChannelParameters = new UdpChannelParameters(

IPAddress.Parse("10.4.0.17"), 20000,

IPAddress.Parse("10.8.0.239"), 20000),

ProtocolParameters = new DnpProtocolParameters(**true**, 2, 65532),

Timeout = 2000,

Retries = 0

TragetProcessor = Processor.MCU

};

IDevice mcu = CpsClient.CreateDevice(ir1params);

IDeviceRequest request = mcu.CreateRequest();

CpsError error = request.Init();

if (error = null)

{

Console.WriteLine("MCU connection failed: {0}", error.Message);

}

reques.Close();

* Creating of the secondary(DSP) processor

IDevice dsp = mcu.CreateDevice(Processor.DSP);

request = dsp.CreateRequest();

CpsError error = request.Init();

if (error = null)

{

Console.WriteLine("DSP connection failed: {0}", error.Message);

}

reques.Close();

## 4.6 CPS functions

All CPS functions (requests) are grouped according to the subsystem they are related to, and each group of requests uses its own specialized interface based on IAsyncRequest .

|  |  |  |
| --- | --- | --- |
| **Functional group** | **Creation method (on IDevice instance)** | **Description** |
| IContexRequest | CreateRequest() | Device initialization, communication parameters settings, locks |
| IAuthRequest | CreateAuthRequest() | Authentication: logon, logoff |
| IMemRequest | CreateMemRequest() | Reading and writing virtual and physical memory, locking and unlocking physical memory, querying virtual memory pages information. |
| IVirtMemRequest | CreateVirtMemRequest() | A more specialized variant for reading and writing virtual memory; compatible with application layer, thus the Client (IContext) may be used as a virtual memory provider for application layer. |
| IFileSystemRequest | CreateFileSystemRequest() | Working with compact flash files. |
| ISerialRequest | CreateSerialRequest() | Management of serial port |
| IDnpRequest | CreateDnpRequest() | Direct execution of DNP requests for objects other than virtual memory (DNP object group 102) and virtual memory objects not covered by other request function groups (authentication, file system, virtual memory pages info, maintenance – restart and download). |
| IMaintenaceRequest | CreateMaintenaceRequest() | Device info, device restart, updating firmware, setting device time. |

The requests in a particular functional group may be executed if the physical device supports corresponding functionality. For example, IAuthRequest requests may be executed if the connected device is an CCP MCU device which supports security (such as IntelliRupter), IFileSystemRequest requests may be executed if the connected device is a CCP device supporting compact flash file system, IDnpRequest requests may be executed if the communication protocol used to connect to the device is DNP, etc.

Which functional groups are supported may be determined by the information in IDeviceInfo structure which is accessible as Info property of IDevice.

All requests above are based on ICpsRequest and their synchronous variant of operations and Get{Operation }Result methods of asynchronous variants return instances of CpsError or CpsRequestResult<T> classes that can be used for analyzing errors or getting operation results. An instance of CpsError is returned if operation does not return result. An instance of CpsRequestResult<T> is returned if operation returns result having type T.

CpsError is a class based on WkError class and it provides the following properties:

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| Source | string | always equals “CPS” |
| **CpsResult** | **CpsResult** | **result of CPS operation** |
| ErrorCode | int | result of CPS operation is conveted to int |
| Message | string | error description |
| InnerError | WkError | information for inner/underlying error |
| InnerException | Exception | an occurred exception for cases when CpsResult is UnhandledException or IOException. |

Own properties of CpsError class are marked as bold.

CpsRequestResutl<T> is a generic class based on WkResult<T> class and it provides the following properties:

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| **CpsError** | **CpsError** | **error occurred in a CPS operation processing or null if a CPS operation is completed successfully** |
| Error | WkError | the same as CpsError but provides less information |
| Result | T | result of successful CPS operation |

Own properties of CpsRequestResutl<T>class are marked as bold.

Also ICpsRequest has Status property that can be used for analyzing the results. When a request is completed with an error this property represents extended information about the error:

public class Status

{

public CpsResult CpsResult { get; set; } /// Result of CPS operation.

public int ExtResult { get; set; } /// Extended error code.

public int InnerExeption { get; set; } /// Occurred exception.

public string Description { get; set; } /// Error description (for user messages).

public string VerboseDescription { get; set; } /// Extended error description (for logging).

}

The CpsResut value is *Success* when a request completed successfully, and is *Aborted* when a request was aborted, otherwise it means an error. The ExtResult property extends(specializes) the CpsResult property. For example: if CpsResult equals *FileOperationError* then ExtResult contains CF error code; or if CpsResult equals CommEngineError then ExtResult contains IcpError for ICP device or IINFields for DNP device.

### 4.6.1 Device context management

All functions described in this part are not specific for a definite device subsystem. These functions allow sharing the access to one physical device for several clients (IDevice instances).

##### Init

After creation the IDevice instance must be initialized. Initialization must be done before performing any other request (must be the first request). This request connects a client to WCF server and gives parameters for communication with a device to the server and waits when connection to the device is established.

Synchronous variant:

CpsError Init(IProgress<long> progressHandler = null);

Asynchronous variant:

void InitAsync(IProgress<long> progressHandler = null);

CpsError GetInitResult();

Typical errors (CpsResult):

|  |  |
| --- | --- |
| InvalidParams | The communication parameters that were set in CreateDevice method are invalid. |
| UnavailableProcessor | Connected device has no processors that were set in communication parameters (TargetProcessor). |
| OperationIsBlocked | Changing of some connection parameters is blocked by other user. |
| Timeout | The device does not answer (a protocol operation is not finished for the time that was set). |
| WcfError | WCF error occured. Convert the ExtResult property to Wcf.WcfErrors enum for details. |

**Example:**

IDeviceRequest request = mcu.CreateRequest();

CpsError error = request.Init();

if (error = null)

{

Console.WriteLine("Initialization failed: {0}", error.Message);

}

reques.Close();

Sometimes connection to device can be lost during communications. In this case IsInitialized property of the IDevice will be false and initialization process must be repeated.

***GetCommParameters***

Provides actual connection parameters. This function can be used if a part of communication parameters is set atomatically. For example, when we connect to serial port with auto baud rate this function provides actual baud rate value.

Synchronous variant:

CpsRequestResult<IBaseCommParameters> GetCommParameters(

IProgress<long> progressHandler = null);

Asynchronous variant:

void GetCommParametersAsync(IProgress<long> progressHandler = null);

CpsRequestResult<IBaseCommParameters> GetCommParametersResult();

##### Release

This command allows ending the work with IDevice instance and releasing the following internal device states:

* Security session;
* FileSystem session;
* PhisycalMemory access.

Synchronous variant:

CpsError Release(IProgress<long> progressHandler = null);

Asynchronous variant:

void ReleaseAsync(IProgress<long> progressHandler = null);

CpsError GetReleaseResult();

Typical errors (CpsError):

|  |  |
| --- | --- |
| Timeout | The device does not answer (a protocol operation is not finished for the time that was set). |
| WcfError | WCF error occured. Convert the ExtResult property to Wcf.WcfErrors enum for details. |

**Example:**

IDeviceRequest request = mcu.CreateRequest();

CpsError error = request.Release();

if (error = null)

{

Console.WriteLine("Releasing failed: {0}", request.Status);

}

reques.Close();

The implementation of Close(Dispose) method of the IDevice is based on the synchronous variant of this function.

##### Lock

This function can be used for blocking a part of CPS functions for other users (IDevice instances). The available locks are represented in UserLock class:

|  |  |
| --- | --- |
| **Lock constant** | **Lock description** |
| ExclusiveLock | Locks all operations for all users |
| MaintenanceLock | Locks resetting device and writing physical memory. |
| VmWriteLock | Locks writing to virtual memory. |
| SetpointLock | Locks writing to setpoint memory. TBD |
| FileSystemLock | Locks all file system operations. |
| FileSystemWriteLock | Locks file system operations that are not read-only. |
| SerialLock | Locks serial port manipulation. |
| MemoryLock | Locks any operations with physical memory. |
| MemoryWriteLock | Locks writing to physical memory. |
| MonitoringLock | Locks resetting the device to maintenance mode. |

If a requested lock is taken by another user the method Lock will be completed only when its user releases this lock.

Synchronous variants:

CpsError Lock(string lockname);

CpsError Lock(IEnumerable<string> lockNames);

In the second case all locks can be taken as one.

Asynchronous variants:

void LockAsync(string lockname);

void LockAsync(IEnumerable<string> lockNames);

CpsError GetLockResult();

**Example:**

IDeviceRequest request = mcu.CreateRequest();

request.LockAsync(UserLock.MaintenanceLock);

if (!request. AsyncWaitHandle.WaitOne(1000))

{

Console.WriteLine("Taking MaintenanceLock failed");

}

##### Unlock

This function releases a taken lock.

Synchronous variant:

CpsError Unlock(string lockname);

Asynchronous variant:

void UnlockAsync(string lockname);

CpsError GetUnlockResult();

**Example:**

IDeviceRequest request = mcu.CreateRequest();

request.Unlock (UserLock.MaintenanceLock);

request.Unlock (UserLock.FileSystemLock);

### 4.6.2 Authentication functions

Some devices support a feature of user authentication (i.e., the IntelliRupter application). Whether a device supports authentication may be queried by the Capabilities property of Device Info.

|  |  |  |
| --- | --- | --- |
| **Protocol** | **TargetProcessor** | **Authentication supported** |
| DNP | MCU | Yes (for most S&C devices) |
| DNP | DSP (via MCU) | Yes (for most S&C devices) |
| ICP | MCU | Yes (for most S&C devices) |
| ICP | DSP | No |

IsLoggenOn property of IDevice represents the current state of authentication.

Logon and Logoff to/from the device are performed by LogOn and LogOff methods of IAuthRequest.

|  |  |
| --- | --- |
| **Function** | **Description** |
| LogOn | Logon to device using supplied user name and password. |
| LogOff | Logoff from device |

To create IAuthRequest instance you should use CreateAuthRequest method of IDevice.

var request = device. CreateAuthRequest ();

if (request == null)

{

Console.WriteLine("Auth not supported.");

return;

}

If some request after completion has CpsError.Auth status, this means that authentication is expired or not set and LogOn procedure is needed.

##### LogOn

It is used for the current user to get security session with a device using supplied user name and password.

Synchronous variant:

CpsError LogOn(

string userName,

string password,

IProgress<long> progressHandler = null);

Asynchronous variant:

void LogOnAsync(

string userName,

string password,

IProgress<long> progressHandler = null);

CpsError GetLogOnResult();

The typical errors (CpsError):

|  |  |
| --- | --- |
| AuthError | Incorrect user name or password. |
| AnotherAuthSession | Another user LoggedOn. |
| AuthNotReady | The auth system is not ready to LogOn (controller Appication is still being initialized) |
| AuthDoesNotEstablished | No session has been established (user has viewer privileges only). |
| AlreadyLoggedOn | Another user already got the authentication session and this session is shared and can be used. |
| UnsupportedOperation | The device does not support authentication. |

**Example:**

if (device.LogOnRequired())

{

var authReq = device.CreateAuthRequest();

CpsError error = authReq.LogOn("admin", password);

if (error != null)

{

Console.WriteLine("Cann't log on to device: " + error.Message);

}

else

{

Console.WriteLine("Auth completed.");

}

authReq.Close();

}

else

{

Console.WriteLine("Auth is not needed. ");

}

##### LogOff

Releases current security session.

Synchronous variant:

CpsError LogOff(IProgress<long> progressHandler = null);

Asynchronous variant:

void LogOffAsync(IProgress<long> progressHandler = null);

CpsError GetLogOffResult();

**Example:**

if (device.IsLoggenOn)

{

var authReq = device.CreateAuthRequest();

authReq.LogOff();

authReq.Close();

}

### 4.6.3 Memory Manipulation Functions

The following functions from IMemRequest interface and IVirtMemRequest can be used for reading or writing raw data from a device:

|  |  |
| --- | --- |
| **Function** | **Description** |
| UnlockMem | Unlocks specified memory class with specified password. |
| ReadMem | Reads specified ranges of specified memory class. |
| WriteMem | Writes specified ranges of specified memory class. |
| VirtMemPagesInfo | Gets information about virtual memory pages. |
| ReadVirtMem | Reads specified blocks of virtual memory. |
| WriteVirtMem | Writes specified blocks of virtual memory. |

The CPS provides access to memory classes that are equal to ICP memory selectors. These memory classes are defined in MemoryClass enum.

public enum MemoryClass : int

{

VirtualMemoryMainApplication,

VirtualMemoryOS,

ProgramStorageMainApplication,

ProgramStorageOS,

PhysicalMemory,

}

##### UnlockMem

A specified device memory can be protected from public using. This method allows unlocking such memory (usually physical memory) using specified password.

Synchronous variant:

CpsError UnlockMem(

MemoryClass mclass,

string password,

IProgress<long> progressHandler = null);

Asynchronous variant:

void UnlockMemAsync(

MemoryClass mclass,

string password,

IProgress<long> progressHandler = null);

CpsError GetUnlockMemResult();

Typical errors (CpsError):

|  |  |
| --- | --- |
| AuthError | Authentification is required. |
| UnsupportedMemory | The device (or protocol) does not support given memory class. |

**Example:**

IMemRequest mem\_req = mcu.CreateMemRequest();

CpsError error = mem\_req.UnlockMem(MemoryClass.PhysicalMemory, password);

if (error != null)

{

Console.WriteLine("Unlock phys memory failed: {0}", error.Message);

}

##### ReadMem

Reads specified ranges of specified memory class and reports progress.

Synchronous variant:

CpsError ReadMem(

MemoryClass mclass,

IEnumerable<VirtMemBlock> blocks,

IProgress<long> progressHandler = null);

Asynchronous variant:

void ReadMemAsync(

MemoryClass mclass,

IEnumerable<VirtMemBlock> blocks,

IProgress<long> progressHandler = null);

CpsError GetReadMemResult();

Typical errors (CpsError):

|  |  |
| --- | --- |
| MemoryLocked | The required memory is locked. |
| UnsupportedMemory | The device (or protocol) does not support given memory class. |
| AuthError | Authentification is required. |

**Example:**

VirtMemBlock phisblock = new VirtMemBlock() {

Address = 0x199200,

Length = 0x4,

Buffer = new byte[0x80]

};

IMemRequest mem\_req = mcu.CreateMemRequest();

CpsError error = mem\_req.UnlockMem(MemoryClass.PhysicalMemory, password);

if (error != null)

{

Console.WriteLine("Unlock phys memory failed: {0}", error.Message);

}

error = mem\_req.ReadMem(MemoryClass.PhysicalMemory, new VirtMemBlock[] { phisblock });

if (error != null)

{

Console.WriteLine("ReadMem failed: {0}", error.Message);

}

mem\_req.Close();

##### WriteMem

Reads specified ranges of specified memory class and reports progress.

Synchronous variant:

CpsError WriteMem(

MemoryClass mclass,

IEnumerable<VirtMemBlock> blocks,

IProgress<long> progressHandler = null);

Asynchronous variant:

void WriteAsync(

MemoryClass mclass,

IEnumerable<VirtMemBlock> blocks,

IProgress<long> progressHandler = null);

CpsError GetWriteResult();

Typical errors (CpsError):

|  |  |
| --- | --- |
| MemoryLocked | The required memory is locked. |
| UnsupportedMemory | The device (or protocol) does not support given memory class. |
| AuthError | Authentification is required. |

**Example:**

VirtMemBlock phisblock = new VirtMemBlock() {

Address = 0x199200,

Length = 0x4,

Buffer = new byte[0x80] { 0x34, 0x33, 0, 0x05 }

};

IMemRequest mem\_req = mcu.CreateMemRequest();

CpsError error = mem\_req.UnlockMem(MemoryClass.PhysicalMemory, password);

if (error != null)

{

Console.WriteLine("Unlock phys memory failed: {0}", error.Message);

}

error = mem\_req.WriteMem(MemoryClass.PhysicalMemory, new VirtMemBlock[] { phisblock });

if (error != null)

{

Console.WriteLine("WriteMem failed: {0}", error.Message);

}

mem\_req.Close();

##### VirtMemPagesInfo

Gets information about virtual memory pages. This function can be used for getting full virtual memory snapshot.

Synchronous variant:

CpsRequestResult<ICollection<VirtMemPageInfo> VirtMemPagesInfo(

PagesSelector selector = PagesSelector.All,

IProgress<long> progressHandler = null);

Asynchronous variant:

void VirtMemPagesInfoAsync(

PagesSelector selector = PagesSelector.All,

IProgress<long> progressHandler = null);

CpsRequestResult<ICollection<VirtMemPageInfo> GetVirtMemPagesInfoResult();

PageSelector is enum:

public enum PagesSelector

{

SmallPages,

BigPages,

SpecialRanges,

All

}

And VirtMemPageInfo is a class based on CCP 0x260-26F:

public class VirtMemPageInfo

{

public VirtMemPageType Type { get; set; }

public uint Address { get; set; }

public int Size { get; set; }

public byte Attributes { get; set; }

public byte Class { get; set; }

}

The special ranges can be read or write using ReadVirtMem or WriteVirtMem.

##### ReadVirtMem

Reads specified ranges of virtual memory and reports progress.

int ReadVirtMem(IEnumerable<VirtMemBlock> virtMem, IProgress<long> progress = null);

bool ReadVirtMem(IEnumerable<VirtMemBlock> virtMem, IProgress<long> progress, out WkError error);

This functions is equal to:

ReadMem(MemoryClass.VirtualMemoryMainApplication, blocks, progress)

**Example:**

VirtMemBlock block = new VirtMemBlock() {

Address = 0x200,

Length = 0x6,

Buffer = new byte[0x6]

};

IVirtMemRequest vmrequest = mcu.CreateVirtMemRequest();

if (vmrequest.ReadVirtMem(new VirtMemBlock[] { block }) != (int)CpsResult.Success)

{

Console.WriteLine("ReadVirtMem failed: {0}", vmrequest.Status);

}

vmrequest.Close();

##### WriteVirtMem

Writes specified ranges of virtual memory and reports progress.

int WriteVirtMem(IEnumerable<VirtMemBlock> virtMem, IProgress<long> progress = null);

bool WriteVirtMem(IEnumerable<VirtMemBlock> virtMem, IProgress<long> progress, out WkError error);

This function is equal to:

WriteMem(MemoryClass.VirtualMemoryMainApplication, blocks, progress)

Typical errors (CpsError):

|  |  |
| --- | --- |
| AuthError | Authentification is required. |

**Example:**

VirtMemBlock block = new VirtMemBlock() {

Address = 0x25200,

Length = 0x4,

Buffer = new byte[0x80] { 0x34, 0x33, 0, 0x05 }

};

IMemRequest vmrequest = mcu.GetMemRequest();

if (vmrequest.WriteVirtMem(new VirtMemBlock[] { block }) != (int)CpsResult.Success)

{

Console.WriteLine("WriteVirtMem failed: {0}", vmrequest.Status);

}

vmrequest.Close();

### 4.6.4 File System functions

Some devices have internal file system (i.e., the IntelliRupter). Whether a device supports a file system may be queried by the Info.Capabilities property of the IDevice instance.

Now CPS provides the following functions for working with a file system:

|  |  |
| --- | --- |
| **Function** | **Description** |
| CreateDirectory | Creates directory |
| DeleteDirectory | Deletes directory |
| DeleteFile | Deletes file |
| ReadFileSystemInfo | Provides information about file system |
| FileInfo | Provides information about a file |
| ReadDirectory | Reads the list of files in particular directory |
| Format | Formats a file system |
| ReadFile | Reads the entire file |
| Rename | Renames file or directory |
| WriteFile | Re-writes or creates new file |

To create IFileSystemRequest request you should use CreateFileSystemRequest method of IDevice

var request = device. CreateFileSystemRequest();

if (request == null)

{

Console.WriteLine("File system not supported.");

return;

}

##### ReadFileSystemInfo

Provides information about file system.

Synchronous variant:

CpsRequestResult<FileSystemInfo> ReadFileSystemInfo(

IProgress<long> progressHandler = null);

Asynchronous variant:

void ReadFileSystemInfoAsync(IProgress<long> progressHandler = null);

CpsRequestResult<FileSystemInfo> GetReadFileSystemInfoResult();

Where FileSystemInfo is a class providing the following members:

|  |  |
| --- | --- |
| **Property** | **Description** |
| Serial | Disk name |
| BlockCount | Disk size in blocks. |
| Size | Disk size in bytes. |
| AllocatedBlocks | Number of used blocks. |
| AllocatedBytes | Number of used blocks |
| FreeBlocks | Number of free blocks. |
| FreeBytes | Number of free bytes. |
| BlockSize | Number bytes in one block. |

Typical errors (CpsError):

|  |  |
| --- | --- |
| FileSystemBusy | Another file session is in progress. |
| FileOperationError | FileSystem operation reports an error. See the ExtResult property for more information. |
| UnsupportedOperation | The device does not support file system operations. |

**Example:**

CpsRequestResult<FileSystemInfo> result = request.ReadFileSystemInfo();

if (result.Error != null)

{

Console.WriteLine("Could not read CF info: {0}", result.Error.Message);

}

else

{

Console.WriteLine("Drive Name: " + result.Result.Serial);

}

##### FileInfo

Provides information about a file or directory.

Synchronous variant:

CpsRequestResult<FileInfo> FileInfo(

string fileName,

IProgress<long> progressHandler = null);

Asynchronous variant:

void FileInfoAsync(

string fileName,

IProgress<long> progressHandler = null);

CpsRequestResult<FileInfo> GetFileInfoResult();

Where FileInfo is a class providing the following properties:

|  |  |
| --- | --- |
| **Property** | **Description** |
| Name | Name of a file or a directory. |
| Size | File size |
| Attributes | Attributes of file or a directory (ReadOnly, Hidden, System, Directory Archive) |
| CreatedTime | Date/time of creation. |
| ModifiedTime | Date/time of modification. |

Typical errors (CpsError):

|  |  |
| --- | --- |
| InvalidParams | Given fileName is empty or incorrect. |
| FileSystemBusy | Another file session is in progress. |
| FileOperationError | FileSystem operation reports an error. See the ExtResult property for more information. |
| UnsupportedOperation | The device does not support file system operations. |

**Example:**

CpsRequestResult<FileInfo> result = request.FileInfo("/info.txt");

if (result.Error != null)

{

Console.WriteLine("Could not read file info: {0}", result.Error.Message);

}

else

{

Console.WriteLine("File size: " + result.Reslut.Size);

}

##### ReadDirectory

Reads the list of files in particular directory.

Synchronous variant:

CpsRequestResult<IEnumerable<FileInfo>> ReadDirectory(

string path,

IProgress<long> progressHandler = null);

Asynchronous variant:

void ReadDirectory Async(

string path,

IProgress<long> progressHandler = null);

CpsRequestResult<IEnumerable<FileInfo>> GetReadDirectoryResult();

Typical errors (CpsError):

|  |  |
| --- | --- |
| InvalidParams | Given dirName is empty or incorrect. |
| FileSystemBusy | Another file session is in progress. |
| FileOperationError | FileSystem operation reports an error. See the ExtResult property for more information. |
| UnsupportedOperation | The device does not support file system operations. |

**Example:**

Console.WriteLine("FilesList: /HISTLOG");

CpsRequestResult<IEnumerable<FileSystemInfo>> result = request.ReadDirectory("/HISTLOG");

if (result.Error != null)

{

Console.WriteLine("Reading directory failed: {0}", result.Error.Message);

}

else

{

foreach (var finfo in result.Result)

{

Console.WriteLine("\t" + finfo.Name);

}

}

##### CreateDirectory

Creates a new directory.

Synchronous variant:

CpsError CreateDirectory(

string newDirectoryPath,

IProgress<long> progressHandler = null);

Asynchronous variant:

void CreateDirectoryAsync(

string newDirectoryPath,

IProgress<long> progressHandler = null);

CpsError GetCreateDirectoryResult();

Typical errors (CpsError):

|  |  |
| --- | --- |
| InvalidParams | Given dirName is empty or incorrect. |
| FileSystemBusy | Another file session is in progress. |
| FileOperationError | FileSystem operation reports an error. See the ExtResult property for more information. |
| UnsupportedOperation | The device does not support file system operations. |
| AuthError | Authentification is required. |

**Example:**

CpsError error = request.CreateDirectory("/TestDir");

if (error != null)

{

Console.WriteLine("Could not create directory: {0}", error.Message);

}

##### DeleteDirectory

Deletes an existing directory (for CCP: directory must be empty).

Synchronous variant:

CpsError DeleteDirectory(

string deletedDirectoryPath,

IProgress<long> progressHandler = null);

Asynchronous variant:

void DeleteDirectoryAsync(

string deletedDirectoryPath,

IProgress<long> progressHandler = null);

CpsError GetDeleteDirectoryResult();

Typical errors (CpsError):

|  |  |
| --- | --- |
| InvalidParams | Given dirName is empty or incorrect. |
| FileSystemBusy | Another file session is in progress. |
| FileOperationError | FileSystem operation reports an error. See the ExtResult property for more information. |
| UnsupportedOperation | The device does not support file system operations. |
| AuthError | Authentification is required. |

**Example:**

CpsError error = request.DeleteDirectory("/TestDir");

if (error != null)

{

Console.WriteLine("Could not delete directory: {0}", error.Message);

}

##### Rename

Changes the name of a file or directory.

Synchronous variant:

CpsError Rename(

string oldFileName,

string newFileName,

IProgress<long> progressHandler = null);

Asynchronous variant:

void RenameAsync(

string oldFileName,

string newFileName,

IProgress<long> progressHandler = null);

CpsError GetRenameResult();

Typical errors (CpsError):

|  |  |
| --- | --- |
| InvalidParams | Given oldFileName or newFileName is empty or incorrect. |
| FileSystemBusy | Another file session is in progress. |
| FileOperationError | FileSystem operation reports an error. See the ExtResult property for more information. |
| UnsupportedOperation | The device does not support file system operations. |
| AuthError | Authentification is required. |

**Example:**

CpsError error = request.Rename("/info.txt", "info1.txt");

if (error != null)

{

Console.WriteLine("Could not rename file : {0}", error.Message);

}

##### DeleteFile

Deletes an existing file.

Synchronous variant:

CpsError DeleteFile(

string deletedFilePath,

IProgress<long> progressHandler = null);

Asynchronous variant:

void DeleteFileAsync(

string deletedFilePath,

IProgress<long> progressHandler = null);

CpsError GetDeleteFileResult();

Typical errors (CpsError):

|  |  |
| --- | --- |
| InvalidParams | Given fileName is empty or incorrect. |
| FileSystemBusy | Another file session is in progress. |
| FileOperationError | FileSystem operation reports an error. See the ExtResult property for more information. |
| UnsupportedOperation | The device does not support file system operations. |
| AuthError | Authentification is required. |

**Example:**

CpsError error = request.DeleteFile ("/info.txt");

if (error != null)

{

Console.WriteLine("Could not delete file: {0}", error.Message);

}

##### Format

Formats a file system (erases all files and directories).

Synchronous variant:

CpsError Format(IProgress<long> progressHandler = null);

Asynchronous variant:

void FormatAsync(IProgress<long> progressHandler = null);

CpsError GetFormatResult();

*Note: for a CCP device in application mode the device will be reset to the maintenance mode, after resetting, CF will be formatted and the device will be reset back to the application mode.*

Typical errors (CpsError):

|  |  |
| --- | --- |
| InvalidParams | Given fileName is empty or incorrect. |
| FileSystemBusy | Another file session is in progress. |
| FileOperationError | FileSystem operation reports an error. See the ExtResult property for more information. |
| UnsupportedOperation | The device does not support file system operations. |
| AuthError | Authentification is required. |

**Example:**

CpsError error = request.Format();

if (error != null)

{

Console.WriteLine("Could not format CF: {0}", error.Message);

}

##### ReadFile

Reads the entire file.

Synchronous variant:

CpsRequestResult<long> ReadFile(

string fileName,

Stream outStream,

IProgress<long> progressHandler = null);

Asynchronous variant:

void ReadFileAsync(

string fileName,

Stream outStream,

IProgress<long> progressHandler = null);

CpsRequestResult<long> GetReadFileResult();

The returned result (long) means the number of bytes read.

Typical errors (CpsError):

|  |  |
| --- | --- |
| InvalidParams | Given fileName is empty or incorrect. |
| FileSystemBusy | Another file session is in progress. |
| FileOperationError | FileSystem operation reports an error. See the ExtResult property for more information. |
| UnsupportedOperation | The device does not support file system operations. |

**Example:**

Programm.WriteLine("ReadFile: '/HISTLOG/EVTLOG00.EVT' ");

var outStream = new MemoryStream();

CpsRequestResult<long> result = request.ReadFile("/HISTLOG/EVTLOG00.EVT", outStream);

if (result.Error != null)

{

Console.WriteLine("Could not read file: {0}", result.Error.Message);

}

else

{

Console.WriteLine("Read bytes: {0}", result.Result);

}

##### WriteFile

Creates a new file or re-writes the existing file.

Synchronous variant:

CpsRequestResult<long> WriteFile(

string fileName,

Stream inStream,

IProgress<long> progressHandler = null);

Asynchronous variant:

void WriteFileAsync(

string fileName,

Stream inStream,

IProgress<long> progressHandler = null);

CpsRequestResult<long> GetWriteFileResult();

The returned result (long) means the number of bytes written.

Typical errors (CpsError):

|  |  |
| --- | --- |
| InvalidParams | Given fileName is empty or incorrect. |
| FileSystemBusy | Another file session is in progress. |
| FileOperationError | FileSystem operation reports an error. See the ExtResult property for more information. |
| UnsupportedOperation | The device does not support file system operations. |
| AuthError | Authentification is required. |

**Example:**

Programm.WriteLine("ReadFile: '/log.txt' ");

var inStream = System.IO.File.Open("text.out", FileMode.Open)

CpsRequestResult<long> result = request.WriteFile("/log.txt", inStream);

if (result.Error != null)

{

Console.WriteLine("Could not write file: {0}", result.Error.Message);

}

else

{

Console.WriteLine("Written bytes: {0}", result.Result);

}

### 4.6.5 Maintenance functions

The functions in this part allow managing the device:

|  |  |
| --- | --- |
| **Function** | **Description** |
| ReadDeviceInfo | Reads specified subset of device information. |
| SetTime | Sets device time. |
| ResetDevice | Restarts some device processors to given mode (application, maintenance). |
| UpdateFirmware | Downloads new firmware to device. |

To create IMaintenanceRequest instance you should use CreateMaintenanceRequest method of IDevice.

##### ReadInfo

Reads specified subset of device information.

Synchronous variant:

CpsRequestResult<IDeviceInfo> ReadDeviceInfo(

DeviceInfoOptions options = DeviceInfoOptions.GeneralInfo,

IProgress<long> progressHandler = null);

Asynchronous variant:

void ReadDeviceInfoAsync(

DeviceInfoOptions options = DeviceInfoOptions.General,

IProgress<long> progressHandler = null);

CpsRequestResult<IDeviceInfo> GetReadDeviceInfoResult();

Where returned IDeviceInfo instance provides the following properties:

|  |  |
| --- | --- |
| Property | Description |
| Processor | current(connected) processor. |
| Series | device series (e.g., Series\_2XX, Series\_Spirit\_MCU, etc.) |
| OSVersion | EOS version |
| Mode | represents device mode (application or maintenance). |
| Capabilities | Capabilities of the device. This is a bit mask containing the following:   * ICP * DNP * Application defined protocol * Compact Flash * Authentication * Multiple processors |
| AuthVersion | authentication version supported by application |
| LastResetStatus | information from 0x200-3 addresses |
| CreationTime | timestamp -when this info was updated last time |
| AppIdent | application ident. |
| AppVersion | application version. |
| HmiVersion | computed from AppVersion according to CCP Platform conventions. |
|  |  |
| Processors | information about all processors of the device |
| SerialPorts | infomation about serial ports. |
| InternalEthernetController | internal ethernet controller settings. |
| ExternalEthernetController | external ethernet controller settings. |
| TimeInfo | time information |

And DeviceInfoOptions specifies reading information:

|  |  |
| --- | --- |
| Option | Description |
| GeneralInfo | General information is marked as red above. It is read always. |
| TimeInfo | Time information will be read. |
| SerialPorts | Information about serial ports will be read. |
| EthernetControllers | Information about Ethernet/IP settings will be read. |
| AllProcessors | The selected options will be read from all processors. |
| All | Full information for all processors will be read from device. |

**Example:**

IMaintenanceRequest request = mcu.GetMaintenanceRequest();

CpsRequestResul<IDeviceInfo> result =

request.ReadDeviceInfo(DeviceInfoOptions.EthernetControllers);

if (result.Error != null)

{

Console.WriteLine("Could not read device info: {0}", result.Error.Message);

}

else

{

Console.WriteLine("IP1: {0}, IP2: {1}",

result.Result.InternalEthernetController.Ip,

result.Result.ExternalEthernetController.Ip);

}

request.Close()

##### SetTime

Sets device time.

Synchronous variant:

CpsError SetTime(

DateTime newTime,

IProgress<long> progressHandler = null);

Asynchronous variant:

void SetTimeAsync(

DateTime newTime,

IProgress<long> progressHandler = null);

CpsError GetSetTimeResult();

Typical errors (CpsError):

|  |  |
| --- | --- |
| UnsupportedOperation | The device does not support this operation. |
| AuthError | Authentification is required. |

**Example:**

IMaintenanceRequest request = mcu.GetMaintenanceRequest();

CpsError error = request.SetTime(DateTime.Now);

if (error!= null)

{

Console.WriteLine("Could not set time: {0}", error.Message);

}

reqest.Close();

##### ResetDevice

Restarts some device processors to given mode (application, maintenance).

Synchronous variant:

CpsError ResetDevice(

ProcessorMode mode,

ProcessorsMask processors = ProcessorsMask.CurrentProcessor,

IProgress<long> progressHandler = null);

CpsError ResetDevice (

ProcessorsMask application = ProcessorsMask.CurrentProcessor,

ProcessorsMask maitenance = ProcessorsMask.None,

IProgress<long> progressHandler = null);

Asynchronous variant:

void ResetDeviceAsync(

ProcessorMode mode,

ProcessorsMask processors = ProcessorsMask.CurrentProcessor,

IProgress<long> progressHandler = null);

void ResetDeviceAsync(

ProcessorsMask application = ProcessorsMask.CurrentProcessor,

ProcessorsMask maitenance = ProcessorsMask.None,

IProgress<long> progressHandler = null);

CpsError GetResetDeviceResult();

Typical errors (CpsError):

|  |  |
| --- | --- |
| ResetDeviceError | The device was reset with errors. For more information see LastResetStatus. |
| AuthError | Authentification is required. |
| Timeout | The device does not answer (a protocol operation is not finished for the time that was set). |

**Example:**

IMaintenanceRequest request = mcu.GetMaintenanceRequest();

CpsError error = request.ResetDevice(ProcessorMode.Application, ProcessorsMask.AllProcessors);

if (error!= null)

{

Console.WriteLine("Device reset with error: {0}", error.Message);

}

reqest.Close();

##### UpdateFirmware

Updates device (processors) firmware.

Synchronous variant:

CpsError UpdateFirmware(

Stream codFile,

UpdateFirmwareOptions options = UpdateFirmwareOptions.None,

IProgress<long> progressHandler = null);

Asynchronous variant:

void UpdateFirmwareAsync(

Stream codFile,

UpdateFirmwareOptions options = UpdateFirmwareOptions.None,

IProgress<long> progressHandler = null);

CpsError GetUpdateFirmwareResult();

And UpdateFirmwareOptions specifies reading information:

|  |  |
| --- | --- |
| Option | Description |
| None | CPS default setting will be used. |
| UseBottleneck | Legacy interface for access to phys memory over virtual memory will be used. (Only for CPP devices connected over DNP.) |
| UsePhysMemMapping | Direct access to phys memory over virtual memory will be used. (Only for CPP devices connected over DNP.) |
| CheckBottleneck | Check address and checksum after any bottleneck operation. |
| NotResetProcessorAfterUpdate | Processor will not be reset automatically after writing firmware. |

For CCP the following operations are done:

* processor is reset to maintenance mode, if it is needed;
* phys memory is locked, if it is needed;
* MCU/DSP bus timings are set longer, if download is to DSP;
* Writing given phys memory block via bottleneck or via VM mapping;
* MCU/DSP bus timings are set normal;
* phys memory is unlocked, if it was locked before;
* download counter is incremented;
* processor is reset to application mode, if NotResetProcessorAfterUpdate is not set;

Typical errors (CpsError):

|  |  |
| --- | --- |
| ResetDeviceError | The device was reset with errors. |
| DeviceReset | The device is reset unexpectedly. |
| AuthError | Authentification is required. |
| Timeout | The device does not answer (a protocol operation is not finished for the time that was set). |

**Example:**

IMaintenanceRequest request = mcu.GetMaintenanceRequest();

CpsError error = request.UpdateFirmware(File.OpenRead(codFile));

if (error!= null)

{

Console.WriteLine("Could not update device firmware: {0}", error.Message);

}

reqest.Close();

### 4.6.6 Serial port management

The functions in this part allow changing serial port parameters:

|  |  |
| --- | --- |
| **Function** | **Description** |
| ChangeBaudRate | Sets new baud rate on the serial port (used to connect to device). |

To create ISerialRequest instance you should use CreateSerialRequest method of IDevice.

### 4.6.7 Direct execution of DNP requests

The functions in this part allow sending DNP requests if the device is connected via DNP.

IDnpRequest function group provides the following

|  |  |
| --- | --- |
| **Method** | **Description** |
| ResetLink | Sends Reset DataLink request. |
| RequestLink | Determines DataLink status. |
| DoRequest | Issues a general DNP application level request. A DNP request consists of function code and a list of DNP object sets. |

To create IDnpRequest instance you should use CreateDnpRequest method of IDevice.

Note that a request will fail if it contains a write operation to a “magic” virtual memory location which is monitored by other request groups. Such addresses are 0x02B1, 0x02F0, 0x02F1 (TBD), 0x80-0x85.

## 4.7 Errors handling

The following types of errors may occur:

* Errors related to communications between a client and the server (e.g., the server was shutdown). In the case when a request is completed with CspError.WcfError status, you should check IsInitialized property of IDevice and if the property is false then you should reinitialize the connection using Init request.
* If some request after completion has CpsError.Auth status, it means that authentication is expired or not set and LogOn procedure is needed.
* Errors related to communications between the server and physical device. Errors of this type are reported as return codes or out parameters of corresponding methods.

Usage errors are caused by an incorrect sequence of method calls initiated by client application. For example, an attempt to make a new request on IAsyncRequest control block while another request is already being executed on that control block. Such errors are reported as exceptions (e.g., InvalidOperationException).

# Scripting API

For scripting in WK6 Micrisoft PowerShell was used. It is a free powerful and flexible script engine, which is pre-installed in all newest Microsoft OS and is easy for learning and does not demand special ([costly](http://lingvopro.abbyyonline.com/ru/Search/GlossaryItemExtraInfo?text=%d0%b4%d0%be%d1%80%d0%be%d0%b3%d0%be%d1%81%d1%82%d0%be%d1%8f%d1%89%d0%b8%d0%b9&translation=costly&srcLang=ru&destLang=en)) IDE. Also the PowerShell has native interface to .NET class libraries (both standard and custom).

For more information about PowerShell script writing see the following documents:

<http://msdn.microsoft.com/en-us/library/dd835506%28v=VS.85%29.aspx>

<http://technet.microsoft.com/en-us/library/cc196356.aspx>

<http://technet.microsoft.com/en-us/library/bb978526.aspx>

<http://technet.microsoft.com/en-us/library/ee177003.aspx>

## 5.1 Basic Information

WinKit 6 scripting includes WinKit 6 cmdlets and WinKit 6 objects. WinKit 6 scripting provides a binding of WinKit 6 functionality to Microsoft PowerShell, so that it can be used from within PS scripts.

For a script to utilize the WinKit 6 functionality, it should import the dll library SandC.WinKit6.Scripting.dll. This is done by the command of the form

Import-Module “.\SandC.WinKit6.Scripting.dll"

The particular path to the library (current working directory in this example) should be adjusted accordingly. For example, the following code loads this dll from installed WK6 HMI Developer Tools:

if ($Env:PROCESSOR\_ARCHITECTURE -eq "x86")

{

$HDTDIR = $Env:ProgramFiles;

}

else

{

$HDTDIR = ${Env:ProgramFiles(x86)}

}

Import-Module ($HDTDIR + "\S&C Electric\WinKit 6\SandC.WinKit6.Scripting.dll")

The user can write his/her own scripts and execute them using IntelliLink (menu Tools -> Play script…) or other PS host. More information about writing scripts can be found in chapter 5.2.

## 5.2 WinKit 6 Cmdlets and Objects

WinKit 6 cmdlets (see chapter 5.3) are used to instantiate WinKit 6 objects (see chapter 5.4) and also to perform some specific tasks (e.g., lengthy operations which provide progress indicator). WinKit 6 objects are .NET objects specifically tailored to be used in scripts. Essentially, they are the wrapper classes on the native WinKit 6 objects.

The cmdlets have parameters, which supply input values. The cmdlets usually return some WinKit 6 object as return value.

The WinKit 6 objects, just as ordinary .NET objects, have properties and methods which may be assigned, retrieved, or invoked in the scripts. They provide the needed functionality.

|  |  |  |  |
| --- | --- | --- | --- |
| **WinKit 6 cmdlets** | | | |
| **Communications/CPS cmdlets** | | | |
| **Cmdlet** | **Return object** | **Description** | **Progress bar** |
| Get-Device | DeviceWrapper | Creates Device object |  |
| Save-Memory | Operation Status | Saves memory snapshot from device to a given file. | yes |
| Update-Firmware | Operation Status | Downloads firmware into a given device from a given .cod file. | yes |
| **Application Layer cmdlets** | | | |
| Get-DataProvider | DataProviderWrapper | Creates application layer data provider. |  |
| Restore-Memory | Operation Status | Loads virtual memory content from a given file in the legacy xdt format into the given device. | yes |

|  |  |
| --- | --- |
| **WinKit 6 objects** | |
| **Object** | **Description** |
| DeviceWrapper | Provides access to a connected physical device. Performs most of the functions related to the connected device via properties and methods. |
| DataProviderWrapper | Provides application layer data provider, based on application profile and implementation, and also a snapshot file or device as data source. Performs most application-layer functions via properties and methods. |

Some examples of using cmdlets and objects can be found in the installed WK6 HMI Developer Tools.

## 5.3 Cmdlets

### 5.3.1. Get-Device

Description

This cmdlet creates a new device wrapper object. This object can be used to execute the CPS operations or for using in Get-DataProvider cmdlet (see 5.3.3.).

Command line syntax

**Get-Device** [-Proto DNP | ICP] [-LocalAddress <UInt16>] [-PeerAddress <UInt16>] [-Timeout <Int32>] [-Retries <Int32>]

Command line options

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **Proto** | communication protocol (DNP or ICP). Optional, default value is DNP. |
| **LocalAddress** | local (master) device address. Optional, default value is 0. (for DNP protocol only). |
| **PeerAddress** | remote (slave) device address. Optional, default value is 65532 for DNP and 8176 for ICP. |
| **Timeout** | sets the timeout for the communication operation. Optional, default values is 500. |
| **Retries** | sets the number of attempts to execute the operation. Optional, default values is 0. |

**Example:**

* $device = Get-Device -Proto "DNP" -LocalAddress 2 -PeerAddress 65532

$device.ConnectUdp("10.1.0.239", 20000) | HaltOnError

* $device = Get-Device -Proto "ICP" -PeerAddress 8176

$device.ConnectSerial("COM1") | HaltOnError

### 5.3.2. Get-DataProvider

Description

This cmdlet creates an application layer data provider. The provider can be used to read or write application object and to save or restore setpoints. It returns null when the data provider for the given cmdlet arguments cannot be created.

Command line syntax

1. **Get-DataProvider** -HmiDef <String> -ProfileName <String> -VMProvider <IVirtMemProvider>
2. **Get-DataProvider** -HmiDef <String> -ProfileName <String> -Snapshot <String>
3. **Get-DataProvider** -HmiDef <String> -ProfileName <String> -XHDFile <String>
4. **Get-DataProvider** -HmiDef <String> -ProfileName <String>
5. **Get-DataProvider** -HmiDef <String> -VMProvider <IVirtMemProvider>
6. **Get-DataProvider** -HmiDef <String> -Snapshot <String>
7. **Get-DataProvider** -HmiDef <String> -XHDFile <String>
8. **Get-DataProvider** -VMProvider <IVirtMemProvider>
9. **Get-DataProvider** -Snapshot <String>
10. **Get-DataProvider** -XHDFile <String>

Command line options

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **HmiDef** | defines a path to the HMI definition file (.hmidef). |
| **ProfileName** | defines a profile name. Alias is Prof. |
| **VMProvider** | refers to the object created using the **Get-Device** cmdlet (see 5.3,1,).Alias is **Device.** |
| **Snapshot** | defines a path to the virtual memory snapshot file (.vm). Alias is **VM**. |
| **XHDFile** | defines a path to the Application Level Snapshot file (.xdss). Alias is **XHD.** |

**Example:**

$dataProvider = Get-DataProvider -Snapshot "IR.vm"

### 5.3.3.Update-Firmware

Description

This cmdlet downloads new firmware to device and displays progress of this operation.

Command line syntax

**Update-Firmware** [-ProgressIndicatorHeader] <String> [-Device] <Device> [-CodeFile] <String>

Command line options

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **ProgressIndicatorHeader** | defines a text that is used when the operation progress is displayed. |
| **Device** | refers to the object created with the **Get-Device** cmdlet. |
| **FirmwareFile** | defines the firmware file (cod format) that will be loaded to the device. |

Returned value

A instance of Cps.Status class. The operation was successful when its CpsResult property is *Success*.

**Example:**

…

$res = Update-Firmware “DSP OS updating …” $device "OS\_code\_file"

if ($res.CpsResult -ne ‘Success’)

{

Write-Warning “Upgrading fails.”

Exit

}

### 5.3.4.Save-Memory

Description

This cmdlet saves the memory snapshot and displays progress of the operation.

Command line syntax

**Save-Memory** [-Device] <Device> [-FileToSaveVM] <String> [-WorkID] <Int32>

Command line options

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **Device** | refers to the object created with the **Get-Device**  cmdlet (see 5.3.1.). |
| **FileToSaveVM** | Defines the name of the file where the memory snapshot will be saved. |
| **WorkID** | Specifies the type of reading. Possible values are:  0 – reading all pages  1 – reading small pages  2 – reading large pages |

Returned value

A instance of Cps.Status class. The operation was successful when its CpsResult property is *Success*.

**Example:**

…

$res = Save-Memory $device "C:/tmp/tmp/tmp.vm"

if ($res.CpsResult -ne ‘Success’)

{

Write-Warning $res.Description

Exit

}

…

### 5.3.5. Restore-Memory

Description

This cmdlet loads virtual memory blocks from a XTD file to the device and displays progress of the operation.

Command line syntax

Restore-Memory [-Device] <Device> [-XdtFile] <String>

Command line options

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **Device** | refers to the object created with the **Get-Device** cmdlet (see 5.3.1.). |
| **XdtFile** | defines the file name where to load XDT data(virtual memory blocks) from. |

Returned value

A instance of Cps.Status class. The operation was successful when its CpsResult property is *Success*.

**Example:**

…

$res = Restore-Memory $device "C:/tmp/tmp/tmp.vm"

if ($res.CpsResult -ne ‘Success’)

{

Write-Warning $res.Description

Exit

}

…

## 5.4 Objects

The full detailed description of WinKit 6 object properties and methods may be found in Cps.Client.ClassLibrary.chm document; see SandC.WinKit6.Scripting namespace.

The full detailed description of WinKit 6 object properties and methods may be found in Cps.Client.ClassLibrary.chm document; see SandC.WinKit6.Scripting namespace.

### 5.4.1. DeviceWrapper

|  |  |
| --- | --- |
| **Method** | **Description** |
| **ConnectSerial** | connects to the device via **serial port**. |
| **ConnectTcp** | connects to the device via TCP/IP |
| **ConnectUdp** | connects to the device via UDP/IP |
| **Timeout** | gets or sets the communication timeout. |
| **Retries** | gets or sets the number of attempts to execute the operation. |
| **ChangeBaudRate** | changes the baud rate. (for Serial channel only) |
|  |  |
| **Close** | closes the device and releases all its resources. |
|  |  |
| **CreateDevice** | creates a new device wrapper for an available processor. |
|  |  |
| **CreateDirectory** | creates new directory. |
| **DeleteDirectory** | deletes the directory. |
| **DeleteFile** | deletes the file name from the CF file system. |
| **GetFile** | reads file from the device and saves it to the local file system. |
| **FileList** | gets files from the directory. |
| **Format** | formats the CF. |
| **PutFile** | reads the file from the local file system and writes it to the device. |
| **Rename** | renames the file. |
|  |  |
| **SetTime** | sets the current date. |
|  |  |
| **Logon** | performs logon to controller |
| **LoggedOn** | indicates whether the authentification has passed. |
|  |  |
| **ReadInfo** | reads additional device information. |
|  |  |
| **ResetLink** | resets the DNP Data Link after the timeout occurs. |
|  |  |
| **ResetDevice** | restarts one or several processors of the device to required mode (application, maintenance). |
| **UnlockMemory** | unlocks physical memory of the device . |
| **UpdateFirmware** | updates the firmware in the controller. |
|  |  |
| **ReadByte** | reads one byte of virtual memory from device. |
| **ReadVmBlock** | reads a number of bytes from the virtual memory. |
| **WriteByte** | writes one byte of virtual memory to device. |
| **WriteVmBlock** | writes the buffer of bytes to the virtual memory. The error code returns. |

|  |  |
| --- | --- |
| **Property** | **Description** |
| Status | status of the last executed operation. |
| Info | device information. |
| Mode | indicates device(device processor) mode (maintenance or application). |
| MaintenanceMode | indicates maintenance mode. |
| LogOnRequired | indicates whether LogOn method is required for non-readonly operation. |
| LoggedOn | indicates whether last LogOn method is completed successfully. |
| LocalAddress | allows to set local DNP address before executing Connect method. |
| PeerAddress | allows to set peer DNP or ICP address before executing Connect method. |
| Retries | allows to set timeout before executing Connect method. |
| Provider | gets Virtual Memory provider. |

**Status** contains the following properties:

|  |  |
| --- | --- |
| **Property** | **Description** |
| CpsResult | Last CPS operation result. See documentation on  *enum SandC.WinKit6.Communications.Cps.CpsResult* |
| Description | Error description. |
| ExtResult | Specializing result code. Its value depends on CpsResult and an executed method. |

**Info** contains the following properties:

|  |  |
| --- | --- |
| **Property** | **Description** |
| Series | the device series (Series\_2XX, CCP\_DSP, CCP\_MCU, etc.) |
| Processor | the type of communication processor (MCU, DSP0 – DSP7). |
| Capabilities | capabilities of device (compact flash, authorization, available communication protocols, etc.) |
| OSVersion | the EOS version (revision). |
| MaintenanceMode | indicates whether the controller is in maintenance mode |
| AppIdent | the application name (IDENT) [not available when in maintenance mode] |
| AppVersion | the application version(revision) [not available when in maintenance mode] |
| AuthVersion | the authentication protocol version [not available when in maintenance mode] |
| LastResetStatus | the status of the last device reset. For more information see CCP Addendum 0x200-0x204. |
| Processors | the list of available communication processors. Connection to a processor can be established using CreateDevice method. |

**Example:**

…

$device = Get-Device -LocalAddress $localAddress

$device.Timeout = 1000

$device.Retries = 2

$res = $device.ConnectUdp ($peerIpAddress, $peerIpPort)

if ($res –eq $false) { WriteErrorAndExit $res }

$res = $device.ResetDevice($false);

if ($res –eq $false) { WriteErrorAndExit $res }

$IRFirmwareDir = "C:\Program Files\S&C Electric\Products\IntelliRupter\Firmware\”

$eos = ($IRFirmwareDir + “MCUEOSLOADER\1.21.43.0\MCUEOSLOADER-1.21.43.0.cod")

$app = ($IRFirmwareDir + “\IRPMIT2D30HR1000\3.2.13.2\IRPMIT2D30HR1000-3.2.13.2.cod")

$res = $device.UpdateFirmware( eos, $true);

if ($res –eq $false) { WriteErrorAndExit $res }

$res = $device.UpdateFirmware($app, $false);

if ($res –eq $false) { WriteErrorAndExit $res }

### 5.4.2.DataProviderWrapper

|  |  |
| --- | --- |
| **ReadObject** | reads a single Application Object by the name. |
| **WriteObject** | writes a single Application Object by the name. |
| **WriteObjects** | writes a collection of instances of a class AppObject. |
| **LoadSetpoints** | loads setpoints from the file (XSPT format) and rerurns instance of AppDataConvertResult class which contains results of conversion. |
| **SaveSetpoints** | saves setpoints to a file (XSTP fromat). |

See below an example of using this method.

**Example:**

…

$dataProvider = Get-DataProvider –HmiDef $hmiDef -VMProvider $device

$res = $dataProvider.SaveSetpoints(“C:\Temp\SptFile.xspt”)

If ($res –eq $false)

{

Write-Error “Cannot save set points.”

Exit 1

}

…

$conversionResult = $dataProvider.LoadSetpoints”SptFile”)

if ($conversionResult.NotConverted.Count -ne 0 -or $conversionResult.NotFound.Count -ne 0)

{

$msg = “Converted AppObjects:” + $conversionResult.Converted.Count

Write-Host $msg

$msg = “Not Converted AppObjects:”” + $conversionResult.NotConverted.Count

Write-Host $msg

$msg = “Not Found AppObjects: “ + $conversionResult.NotFound.Count

Write-Host $msg

}

if (!$dataProvider.WriteObjects($conversionResult.Converted))

{

Write-Warning “Impossible Upload Setpoints”

$err = “Last error : ” + $dataProvider.Message

Write-Error $err

Exit 1

}

if (!$dataProvider.WriteObject($sptCmd, $sptCmdValue))

{

Write-Warning “Impossible Upload Setpoints”

$err = “Last error : ” + $dataProvider.Message

Write-Error $err

Exit 1

}

$obj = $dataProvider.ReadObject($sptDataStatus)

if ($obj -eq $null)

{

Write-Warning “Impossible Upload Setpoints”

Exit 1

}

if (!$dataProvider.WriteObject($sptCmd, "Apply"))

{

Write-Warning “Impossible Upload Setpoints”

$err = “Last error : ” + $dataProvider.Message

Write-Error $err

Exit 1

}

$obj = $dataProvider.ReadObject($sptDataStatus)

if ($obj -eq $null)

{

Write-Warning “Impossible Upload Setpoints”

Exit 1

}

Write-Host “Setpoints loaded.”

…

## 5.5 IntelliLINK Scripts

IntelliLINK contains its own implementation of the PS host and uses it for running scripts (provided they meet the constrains described in chapter 5.2.3.) via the following IntelliLINK menu options:

|  |  |  |
| --- | --- | --- |
| **Menu item** | **Script executed** | **Script parameters passed** |
| Save Memory Snapshot | Save-MemorySnapshot.ps1 | $device, $dataFile |
| Load XDT File | Load-XDT.ps1 | $device, $dataFile |
| Update Firmware | Upgrade-Firmware.ps1 | $device |
| Play Script | Any (user selected) | $device |

### 5.5.1 IntelliLINK Built-In Scripts

When the user selects “Save Memory Snapshot”, “Load XTD File”, or “Update Firmware” menu item IntelliLINK runs built-in scripts that are located in IntelliLINK installation directory, usually it is %ProgramFiles%\S&C Electric\IntelliLINK6 . Other scripts may be run via “Play Script” menu option.

The script Upgrade-Firmware.ps1 determines the controller type of a connected device ($device), and invokes another (controller type and revision-specific) script that will update a device firmware. This (second) script should be installed by a particular product installer.

The script Load-XDT.ps1 reads virtual memory blocks from a file ($datafile - user selected) and writes these blocks to a connected device ($device).

The script Save-MemorySnapshot.ps1 reads all virtual memory from a connected device ($device) and saves it in a file ($datafile - user selected) in vm format.

When the script is executed by the built-in host, it is passed the $device parameter, which contains the Device object for the currently connected controller.

When “Save Memory Snapshot” or “Load XDT File” menu items are selected, the user is prompted with a file selection dialog. After the user selects a file, the file name is passed to the script as $dataFile parameter.

The built-in scripts define a set of other parameters, which makes them possible to run outside of the built-in host. These are the communications parameters which are used to invoke Get-xxxDevice cmdlet and thus obtain the $device object (which was not passed on the command line in the case of script execution outside the built-in host).

### 5.5.2 User-authored Scripts

User-authored scripts (provided they meet the constrains) can be run via Play Script menu option. They should define the $device parameter to be able to utilize it, e.g.

Param ( [object]$device = $null )

For correct releasing created objects it is recommended to [enframe](http://lingvopro.abbyyonline.com/ru/Search/LingvoArticlesAndExtraInfo?text=%d0%be%d0%ba%d0%b0%d0%b9%d0%bc%d0%bb%d1%8f%d1%82%d1%8c&translation=enframe&srcLang=ru&destLang=en&dictionaries=LingvoUniversal%20%28En-Ru%29&author=) your code in the following way:

$myDevice = $null;

try

{

# your code

   ....

   $myDevice = ...

   ...

}

finally

{

       if( $myDevice -ne $null ) { $myDevice.Close() }

}

And for detecting .NET errors you should write in the beginning of a script the following code:

Trap [Exception]

{

Write-Host

Write-Host $("TRAPPED: " + $\_.Exception.GetType().FullName)

Write-Host $("TRAPPED: " + $\_.Exception.Message)

Exit

}

### 5.5.3 IntelliLINK Built-In PowerShell Host (constrains and specialities)

IntelliLINK built-in PS host implements part of the PSHostUserInterface interface. Thus, it cannot run arbitrary PS script. There are the following constrains:

1. The host can display messages created by Write-Host and Write-Warning cmdlets. It does not support Write-Error, Write-Verbose, and Write-Debug cmdlets, which should not be used.
2. Foregroundcolor and backgroundcolor arguments of Write-Host and Write-Warning cmdlets are not supported.
3. Displaying object information by placing the object on the command line does not work.

E.g.,

$a=”A”

$a

does not work, while

$a=”A”

Write-Host $a

works.

1. Clear-Host cmdlet does not work, because the host cannot change screen position, size, or header.

The host *supports* Write-Progress cmdlet.

The host *supports* Read-Host cmdlet and PromptForChoice method for user interaction. The dialog shown on Read-Host allows to enter string or to select file or folder name.

Here is an example of its usage:

$yes = New-Object System.Management.Automation.Host.ChoiceDescription("&1 Yes", "Yes")

$no = New-Object System.Management.Automation.Host.ChoiceDescription("&2 No", "No")

$options = [System.Management.Automation.Host.ChoiceDescription[]]($yes, $no)

$result = $host.ui.PromptForChoice(“Message Text”, $question, $options, 0)

switch ($result)

{

0 { # Yes, (is 0 item in array)

…

}

1 { # No, (is 1 item in array)

…

}

}

$fileName = Read-Host “Enter firmaware file”

# CPS Server

CpsServer.exe is a console application that provides CPS server functionality. Also, this application may be installed as a Windows service.

The CpsServer.exe has the following command line arguments:

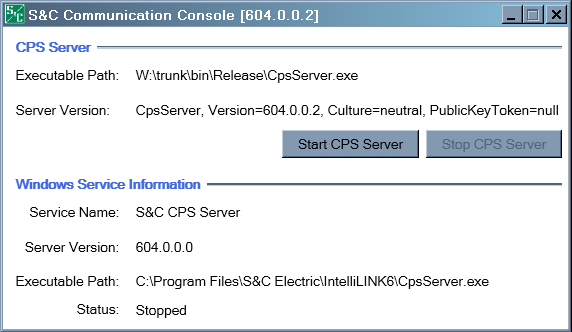
|  |  |
| --- | --- |
| **Command line argument(s)** | **Description** |
| -i=<config\_file>  or  /i=<filename> | Sets a config file. |
| -application  or  / application  or  -a  or  /a | Starts CPS server as a console application. If –not-hide argument is not set, this application will be hidden after start. |
| -not-hide  or  /not-hide | Allows console application window be visible after start. |
| -start  or  /start | Attempts to start CPS server as a Windows service. |
| -stop  or  /stop | Attempts to stop CPS server that is running as a Windows service. |

Starting CpsServer.exe without arguments or with only the –i argument is used for staring the CPS server under Windows Service Manager.

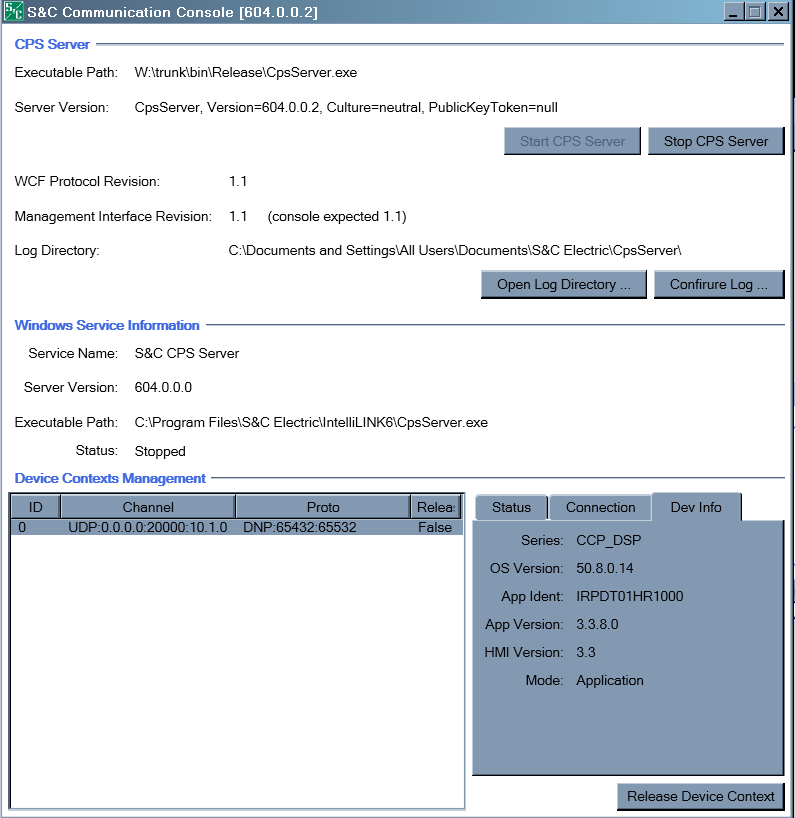
# CPS Console

The CPS console is a Windows application for CPS server management. Only one console window at a time can be shown. An attempt to run the same console will show the current console window (if the console window is hidden in tray, it will be restored).

If CPS server has not been started then the console displays information for CPS server that can be started and information about Windows Service if it is registered. Button “Start CPS Server” is active and can be used for starting CPS server as hidden application



If CPS server is started then the console addidtionally displays information that is taken from the server over the Management WCF interface (server info, active device contexts). The button “Stop CPS Server” is made active and can be used for stopping CPS server (if CPS server is started as Windows Service it also can be stopped).



Using Device Context Management section, active device connections and their properties can be observed, and connections may be terminated.

# Appendix 1. Usage Basics

## 7.1 Logging

All WinKit6 libraries (and as a result CPS and BPS) use logging system based on Microsoft Enterprise Library. The logging can be configured in an application config file or programmatically.

Any WK6 library marks its log records with specific “category”. For CPS the category is named “CPS”, for BPS – “BPS”. Also any log record has a severity (Critical, Error, Warning, Information, etc.), for more information see help for System.Diagnostics.TraceEventType enum.

### 7.1.1 Configuring logging programmatically

The supporting(utility) classes are located in SandC.WinKit6.Common.Logging namespace.

For a simple logging to a text file you should use the following code:

Logger.LogToFile(@“C:\MyApp.log”);

More flexible logging options are provided by a special FileTraceListenerSettings class. This class contains the following properties:

|  |  |  |
| --- | --- | --- |
| **Property** | **Default Value** | **Description** |
| LoggingDirectory | {MyDocuments}/ {CompanyName}/ {ProductName}/ | The directory for writing log file(s). |
| LogFilePrefix | null | The log file prefix. |
| AppendProductInfo | True | If this option is set then product info (assambly name and version) will be added to log file prefix. |
| AppendPid | False | If this option is set then process ID will be added to log file prefix (after product info). |
| RollingEnabled | True | If this option is set then an instance of RollingFlatFileTraceListener class with corresponding RollInterval, RollSizeKB and MaxArchivedFiles params will be created, otherwise instance of FlatFileTraceListener class will be created. RollingFlatFileTraceListener class allows purging current log depending on some conditions and copying old content to rolled (archived) files. *The log file will be rolled if any condiotion is true.* |
| RollSizeKB | 10240 | Maxium log file size (KB) before rolling. |
| RollInterval | None | Rolling period. |
| MaxArchivedFiles | 10 | Maximum number of archived(rolled) log files. |
| TruncateOnStart | False | If this option is set then old log file content with will be deleted or rolled on logging initialization. |
| PurgingOldFilesEnabled | False | If this option is set then all log files matching the mask {DIR}\[{PREFIX}.][{PRODUCT\_NAME}] and modified earlier than {PurgingLifetime} days before now will be deleted. |
| PurgingLifetime | 3 | Period (days) for automatic purging of old log files. |
| Filter | null | Settings of logged entry filter. This is an instance of LogFilterData class. |

The full log file name is:

{DIR}\[{PREFIX}.][{PRODUCT\_NAME}.{VERSION}.][{PID}.]log.

and archived (rolled) log file name is:

{DIR}\[{PREFIX}.][{PRODUCT\_NAME}.{VERSION}.][{PID}.]{TIMESTAMP}.{SEQ}.log

For logging to a file with filtering by category and severity you should use the following code:

LogFilterData filter = new LogFilterData();

filter.DefaultLevel = TraceEventType.Warning;

filter.SetFilter("CPS", TraceEventType.Information);

filter.SetFilter("BPS", TraceEventType.Verbose);

FileTraceListenerSettings settings = new FileTraceListenerSettings() {

LoggingDirectory = null,

LogFilePrefix = @“C:\MyApp.log”,

AppendProductInfo = false,

RollingEnabled = false,

Filter = filter

};

Logger.LogToFile(settings);

If rolling is initialized then the PerformRoll() method can be used to perform immediate rolling (the main log file will be truncated).

Logger.PerformRoll();

Also for truncating the current log of rolled type as well as for flat the StartNewLogFile method can be used.

Logger. StartNewLogFile();

But some log messages may be lost when this method is used with a flat log file, because logging can be disabled and then started again.

To set a custom log writer you need to add references Microsoft.Practices.EnterpriseLibrary.Common.dll and Microsoft.Practices.EnterpriseLibrary.Logging.dll to your project and after that to use the following code:

Logger.LogWriter = new MyLogWriter ();

Here MyLogWriter class is based on abstract LogWriter class. Also you can use LogWriterImpl class (a simple implementation of LogWriter).

To disable logging:

Logger.DisableLogging();

By default if logging is not set programmatically the logging setting is checked with an application config file for each log record writing. And if logging is also not set in the config file then log record writing finishes with an exception throwing and so active logging can cause performance degradation.

### 7.1.2 Configuring logging with an application config file

Add in app.config file (or in <YourProgram>.exe.config) the following definition:

|  |
| --- |
| <configSections>  <section name="loggingConfiguration" type="Microsoft.Practices.EnterpriseLibrary.Logging.Configuration.LoggingSettings, Microsoft.Practices.EnterpriseLibrary.Logging, Version=5.0.414.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" requirePermission="true"/>  …  </configSections>  <loggingConfiguration name="" tracingEnabled="true" defaultCategory="General">  <listeners>  <add name="Event Log Listener"  type="Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners.FormattedEventLogTraceListener, Microsoft.Practices.EnterpriseLibrary.Logging, Version=5.0.414.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"  listenerDataType="Microsoft.Practices.EnterpriseLibrary.Logging.Configuration.FormattedEventLogTraceListenerData, Microsoft.Practices.EnterpriseLibrary.Logging, Version=5.0.414.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"  source="Enterprise Library Logging"  formatter="BasicTxtFormatter"  log=""  machineName="."  traceOutputOptions="None"/>  <add name="Rolling Flat File Trace Listener"  type="Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners.RollingFlatFileTraceListener, Microsoft.Practices.EnterpriseLibrary.Logging, Version=5.0.414.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"  listenerDataType="Microsoft.Practices.EnterpriseLibrary.Logging.Configuration.RollingFlatFileTraceListenerData, Microsoft.Practices.EnterpriseLibrary.Logging, Version=5.0.414.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"  fileName="MyApp.log"  footer=""  formatter="BasicTxtFormatter"  rollFileExistsBehavior="Overwrite"  timeStampPattern="yyyy-MM-dd"  rollSizeKB="1024"  maxArchivedFiles="5"  traceOutputOptions="None"/>  </listeners>  <formatters>  <add type="Microsoft.Practices.EnterpriseLibrary.Logging.Formatters.TextFormatter, Microsoft.Practices.EnterpriseLibrary.Logging, Version=5.0.414.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"  template="Timestamp: {timestamp(yyyy-MM-dd hh:mm:ss.fff)}&#xD;&#xA;EventId: {eventid}&#xD;&#xA;Category: {category}&#xD;&#xA;Message: {message}&#xD;&#xA;Severity: {severity}&#xD;&#xA;Extended Properties: {dictionary({key} - {value}})}&#xD;&#xA;"  name="BasicTxtFormatter"/>  </formatters>  <categorySources>  <add switchValue="All" autoFlush="true" name="General">  <listeners>  <add name="Rolling Flat File Trace Listener"/>  </listeners>  </add>  </categorySources>  <specialSources>  <allEvents switchValue="All" name="All Events">  <listeners>  <add name="Rolling Flat File Trace Listener"/>  </listeners>  </allEvents>  <notProcessed switchValue="All" autoFlush="true" name="Unprocessed Category">  <listeners>  <add name="Rolling Flat File Trace Listener"/>  </listeners>  </notProcessed>  <errors switchValue="All" name="Logging Errors &amp; Warnings">  <listeners>  <add name="Event Log Listener"/>  </listeners>  </errors>  </specialSources>  </loggingConfiguration> |

Also “EntLib Config” application from the Enterprise Library distributive can be used for modification of an application config file.

## 7.2 IAsyncRequest interface

Access to major functionality of the libraries is provided via API that allows asynchronous access, simultaneous access, and the possibility to abort the outgoing request. This is achieved uniformly in all libraries via IAsyncRequest pattern (similar in various respects to IAsyncResult pattern of .NET).

*Note: IAsyncRequest pattern was developed before the release of .NET 4 which provides similar functionality, allowing aborting a call, in CancellationToken and CancellationSource classes. In the future, internal workings of the libraries may be implemented through the use of these classes, and additional APIs may be added to provide the framework suggested by these classes.*

A general pattern is as follows: usually, there is a class which provides some services - e.g., a software interface to the device to which requests can be made. Each request is executed on a *Request Control Block (RCB)* class, specific to the service, on its own instance of RCB. Each RCB is capable of executing only one request at a time. After RCB completes execution of a request, it may be re-used to execute another request. There may be several simultaneously running RCBs on the same logical request handler(for example on the same DNP connection).

Request control block class implements IAsyncRequest interface. The interface has the following members:

public interface IAsyncRequest

{

bool CompletedSynchronously {get;}

bool IsCompleted {get;}

WaitHandle AsyncWaitHandle {get;}

bool IsAborted {get;}

Action<IAsyncRequest> Callback {get; set;}

object UserToken {get; set;}

void Abort();

void Close();

}

|  |  |
| --- | --- |
| **Interface Member** | **Description** |
| CompletedSynchronously | The request completed synchronously. The result is available immediately. |
| IsCompleted | The request has completed (synchronously or asynchronously). |
| AsyncWaitHandle | A WaitHandle which becomes signaled when request is completed. |
| IsAborted | True, if the request was aborted. |
| IsClosed | The request control block was disposed and cannot be used anymore. |
| Callback | A delegate which is called when asynchronous execution of the request completes. |
| UserToken | Arbitrary object that may be set or get by the user at any time. The library never sets or gets the UserToken object. |
| Abort() | Aborts the execution of request that is being executed on this RCB. |
| Close() | Disposes the resources associated with this RCB. |

Members of the IAsyncRequest interface provide properties and methods common to all RCBs. Each type of particular RCB has its own specific methods to initiate a request and to get the results of the request. These methods follow the pattern:

* If synchronous form of a request is

int Operation(TIn inParam, ref TRef refParam, out TOut outParam);

then in RCB definition the following methods of asynchronous pattern must be also defined:

void OperationAsync(TIn inParam, ref TRef refParam);

int GetOperationResult(ref TRef refParam, out TOut outParam);

The method OperationAsync initiates a request. The method GetOperationResult may be called when the request is completed and returned int value as a completion code. If it is 0 then the request is completed successfully. If it is 1 then the request was aborted by user. Otherwise it is an error code specific for this RCB.

* If synchronous form of a request is

void Operation(TIn inParam, ref TRef refParam, out TOut outParam);

then in RCB definition the following methods of asynchronous pattern must be also defined:

void OperationAsync(TIn inParam, ref TRef refParam);

bool GetOperationResult(ref TRef refParam, out TOut outParam);

The method GetOperationResult returns false if the request was aborted by user, otherwise - true.

**For example:**

We have a method for reading a byte from a file starting with some position:

void Read(string file, int position,out byte value, out IOException error);

Then asynchronous pattern of the method is:

void ReadAsync(string file, int position);

bool GetReadResult(out byte value, out IOException error);

The second method returns false if async operation is aborted.

**Operation details:**

1. Request completed synchronously:

If a request completes synchronously, the Callback is not called, the value of CompletedSychronously is true, IsCompleted is true and AsyncWaitHandle is in signaled state. Client application should check the CompletedSychronously property to determine if the request has completed synchronously.

1. Request completed asynchronously:

Inside the client application, the end of an asynchronously executed request can be determined with:

* receiving the Callback (is registered).
* transition of AsyncWaitHandle event to signaled state.

In both cases , when the request has completed the execution, IsCompleted is true and AsyncWaitHandle is signaled, then GetOperationResult may be called to obtain operation status.

If Abort() or Close() were executed, the RCB cannot be used for further requests.

If Abort() was called, the Callback may or may not be called, depending on the state of request execution during which the Abort() was called.

Close() executes Abort() internally, so that any running request is terminated.

Close() should be called eventually for every RCB so that resources associated with the RCB could be released.

## 7.3 Virtual Memory Manipulation Interface

This is an interface for the access to abstract memory. This interface provides operations for reading and writing collections of memory blocks.

This interface consists of:

* IVirtMemRequest interface

providing read and write operations based on the asynchronous pattern;

* IVirtMemProvider interface

containing only method for creating IVirtMemRequest instances;

* VirtMemBlock class

representing a block of virtual memory and used in read and write operations.

All these interfaces and classes are located in SandC.WinKit6.Common.VirtualMemory namespace. Also in this namespace there are several implementations of IVirtMemProvider working with files with specific formats (vm, xsd).

* READ operation:

*The VirtMemBlock collection should be given to the operation. An internal buffer of each block must be initialized. And after success completion of the operation the buffers will contain the read operation results.*

* + Synchronous methods

int ReadVirtMem(

IEnumerable<VirtMemBlock> virtMem,

IProgress<long> progress = null);

*Returns zero for success.*

*Returns one if the request is aborted in the other thread.*

*Returns error coder otherwise.*

bool ReadVirtMem(

IEnumerable<VirtMemBlock> virtMem,

IProgress<long> progress, out WkError error);

*Returns true and error is null when success.*

*Returns true and error is not null when error situation occurs.*

*Returns false if the request is aborted in the other thread.*

* + Asynchronous methods

void ReadVirtMemAsync(

IEnumerable<VirtMemBlock> virtMem,

IProgress<long> progress = null);

int GetReadVirtMemResult();

*Returns zero for success.*

*Returns one if the request is aborted.*

*Returns error coder otherwise.*

bool GetReadVirtMemResult(out WkError error);

*Returns true and error is null when success.*

*Returns true and error is not null when error situation occurs.*

*Returns false if the request is aborted.*

* + Example

List<VirtMemBlock> blocks = new List<VirtMemBlock>();

blocks.Add(new VirtMemBlock()

{ Address = 0x200, Length = 0x100, Buffer = new byte[0x100] });

blocks.Add(new VirtMemBlock()

{ Address = 0x300, Length = 0x100, Buffer = new byte[0x100] });

IVirtMemRequest vmreq = provider.CreateVirtMemRequest();

int result;

if ((result = vmreq.ReadVirtMem(blocks)) != 0)

{

Console.WriteLine("Read error: " + result);

}

foreach (VirtMemBlock block in blocks)

{

Console.WriteLine(block.AsString());

}

* WRITE operation:

*The VirtMemBlock collection should be given to the operation. An internal buffer of each block must be initialized and data from the buffer will be written.*

* + Synchronous methods

int WriteVirtMem(

IEnumerable<VirtMemBlock> virtMem,

IProgress<long> progress = null);

*Returns zero for success.*

*Returns one if the request is aborted in the other thread.*

*Returns error coder otherwise.*

bool WriteVirtMem(

IEnumerable<VirtMemBlock> virtMem,

IProgress<long> progress,

out WkError error);

*Returns true and error is null when success.*

*Returns true and error is not null when error situation occurs.*

*Returns false if the request is aborted in the other thread.*

* + Asynchronous methods

void WriteVirtMemAsync(

IEnumerable<VirtMemBlock> virtMem,

IProgress<long> progress = null);

int GetWriteVirtMemResult();

*Returns zero for success.*

*Returns one if the request is aborted in the other thread.*

*Returns error coder otherwise.*

bool GetWriteVirtMemResult(out WkError error);

*Returns true and error is null when success.*

*Returns true and error is not null when error situation occurs.*

*Returns false if the request is aborted in the other thread.*

# Appendix 2. Base Protocol Services

## 8.1 Overview

Base protocol system (BPS) is a WK6 low-level communication library that provides access to basic protocol and physical channel functionality. The primary intent is to provide a high-level interface for the communication with a device and to hide protocols and physical links implementation. The BPS library can be used to write any protocol oriented software.

The BPS is implemented as in-process library. Serial ports and sockets opened by the BPS library belong to the process and thus are not accessible to other processes on the same computer. However, there may be several processes using BPS and running on the same computer, provided that they open different serial ports and sockets. A single process may instantiate several instances of BPS engine, provided that the engines use different serial ports and sockets.

## 8.2 Main features

The BPS 1.0 provides the following features:

* Physical links:
  + Serial port
  + UDP/IP
  + TCP/IP
* Communication protocols:
  + DNP
  + ICP
* DNP connection (master device) functions:
  + Send some DataLink requests (ResetLink, RequestLink).
  + Send Application layer request.
  + Read / Write virtual memory over DNP.
* DNP (slave) device functions:
  + Support for sending response on a received request.
  + Send unsolicited responses.
* ICP connection functions:
  + Send ICP request (Read or Write virtual or physical memory block).

More protocols and physical channels may be added in the future.

It is supposed that only one BPS instance can work with a device at a time. If two BPS on the same host work with the same physical link (for UDP this case is possible), the result will be unpredictable.

The BPS works as if it worked with an abstract connection to device which executes one operation at a time. Therefore two users cannot work simultaneously with one device.

The BPS does not know anything about the virtual memory architecture of the device, order of read/write operations in multi-operation commands (like authentication), internal privileges etc. (It's the CPS that provides these functions).

The BPS supports the case when one physical link services several devices. This link has the following restrictions:

* If the user sets DNP address (65532) to DnpPeerAddress a physical link will only communicate with the first DNP device having answered.
* A physical link can communicate with only one ICP device.

The physical link opens when the device starts to be used and closes when the last device on the link is closed.

## 8.3 Using BPS library

### 8.3.1 Configuring client application

To start working with BPS add to your project the following references:

* SandC.WinKit6.Communications.Common.dll
* SandC.WinKit6.Communications.BPS.dll
* SandC.WinKit6.Communications.Dnp.dll

### 8.3.2 Creating a device connection

All BPS services are accessed by creating an instance of BpsCore class:

BpsCore bps = new BpsCore();

BpsCore provides methods for creating connections and devices (device emulators). A connection is a class that represents the particular physical devices to which the communication request will be made. The device (emulator) allows sending responses on the requests received over a physical channel.

Both connection and device are based on abstract class CommEntry that defines the following members.

public IPhysChannel Channel {get;}

public IChannelParameters ChannelParameters {get;}

public IProtocolParameters ProtocolParameters {get;}

public int Timeout {get; set;}

public int Retries {get; set;}

public void Close();

|  |  |
| --- | --- |
| **Member** | **Description** |
| Channel | Represents methods for channel manipulation. For example sets baud rate on SerialChannel. |
| ChannelParameters | Gets current channel parameters. |
| ProtocolParameters | Gets current protocol parameters. |
| Retries | Gets or sets the number of attempts to perform a request. |
| Timeout | Gets or sets timeout used for a singe operation, the full operation timeout can be calculated as (Timeout+1) \* Retries. |
| Close | Detaches the CommEntry from physical channels. If channel has no entries it will be released. |

To create a CommEntry with communication parameters (see their descriptions below) you can use:

public CommEntry CreateCommEntry(ICommParameters commParams);

Also several convenient methods are available which do not require to set many of the communication parameters in ICommParameters and return a more specialized derived class instead of generic base class CommEntry.

public CommEntry CreateCommEntry(IProtocolParameters protocol, IChannelParameters channel);

public DnpConnection CreateDnpUdpConnection(IPAddress localIP, ushort localPort, IPAddress peerIP, ushort peerPort, ushort localDnpAddress, ushort peerDnpAddress, DnpProfile dnpProfile);

public DnpConnection CreateDnpTcpConnection(IPAddress localIP, ushort localPort, IPAddress peerIP, ushort peerPort, ushort localDnpAddress, ushort peerDnpAddress, DnpProfile dnpProfile);

public DnpConnection CreateDnpTcpConnection(IPAddress peerIP, ushort peerPort, ushort localDnpAddress, ushort peerDnpAddress, DnpProfile dnpProfile);

public DnpConnection CreateDnpSerialConnection(string serialPort, ushort localDnpAddress, ushort peerDnpAddress, DnpProfile dnpProfile);

public DnpDevice CreateDnpUdpDevice(IPAddress localIP, ushort localPort, IPAddress peerIP, ushort peerPort, ushort localDnpAddress, ushort peerDnpAddress, DnpProfile dnpProfile);

public DnpDevice CreateDnpTcpDevice(IPAddress localIP, ushort localPort, IPAddress peerIP, ushort peerPort, ushort localDnpAddress, ushort peerDnpAddress, DnpProfile dnpProfile);

public DnpDevice CreateDnpSerialDevice(string serialPort, ushort localDnpAddress, ushort peerDnpAddress, DnpProfile dnpProfile);

The instances created by Create<Proto>Device and Create<Proto>Connection methods are instances of derived classes which may be (depending on particular parameters) DnpConnection, IcpConnection or DnpDevice.

### 8.3.3 Communication parameters

A whole set of communication parameters needed to establish a connection to a device is represented by ICommParameters interface:

public interface IBaseCommParameters

{

IProtocolParameters ProtocolParameters { get; }

IChannelParameters ChannelParameters { get; }

int Retries { get; set; }

int Timeout { get; set; }

}

IProtocolParameters is a base for several particular protocol parameter classes:

public interface IDnpProtocolParameters : IProtocolParameters

{

bool Master { get; }

ushort LocalDnpAddress { get; }

ushort PeerDnpAddress { get; }

ushort DnpProfile { get; }

}

public interface IIcpProtocolParameters : IProtocolParameters

{

ushort Address { get; }

}

IChannelParameters is a base class for several channel parameter classes:

public interface IChannelParameters

{

ChannelType ChannelType { get; }

}

public interface ISerialChannelParameters : IChannelParameters

{

string PortName { get; }

int BaudRate { get; }

}

public interface INetworkChannelParameters : IChannelParameters

{

IPEndPoint Local { get; }

IPEndPoint Remote { get; }

}

A simple implementation of this interface contains in classes:

|  |  |
| --- | --- |
| **Class** | **Description** |
| DnpProtocolParameters | Represents DNP communication parameters. It implements IDnpProtocolParameters. |
| IcpProtocolParameters | Represents ICP communication parameters. It implements IIcpProtocolParameters. |
| SerialChannelParameters | Represents parameters for communication using a serial port. It implements ISerialChannelParameters. |
| TcpChannelParameters | Represents parameters for communication using a TCP/IP. It implements INetworkChannelParameters. |
| UdpChannelParameters | Represents parameters for communication using a UDP/IP. It implements INetworkChannelParameters. |
| BaseCommParameters | Complete communication parameters. It implements IBaseCommParameters. |

**Example:**

* Create a connection to a DNP slave device over UDP.

CommBaseParameters dnpConnectionParams = new CommBaseParameters() {

ChannelParameters = new UdpChannelParameters(

IPAddress.Parse("10.4.0.17"), 20000,

IPAddress.Parse("10.8.0.239"), 20000),

ProtocolParameters = new DnpProtocolParameters(**true**, 2, 65532),

Timeout = 2000,

Retries = 0

};

DnpConnection dnpConnection = BpsCore.CreateCommEntry(dnpConnectionParams)

as DnpConnection;

* Create a slave device (emulator) that handles request on a serial port.

CommBaseParameters dnpSlaveParams = new CommBaseParameters() {

ChannelParameters = new SerialChannelParameters("COM1", 9600),

ProtocolParameters = new DnpProtocolParameters(**false**, 2, 56),

Timeout = 2000,

Retries = 0

};

DnpDevice dnpSlave = BpsCore.CreateCommEntry(dnpConnectionParams)

as DnpConnection;

### 8.3.4 Communication Channels

A channel represents a particular physical link between a local computer and a remote device. There are three types of channels: serial, TCP/IP, UDP/IP. More channel types may be added in the future.

A channel used by a particular Device is accessible via the following property of CommEntry class:

public IPhysChannel Channel { get; set; }

The IPhysChannel interface is an abstract root for particular channel classes. It provides the property

ChannelType ChannelType { get; set; }

to determine particular channel types (Serial, Tcp or Udp).

Channel parameters determine a particular physical route to a remote device. Hence, changing such parameters changes the identity of the device. Thus, channel parameters cannot be changed “on the fly”. However, there are some channel attributes that do not change the route to the device. Such attribute is, e.g., the baud rate for a serial channel. More channel attributes will be added later: CTS/RTS behavior and parity for serial channels.

**Example:** serial baud rate auto-negotiation on DNP connection.

if (masterDevice.Channel.Serial != null)

{

var rates = SerialChannelParameters.UsefulBaudRates;

var req = masterDevice.CreateRequest();

int idx = rates.Length;

int res;

while ((res = req.ResetLink()) != 0 && idx > 0)

{

masterDevice.Channel.Serial.BaudRate = rates[--idx];

}

Console.WriteLine("BaudRate: " + rates[idx]);

}

### 8.3.5 Connections.

Connection is a class that represents the particular physical devices to which the communication request will be made. An instance of a connection class has properties that represent the current device state and has Create<Proto>Request() method for performing protocol based operations.

#### *8.3.5.1 DnpConnection*

DnpConnection has the following members:

public Action<Fragment> UnsolicitedResponse;

public DnpRequest CreateDnpRequest();

The UnsolicitedResponse delegate is invoked when unsolicited message is received.

CreateDnpRequest method creates a request control block (see IAsyncRequest section) on which a request to the device may be executed. The DnpRequest class inherits from IAsyncRequest and provides the following methods:

public int ResetLink();

public int RequestLink();

public int DoRequest(

Fragment.FunctionCode funcCode,

ICollection<Dnp.DnpObjectSet> dnpobjs,

List<Dnp.DnpObjectSet> result);

and their asynchronous variants. The DoRequest method allows sending arbitrary DNP application layer request to the device.

public Dnp.AppLevel.Fragment.IINFields IIN;

contains the IIN received in the last (unsolicited) response from the device.

Also the class DnpRequest implements a more straightforward access to virtual memory (DNP object 102) via methods

public int ReadVirtMem(

ICollection<VirtMemBlock> virtMem,

IProgress<long> progress = null);

public int WriteVirtMem(

ICollection<VirtMemBlock> virtMem,

IProgress<long> progress = null);

and their asynchronous variants. For more information see 4.3. These methods may be thought of as convenience functions for DoRequest for a special case of object 102-only request.

#### *8.3.5.2 IcpConnection*

IcpConnection has the following members:

public DnpRequest CreateIcpRequest();

CreateIcpRequest method creates a request control block (see IAsyncRequest section) on which a request to the device may be executed. The IcpRequest class inherits from IAsyncRequest and provides the following methods:

public int Read(

MemoryClass memoryClass,

IEnumerable<VirtMemBlock> blocks,

IProgress<long> progress = null)

public int Write(

MemoryClass memoryClass,

IEnumerable<VirtMemBlock> blocks,

IProgress<long> progress = null)

and their asynchronous variants.

In ICP request the following memory classes can be used

public enum MemoryClass : byte

{

VirtualMemoryMainApplication = 0,

VirtualMemoryOS = 12,

ProgramStorageMainApplication = 13,

ProgramStorageOS = 14,

PhysicalMemory = 15,

}

Returned error code is one of enum IcpRequest.ErrorCode items. If error code equals ErrorCode.IcpError then IcpRequest.IcpError property contains the following ICP error code:

public enum IcpError : byte

{

IllegalFunctionRequest = 1,

IllegalVirtualAddressRange = 2,

IllegalByteCount = 3,

SlaveFailure = 4,

Busy = 5,

InvalidPassword = 6,

WrongDataAlignment = 7,

AccessDenied = 8,

}

Also the class IcpRequest implements a more simple access to application virtual memory via methods

public int ReadVirtMem(

ICollection<VirtMemBlock> virtMem,

IProgress<long> progress = null);

public int WriteVirtMem(

ICollection<VirtMemBlock> virtMem,

IProgress<long> progress = null);

and their asynchronous variants. For more information see 4.3

## 8.4 Important limitations of BPS

1. Although in other parts of WinKit6 libraries several requests (IAsyncRequest) may be executed simultaneously, the BPS has the following limitation: for every particular DnpConnection there may be only one request being executed at a time. An attempt to execute another request while there is a request being executed which has not completed yet will result in InvalidOperationException. This is by design. The reason is that in most cases upon receiving a response from the device there is a need to make the next request get some status information (e.g.: authentication session status). Simultaneous executing of several requests would make that impossible.
2. In general, it is possible to make several simultaneous requests to different devices sharing the same physical channel. However, the following rule applies: a request to a device with Self address (0xFFFC) may only be issued if there is no other executing request *to any device on the same physical channel with the same local (master) address.* The reason is that it is impossible to distinguish responses from the device with Self address and the response from some other device with particular address.

## 8.5 Program example

A very simple example that sends a reset link request to a device via UDP/IP connection and reads a virtual memory block:

var engine = new BpsCore();

var device = engine.CreateDnpUdpConnection(

IPAddress.Parse("10.1.0.17"), // local address

20000, // local port

IPAddress.Parse("10.1.0.239"), // device address

20000, // device DNP port

2, // local dnp address

65532); // device dnp address (65532 - self)

device.Timeout = 5000;

device.Retries = 1;

DnpRequest request = device.CreateDnpRequest();

int result;

if ((result = request.ResetLink()) != 0)

{

Console.WriteLine("SendResetLink failed: {0}", result);

}

VirtMemBlock block = new VirtMemBlock() {

Address = 0x200,

Length = 0x80,

Buffer = new byte[0x80]

};

if ((result = request.ReadVirtMem(new VirtMemBlock[] { block })) != 0)

{

Console.WriteLine("ReadVirtMem failed: {0}", result);

}

request.Close();

device.Close();

engine.Close();

## 8.6 Working over DNP

### 8.6.1 DNP Objects

All DNP objects are represented by classes derived from the abstract class DnpObject. The classes reside in the namespace SandC.WinKit6.Communications.Dnp.ObjectLibrary. For each DNP objects group there is a partial class, e.g. AnalogInput, AnalogOutput, BinaryInput, Counter, etc. DNP objects of a given group with particular variation are represented by an inner class of the group class, having the name of a particular variation. E.g., BinaryInput.PackedFormat, BinaryInputEvent.AbsolutTime.

Each particular DNP object type has properties representing the data contained in the object. For instance, BinaryInput.PackedFormat has

public bool Value;

while BinaryInputEvent.AbsolutTime has

public bool Value;

public BinaryInput.StatusFlags Flags;

public DnpTime Time;

Objects of these types may be created as

var binaryInputPackedFormat = new BinaryInput.PackedFormat(true);

var binaryInputEventAbsoluteTime = new binaryInputEvent.AbsoluteTime(true,

BinaryInput.StatusFlags.Online, new DnpTime(DateTime.Now));

Consult the accompanying .chm documentation for detailed description of DNP objects properties and available constructors.

### 8.6.2 DNP Object Sets

#### *8.6.2.1 Definitions*

DNP object sets are classes based on DnpObjectSet class that provides the following properties:

|  |  |
| --- | --- |
| **Property** | **Description** |
| Group | Represents the group of contained objects. |
| Variation | Represents the variation of contained objects. |
| WithValues | Indicates whether the object set contains actual values of the objects or not. |

Also this class supports a visitor pattern for various handling of object sets. For more information see 8.6.2.3

DNP object sets are represented by the following template classes:

|  |  |  |  |
| --- | --- | --- | --- |
| **DNP Object Set Class** | **Qualifiers** | **Main properties** | **Qualifier automatically selected based on:** |
| DnpObjectSetArray | 00, 01, 02,  03, 04, 05 | Start – first index; Count – number of elements | Maximum (Stop) index |
| DnpObjectSetIndexed | 17, 18, 19,  27, 28, 29,  37, 38, 39 | Objects – the list of pairs (index, object). | Number of objects; maximum index used |
| DnpObjectSetAll | 06 | - | 06 |
| DnpObjectSetList | 4B, 5B, 6B | Objects – the list of objects. | Maximum size of contained object |
| DnpObjectSetCount | 07, 08, 09 | Count – number of objects | Selected based on Count value |

DNP object sets are created using corresponding constructors. For instance,

var objectSet = new DnpObjectSetArray<BinaryInput.PackedFormat>(0, 100, false);

The last argument of the constructor in this case indicates that the object set does not contain actual values of the objects, but only object set header. Thus the created object set may be used in a read request.

For object sets that contain actual values of the objects, those values may be added by corresponding methods.

#### *8.6.2.2 Creating DNP Object Sets*

Sometimes a DNP Object Set should be created in such a way that the resulting application fragment size does not exceed a given value (determined by a device profile). There are helper functions towards this end: classes FragmentBuilder, ObjectSetIndexedBuilder, ObjectSetListBuilder.

FragmentBuilder is initialized with the maximum fragment size. Its method

int GetArrayCount<TDnpObject>(uint start);

provides the maximum possible size of the array of TDnpObject objects with starting index start.

ObjectSetIndexedBuilder and ObjectSetListBuilder provide methods

public bool AddObject(uint index, TDnpObject dnpObject);

public bool AddObject(TDnpObject dnpObject);

which add the object to the object set if it is possible without overflowing the fragment size, or return false otherwise.

#### 8.6.2.3 Processing DNP Objects and DNP Object Sets

Sometimes there is a need to write a code that will process DNP objects or DNP object sets, and exact types of those objects (object sets) are not known in advance. For instance, in handling of a received fragment. Towards this end, the DNP library uses the Visitor pattern.

Processing of an arbitrary DNP object (that is, a derived class of the DnpObject class) can be implemented as a class implementing the following interface:

public interface IDnpObjectHandler

{

bool Visit<TDnpObject>(TDnpObject dnpObject)

where TDnpObject : DnpObject, new();

}

The class DnpObjectHandler provides an implementation of IDnpObjectHandler. Handlers for particular DNP object types may be added by a method

public void AddHandler<TDnpObject>(Predicate<TDnpObject> handler)

where TDnpObject : DnpObject;

See DnpObjectViewer program example.

Processing an arbitrary DNP object set (that is, a template class DnpObjectSetXXX<TDnpObject> where exact type of DnpObject is not known in advance) can be implemented as a class implementing the following interface:

public interface IDnpObjectSetHandler

{

bool VisitArray<TDnpObject>(DnpObjectSetArray<TDnpObject> objectArray) where TDnpObject : DnpObject, new();

bool VisitBlockArray(DnpObjectSetBlockArray objectArray);

bool VisitIndexed<TDnpObject>(DnpObjectSetIndexed<TDnpObject> objectSetIndexed) where TDnpObject : DnpObject, new();

bool VisitAll<TDnpObject>(DnpObjectSetAll<TDnpObject> objectSet) where TDnpObject : DnpObject, new();

bool VisitList<TDnpObject>(DnpObjectSetList<TDnpObject> objectSet) where TDnpObject : DnpObject, new();

bool VisitCount<TDnpObject>(DnpObjectSetCount<TDnpObject> objectSet) where TDnpObject : DnpObject, new();

}

The class DnpObjectSetHandler provides an implementation of IDnpObjectSetHandler. Handlers for particular DNP object sets may be added by methods:

public void AddArrayHandler<TDnpObject>(Predicate<DnpObjectSetArray<TDnpObject>> handler) where TDnpObject : DnpObject, new();

public void SetBlockArrayHandler(Predicate<DnpObjectSetBlockArray> handler)

public void AddIndexedHandler<TDnpObject>(Predicate<DnpObjectSetIndexed<TDnpObject>> handler) where TDnpObject : DnpObject, new();

public void AddAllHandler<TDnpObject>(Predicate<DnpObjectSetAll<TDnpObject>> handler) where TDnpObject : DnpObject, new();

public void AddListHandler<TDnpObject>(Predicate<DnpObjectSetList<TDnpObject>> handler) where TDnpObject : DnpObject, new();

public void AddCountHandler<TDnpObject>(Predicate<DnpObjectSetCount<TDnpObject>> handler) where TDnpObject : DnpObject, new();

This may look frightening, but actual code is very clear. A sample from DnpObjectViewer (printing values of various DNP objects):

this.AddHandler<BinaryInput.PackedFormat>((binaryInput) =>

{

this.output.AppendFormat(" {0} ", binaryInput.Value);

return true;

});

this.AddHandler<BinaryInput.WithFlags>((binaryInput) =>

{

this.output.AppendFormat(" value={0} ", binaryInput.Value);

this.output.AppendFormat("flags={0} ", binaryInput.Flags);

return true;

});

this.AddHandler<BinaryInputEvent.AbsoluteTime>((binaryInputEvent) =>

{

this.output.AppendFormat(" value={0} ", binaryInputEvent.Value);

this.output.AppendFormat(" flags={0} ", binaryInputEvent.Flags);

this.output.AppendFormat("time={0} ", binaryInputEvent.Time.ToDateTime());

return true;

});

# Appendix 3. Scripts localization

The built-in scripts follow the guidelines for PS script localization. The localized strings which should be used as messages to the end-user in a script file .\<scriptname>.ps1, are contained in the file

.\<locale>\<scriptname>.psd1

The file contains a dictionary key=value in the following syntax:

ConvertFrom-StringData @'

ImpossibleCloseDevice = Impossible to close the device.

ImpossibleDownloadSetpoints = Impossible to download the setpoints.

‘@

The script may then display localized messages as follows:

Import-LocalizedData -BindingVariable UserMessages -FileName <scriptname>.psd1

Write-Host $UserMessages. ImpossibleCloseDevice

The script engine automatically selects the appropriate \<locale>\ subdirectory based on the current locale.