SPICE 1.0 - Documentation

Projekttitel:	SPICE 1.0
Dokumentname:	SPICE_1.0_Documentation.docx
Erstellt durch:	Mueller Lukas
Erstellt am:	23.10.2015
Letzte Änderung durch:	Mueller Lukas
Letzte Änderung am:	29.10.2015
Gedruckt am:	29.10.2015
Revisionsstand:	Freigegeben

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

SPICE_1.0_Documentation.docx
Print: 29.10.2015

Seite 1 / 14

Änderungsnachweis

Version	Änderung	Autor	Datum
1.0	Created	MLU	23.10.15

Inhaltsverzeichnis

INSTITUTE FOR LAB AUTOMATION AND MECHATRONICS

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

1   Introduction

SPICE stands for **S**iLA **P**rovider **I**mplementation **C**ommunity **E**quipment.

Idea is to have a public open source available base for a SiLA ServiceProvider to simplify the entrance to the SiLA World / SiLA Device integration for device manufacturer. And if the interest is high enough there will also be the power in future to enhance SPICE and adapt to changed SiLA DCDIS / DCIS versions.

SPICE can be used as *SiLAConverter* like the developer *ILT Institute for Lab Automation and Mechatronics* to equip legacy devices with a SiLA Interface or it can by use by device manufacturer to fully integrate a SiLA Interface within their devices.

An implementation based on SPICE would be called a SPI (**S**iLA **P**rovider **I**mplementation) like the also open source available *SPI_Multidrop*.

1.1   License

SPICE is provided under the Boost Software License. SPICE is not and has no intention to be part of the Boost libraries. It just uses this very open license that encourages both commercial and non-commercial use.

```
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

Source: http://www.boost.org/users/license.html

1.2   Language / Requirements

As one of the unique open source libraries SPICE is written in C++. This is to get a better possibility for device manufacturer to integrate a SiLA Interface within their devices.
- C++ 11
- Platform supporting multiple threads

SPICE_1.0_Documentation.docx
Print: 29.10.2015
Seite 3 / 14

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUTE FOR LAB AUTOMATION
AND MECHATRONICS

1.3   SiLA Version

- SPICE 1.0 implements the SiLA DCDIS 1.3.08 specification.

1.4   Features

Standard:
- Full implementation of the state machine
- Automatic command handling (synchronous responses, get parameters and check limits, error handling communication, response communication)
- Command buffering supported
- Parallel command execution supported
- Implementation of all mandatory commands
- Implementation of some common commands
- Automatic WSDL-file generation
- WSDiscovery supported

Programming:
- Modular structure.
- Modules are just connected over interfaces and base classes.
- Possibility to replace the EthernetServer and XMLParser with other implementations based on other libraries. (Currently available implementation uses the POCO-libraries).
- Possibility to replace the Core with a slim version without features like command buffering and parallel command execution. (Not available yet. Could be generated as there are enough requests)
- Base classes for commands. You just have to derivate and implement the methods.
- Classes for DataTypes and DataSets to easy work with parameters and configuration sets.

Multiple devices:
- One application can have multiple instances of the core to provide more than one SiLA device. They have to differ over there URI-Paths. This feature is interesting for SiLAConverters with legacy devices to support more than one device with only one SiLAConverter hardware.
- One Core can be registered at different EthernetServer to support multiple ports or HTTP and HTTPS at the same time.

Organizational:
- Template project available
- Real implementation as example available (SPI_Multidrop)
- Tested with different Process Management Systems

1.5   Missing

- Autoreset after timeout
- DataEvent generation
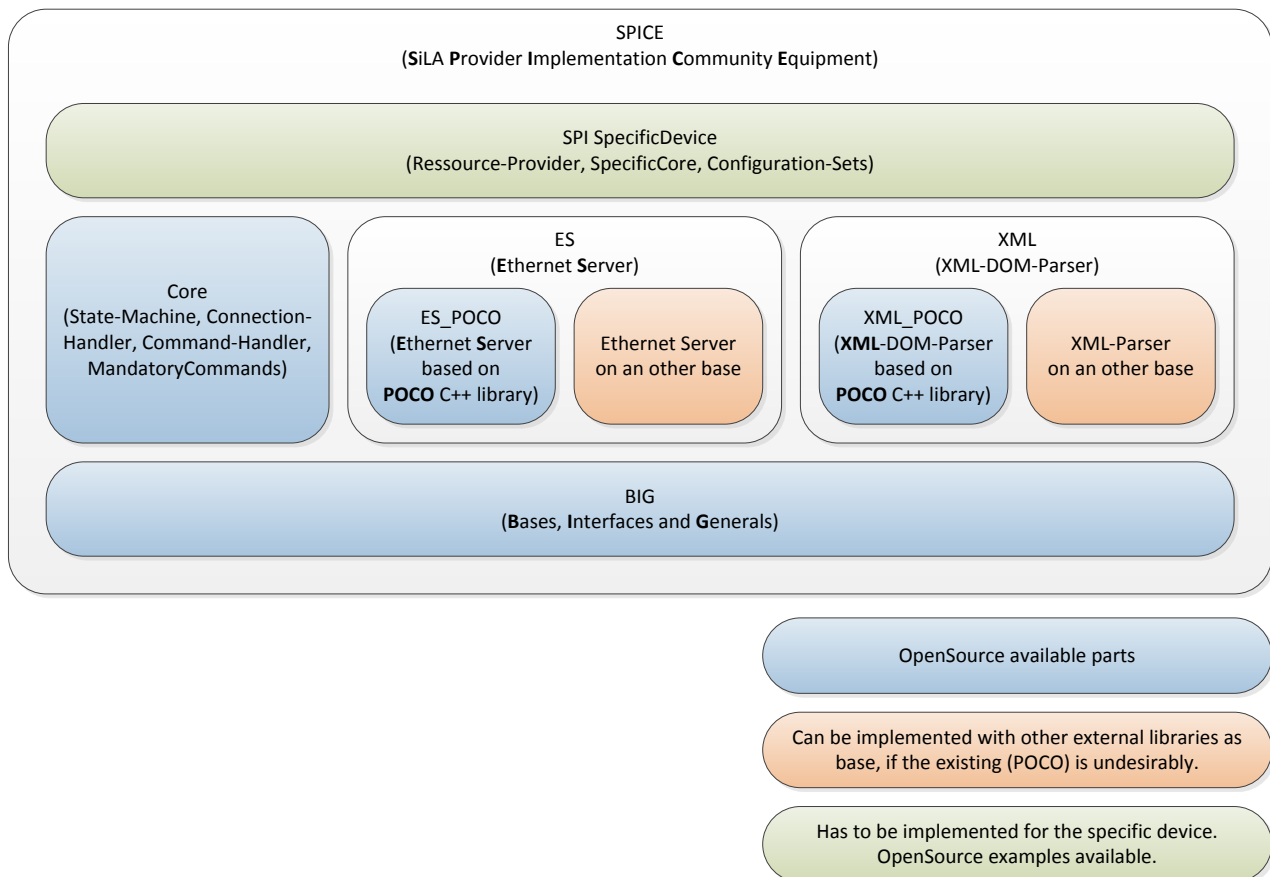- Special DataTypes

1.6   Possible features for future

- Additional EthernetServer and XMLParser implementations based on other libraries to get a choice.
- EthernetServer with HTTPS-support.
- Optional extension field to error events.
- Better / easier ResponseData generation.
- General logging system

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

SPICE_1.0_Documentation.docx
Print: 29.10.2015

Seite 4 / 14

2 Structure overview

A typical SPI device implementation has 5 parts

- BIG-Library
- Core
- EthernetServer (ES)
- XMLParser (XML)
- SPI_SpecificDevice. Only this one has to be implemented.

SPICE
(SiLA Provider Implementation Community Equipment)

SPI SpecificDevice
(Ressource-Provider, SpecificCore, Configuration-Sets)

Core
(State-Machine, Connection-Handler, Command-Handler, MandatoryCommands)

ES
(Ethernet Server)

ES_POCO
(**E**thernet **S**erver based on **POCO** C++ library)

Ethernet Server on an other base

XML
(XML-DOM-Parser)

XML_POCO
(**XML**-DOM-Parser based on **POCO** C++ library)

XML-Parser on an other base

BIG
(Bases, Interfaces and Generals)

OpenSource available parts

Can be implemented with other external libraries as base, if the existing (POCO) is undesirably.

Has to be implemented for the specific device. OpenSource examples available.

Most important points:

- The BIG-Library provides bases and interfaces for derivation by Core, ES, XML and SPI_SpecificDevice.
- The Core-, ES- and XML-Libraries just work over the BIG-Interfaces with each other. They don't reference to each other directly (possibility to replace ES and XML)
- The SPI_SpecificDevice creates the different instances (Core, ES, XML) and connects them to each other.

3 SPI implementation guide

3.1 Structure

The following simplified class diagram shows you a typical structure of a device specific implementation. The green classes should be implemented.
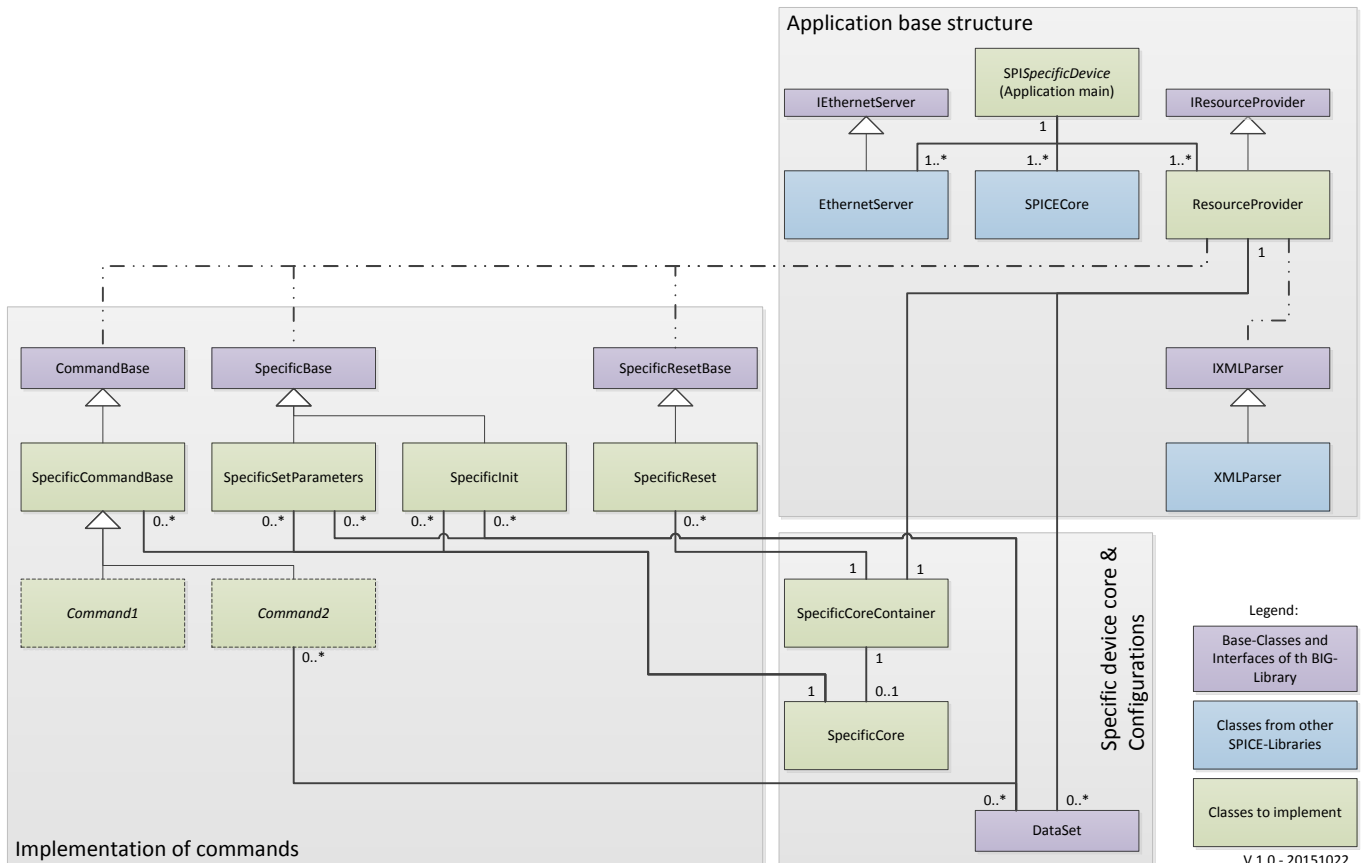
- 🔥 **Application base structure**
  Create instances of the used Core, EthernetServer, XMLParser and the own created ResourceProvider. Connects them to each other.

- 🔥 **Implemenation of commands**
  Commands which can be called by SiLA. Will be created and given to the core by the ResourceProvider.

- 🔥 **Specific device core & configurations**
  The specific core of your device. Should provide methods to the implemented commands.

If you want to work with this structure you can use the template if you want. If you have problems using it there is at least one real implementation available. **It is really recommended that you take a look to these examples**.

HSR
HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

SPICE_1.0_Documentation.docx
Print: 29.10.2015

Seite 6 / 14

3.2 Explanation of the classes to implement

3.2.1 SPISpecificDevice

- This class is the entry point of the application
- It creates and keeps the instances for the Core, the used EthernetServer and the ResourceProvider and registers them at each other.

Rules:
- It is possible to have more than one device per application. For each device create a Core- and a ResourceProvider-instance.
- It is allowed to register multiple cores at one EthernetServer. They have to differ with their URI-Paths.
- It is allowed to register one core at multiple EthernetServers. As example at a HTTP- and a HTTPS-Server. The servers have to differ with their port-usage.
- WSDiscovery is not tested with multiple cores or multiple EthernetServers. Use with care.

3.2.2 ResourceProvider

- Derivation of *IResourceProvider*
- This class provides any needed resources of the specific implementation to the core.
- Creates and keeps an instance of the *SpecificCoreContainer*, which contains the *SpecificCore*.
- Create and keeps the instances of configurations (DataSet)
- Creates and gives XMLParser-instances to the core
- Creates and gives Command-instances to the core. Commands can get references to the *SpecificCore* and the configurations (DataSet).
- Provides the configurations and other informations (deviceClass, device identification) to the core.

3.2.3 SpecificCoreContainer

- Creates and keeps the instance of the *SpecificCore*
- Used to be able to have a deep reset by deleting the *SpecificCore*-instance and creating a new one during the execution of a Reset-command.
- In case of a full integration and the *SpecificCore* is more than a communication core and should not be delete during a Reset-command, you may not need this class.

3.2.4 SpecificCore

- The core of your implemented device.
- Should provide methods which can be called by the command implementations.
- In case of converting for a legacy device this is often a communication core. As example providing and handling a RS232-connection.
- In case of a full integration it can really be the core of your device.

3.2.5 SpecificCommandBase

- Derivation of the *CommandBase*.
- You can implement general functions of your commands here and derivate your commands from this specific command base. Then they are available at every command.
- If you don't have such functions you could remove this class and derivate your commands directly from *CommandBase*.

3.2.6 SpecificSetParameters / SpecificInit

- Derivation of the *SpecificBase*
- *Initialize* and *SetParameters* are implemented within the core. But they provide callbacks to this base, so that it is possible to execute specific actions like activating changed parameters at the specific core.

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

SPICE_1.0_Documentation.docx
Print: 29.10.2015

Seite 7 / 14

3.2.7 SpecificReset

- Derivation of *SpecificResetBase*
- *Reset* is implemented within the core. But it provides callbacks to this base. Different to the *SpecificBase* cause of possibility to change the simulation mode during the reset command. Required to know the old and the new mode.

3.3 Detailed explanation of the most important bases and interfaces

Just a short detailed explanation of the most important bases and interfaces. Their parameters and return values can be found in the Doxygen-Documentation.

3.3.1 CommandBase

Function: Base class for your commands. Your commands have to derivate from the CommandBase. It provides a by the core known interface to work with your command implementations.

Methods:

setCommandCallback

Function: The core gives an instance of the callback-interface to your command. You will need it to give back information (as example: throw an error)

Implementation: DO NOT OVERRIDE!

getCommandName

Function: Give back the name of the command as string.

Implementation: Abstract – You have to implement this method.

getCommandDescription

Function: Give back the description of your command. The description is implemented in the WSDL-file.

Implementation: Abstract – You have to implement this method.

createAndGetAdditionalCommandParameters

Function: Within this function you can define the parameters which your command has. You will create them as DataType specific derivations of the Class DataEntry. For each parameter you can define limits, a default value, a description and if it is required or optional.

Implementation: You just have to implement it in case your command has additional parameters.

getResponseDataInformation

Function: If your command has data sets to send with the ResponseData, they have to be created within this function. Is used by the core to generate the WSDL-file.

Implementation: You just have to implement it in case your command has to give data back at the end of the execution.

calculateEstimatedDuration

Function: Gives back the estimated duration of the command. Has to be able to be called before the defined parameters are given (for generation of the WSDL) and after they are given and calculate the real time (at command launch). Can be solved easy by define default values for the parameters. If time can't be calculated, return 0.

Implementation: Abstract - You have to implement this function.

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

SPICE_1.0_Documentation.docx
Print: 29.10.2015

Seite 8 / 14

⚙ isCommonCommand

Function:	Gives back the information if you command is a common command (defined in the device class) or an own definition.
Implementation:	Abstract - You have to implement this method.

⚙ additionalCommandCheck

Function:	You defined parameters will be checked on the defined limits and if they are present in case they aren't optional. But if additional checks are required, as example dependencies between parameters, the can be done with this method.
Implementation:	Just required if you want additional checks for your parameters.

⚙ readyForStart

Function:	Is called to decide if the command can be started now. Default implementation is that it is ready if no other command before is active anymore. So this function is needed if you want to be able to execute more than one command parallel.
Implementation:	Only needed in case that you want to be able to execute commands parallel.

⚙ processing

Function:	This is the main execution function. It will be called when the command is executed by the core. Here you implement your actions and maybe calls to you SpecificCore.
Implemenation:	Abstract – You have to implement this method

⚙ simulationProcessing

Function:	This is the main execution function in case the simulation mode is active. Default implementation is that the calculated estimated duration is executed as waiting time.
Implementation:	You just have to implement this if you want to have another behavior in simulation mode.

3.3.2   ICommandCallback

This interface is given to your command implementation and provides functions to interact if the core.

⚙ hasToPause / hasToAbort / hasToReset / hasToAbortOrReset

Function:	Methods which have to be called periodic during the processing to be able to react on such requests. **Pause**: Try to reach a stable paused state as soon as possible. If reached call the Method enterPause. **Abort**: Stop / abort as fast as possible and reach a secure state if possible. If during this process an error occurs or not a secure state can be reached throw an error. **Reset**: Directly return as fast as possible. Don't do any actions anymore. Implement such actions as part of the Reset-command.

⚙ enterPause

Function:	Has to be called after your command reaches the pause state. Gives the core the information that the pause state is reached. Will be stay within this method until a DoContinue, Reset or Abort is executed.

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

SPICE_1.0_Documentation.docx
Print: 29.10.2015

Seite 9 / 14

🌶 throwError

Function: Method to throw an error. Gives back the by the PMS selected ContinuationTask. Also call this function if there is no real ContinuationTask in case of a fatal error.

🌶 getWaitingCommandsCount

Function: Gives information on how many commands are in the command buffer. Can be used for "readyForStart"-decisions.

🌶 getActiveCommandCount

Function: Gives information on how many commands are currently active. Can be used for "readyForStart"-decisions.

🌶 getActivCommandsList

Function: Gives a map with the requestIds and commandNames of all active commands. Can be used for "readyForStart"-decisions.

🌶 setResponseEventData

Function: If the command gives back any responseData it can be set with this command as formatted xml-file.

🌶 getParameterSet

Function: Gets the parameterSet of the device.

🌶 getConfigurationSet

Function: Gets the configurationSet for the requested configLevel.

🌶 configurationSetHasChanged

Function: Can be called to send information to the ResourceProvider that a configurationSet has changed and the new configuration should may be written to the hard drive.

🌶 getNewXMLParserInstance

Function: Gets a new instance of the XMLParser created by the ResourceProvider

3.3.3 IResourceProvider

This interface has to be implemented by your ResourceProvider to provide everything the core needs.

🌶 getInstanceOfAllCommands

Function: Is called to be able to generate the WSDL-file. It has to give one instance of every implemented command.

🌶 createCommand

Function: Create and return an instance of a command of the given command name. If there is no such command return a *nullptr*.

🌶 getSpecificInit

Function: Create and return an instance of your *SpecificInit*-implementation

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

SPICE_1.0_Documentation.docx
Print: 29.10.2015

Seite 10 / 14

🔥 getSpecificSetParameters

Function: Create and return an instance of your *SpecificSetParameter*-implementation

🔥 getSpecificReset

Function: Create and return an instance of your *SpecificReset*-implementation

🔥 getParameterSet

Function: Return a shared_ptr to the *ParameterSet*.

🔥 getConfigurationSet

Function: Return a share_ptr to the *ConfigurationSet* of the given *configLevel*. Return *nullptr* if no such *configLevel* exists.

🔥 configurationSetHasChanged

Function: Is called as information that the *configurationSet* of the given *configLevel* has changed. Possible safe-actions can be executed.

🔥 getDeviceIdentification

Function: Return the device specific identification information to the core.

🔥 getDeviceClass

Function: Return the device class

🔥 getCoreConfigurationParameter

Function: General return of different parameters requested by name.

URI_PATHNAME -> as example: "Dispenser"

SERIAL_NUMBER_EXTENSION -> Extension to the readed SerialNumber of the SerialNumber-file. Only needed if more than one core is used within the application. Default return: "" (empty string)

USE_WSDISCOVERY -> "true" or "false". Enable or disable WSDiscovery.

NICELABFIX_SUBSTATES -> "true" activates a niceLab specific fix. Will be removed in future. Couldn't be tested with a current version of niceLab.

SILA_NS_PREFIX -> If the SOAP-XML-files should not use a global namespace the requested prefix could be defined here. NOT RECOMMENDED!. Default: "" (empty string). Example: "sila:"

🔥 getNewXMLParserInstance

Create and return a new XMLParser instance to the core.

4 General KnowHow

4.1 Build on Windows

For a build on Windows you can use the given VisualStudio solution.

A Debug-build generates static libraries named *SPICE_...d.lib*
- SPICE_BIGd.lib
- SPICE_Cored.lib
- SPICE_ES_POCOd.lib
- SPICE_XML_POCOd.lib

A Release-build generates static libraries named *SPICE_...lib*
- SPICE_BIG.lib
- SPICE_Core.lib
- SPICE_ES_POCO.lib
- SPICE_XML_POCO.lib

4.2 Build on Linux / Debian

For a build on Linux / Debian the development environment *Code::Blocks* is used. So there is no *makefile* available yet. Just use the given *Code::Blocks Workspace* like the VS solution on Windows.

Compilation is done with GNU GCC.

A Debug-build generates static libraries named *libSPICE_...d.a*
- libSPICE_BIGd.a
- libSPICE_Cored.a
- libSPICE_ES_POCOd.a
- libSPICE_XML_POCOd.a

A Release-build generates static libraries named *libSPICE_....a*
- libSPICE_BIG.a
- libSPICE_Core.a
- libSPICE_ES_POCO.a
- libSPICE_XML_POCO.a

4.3 POCO-Libraries

You can download the POCO-Libraries here: http://pocoproject.org/
- The *BasicEdition* without any external dependencies is enough.
- The POCO-Libraries are also licensed under the Boost Software License.
- SPICE 1.0 was developed using POCO 1.4.6p4
- SPICE 1.0 was also tested using POCO 1.6.1. On Windows there can be problems with some Debug-symbols. On Linux and Release-version on Windows work good.

Used POCO-Libraries for ES_POCO and XML_POCO:
- PocoFoundation
- PocoNet
- PocoXML

Use the dynamic libraries. The POCO-Libraries have some cross references between each other. That could give you problems linking the static libraries.

INSTITUTE FOR LAB AUTOMATION
AND MECHATRONICS

4.4 Preprocessor-Symbols

The following symbols are currently available for the preprocessor:

BIG

- LINUX -> Use if builded on LINUX

Core

- LINUX -> Use if builded on LINUX
- WSDISCOVERY_DEBUG -> Gives Debug-information about the wsdiscovery activities within the core.
- SILACONVERTER_BUILD -> Changes the path of the SerialNumber-file to the required folder for a SiLAConverter usage.

ES_POCO

- LINUX -> Use if builded on LINUX
- WSDISCOVERY_DEBUG -> Gives Debug-information about the wsdiscovery activities within the ES
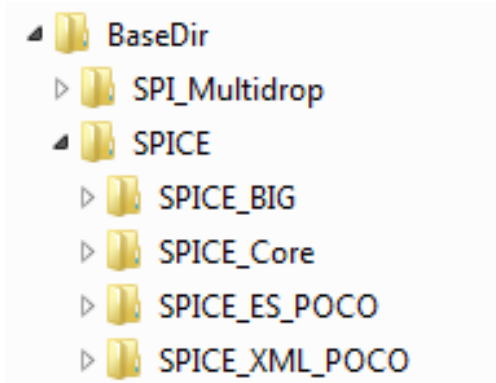
XML_POCO

- LINUX -> Use if builded on LINUX

4.5 SerialNumber

- The SerialNumber is internal an *unsinged Long*. That allows you to have a number in the range of 0...4'294'967'295.

- If a SERIAL_NUMBER_EXTENSION is given it is attached to the *string* of the SerialNumber before it is converted to the *unsigned Long*. So the SerialNumber and the SERIAL_NUMBER_EXTENSION together have to be within the range. (Only needed if more than one core is used within the application)

- The SerialNumber has to be in a file *SerialNumber.txt* in the working directory. Content within the file is just the number, nothing else (at best also without a line break).

- With the PreProcessor-symbol SILACONVERTER_BUILD the file is searched at the relative path: *../SerialNumber/SerialNumber.txt* started at the working directory.

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

SPICE_1.0_Documentation.docx
Print: 29.10.2015

Seite 13 / 14

4.6 Directory tree

The available example and template and their *VisualStudio* and *Code::Blocks* projects are prepared to have the following directory tree.

- BaseDir
  - SPI_Multidrop
  - SPICE
    - SPICE_BIG
    - SPICE_Core
    - SPICE_ES_POCO
    - SPICE_XML_POCO

- The SPICE-Libraries within a directory *SPICE*
- The SPICE-Libraries are builded with the Solution or Workspace which you can find in the directory *SPICE*
- The SPI_*Multidrop* or SPI_*Template* directory is on the same level like the *SPICE* directory.

5 Documentation

There is a complete Doxygen-Documentation of the BIG, Core, ES_POCO and XML_POCO libraries.

It will help to understand the different methods with their parameters and return values.