LABVIEW INSTRUMENT CONTROL TOOLBOX
SADA NÁSTROJŮ PRO OVLÁDÁNÍ PŘISTROJŮ V LABVIEW

DIPLOMOVÁ PRÁCE
MASTER’S THESIS

AUTOR PRÁCE
BC. CTIBOR MAZAL

VEDOUCÍ PRÁCE
DOC. ING. PETR BLAHA, PH. D.

BRNO 2015
Diploma thesis

Master's study field
Cybernetics, Control and Measurements

**Student:** Bc. Ctibor Mazal

**Year of study:** 2

**ID:** 134556

**Academic year:** 2014/15

**TITLE OF THESIS:**

LabVIEW instrument control toolbox

**INSTRUCTION:**

The goal is to provide a LabVIEW virtual instrument toolset which enables to create various control panels for devices connected to single PC controller.

- Become familiar with instrument control layer for different instruments e.g. PXI analog and digital data acquisition cards, PXI digital multimeter, power supplies and optionally some others.
- Design unified signal based interface for instrument control.
- Design Configuration file management which shall support ini file standard and may support alternatives as well.
- Implement signal approach to instrument control in LabVIEW environment which must support Windows 7 operating system.
- Ensure compatibility with vendor drivers as well as Instrument Abstraction Layer.
- Verify created toolbox.

**REFERENCE:**


**Assigment deadline:** 9. 2. 2015

**Submission deadline:** 18. 5. 2015

**Head of thesis:** doc. Ing. Petr Blaha, Ph.D.

**Consultant:** Ing. Miloš Machat

**Subject Council chairman**

**WARNING:**

The author of this diploma thesis claims that by creating this thesis he/she did not infringe the rights of third persons and the personal and/or property rights of third persons were not subjected to derogatory treatment. The author is fully aware of the legal consequences of an infringement of provisions as per Section 11 and following of Act No 121/2000 Coll. on copyright and rights related to copyright and on amendments to some other laws (the Copyright Act) in the wording of subsequent directives including the possible criminal consequences as resulting from provisions of Part 2, Chapter VI, Article 4 of Criminal Code 40/2009 Coll.
Abstract

This diploma thesis is containing the description of the LabVIEW Instrument Control Toolbox project. Initial preparations like the development environment choosing process, as well as the instrument driver layer choice are present along with the project requirements. A signal approach to the instrument control is defined and described in detail. This thesis also contains the main project development in The National Instruments LabVIEW and at the end, a detailed description and user guidance for each developed and fully integrated toolbox module.

Keywords

Statement

“I proclaim that my diploma thesis about LabVIEW instrument control toolbox has been created as my own work under the guidance of my supervisor and with use of technical articles and other informational sources, which are all cited at the end of the thesis.

As the author of the diploma thesis I proclaim, that I haven’t violated any copyrights of third persons and I am fully aware of consequences of their violation according to §11 and Copyright Act no. 121/2000 Sb, including possible criminal responsibilities emerging from subpart VI part 4 Penal Code no. 40/2009 Sb.

Brno, 15th May 2015 ............................................

Signature

Acknowledgement

I would like to express my gratitude to my consultant Ing. Miloš Machat for the useful comments, remarks and engagement through the learning process of this diploma thesis. Furthermore I would like to thank Ing. Pavel Střítecký for introducing me to the topic and for the support on the way.

Brno, 15th May 2015 ............................................

Signature
# Table of contents

INTRODUCTION.......................................................................................................................... 1

1. THEORETICAL ANALYSIS .................................................................................................... 2

   1.1 DEVELOPMENT ENVIRONMENT CHOOSING PROCESS ................................................. 2
       1.1.1 NI LabWindows / CVI .................................................................................... 3
       1.1.2 NI LabVIEW ............................................................................................ 4
       1.1.3 Decision .................................................................................................... 5

   1.2 COMPANY PROJECT ANALYSIS .................................................................................. 7
       1.2.1 Survey .................................................................................................. 7

   1.3 INSTRUMENT DRIVER LAYER .................................................................................. 9
       1.3.1 Standard Commands for Programmable Instrumentation (SCPI) ......................... 9
       1.3.2 Vendor provided Drivers .......................................................................... 10
       1.3.3 Instrument Abstraction Layer ...................................................................... 12
       1.3.4 Customer Layer ........................................................................................ 14

   1.4 PROJECT REQUIREMENTS ....................................................................................... 15

2. SIGNAL APPROACH ......................................................................................................... 16

   2.1 SIGNAL APPROACH ARCHITECTURE .................................................................... 16

   2.2 SOURCE .............................................................................................................. 17

   2.3 MEASUREMENT .................................................................................................... 18

   2.4 CONNECTION ....................................................................................................... 19

   2.5 COMMUNICATION ............................................................................................... 20

3. IMPLEMENTATION IN LABVIEW .................................................................................... 21

   3.1 PROJECT DETAILS ............................................................................................... 22

   3.2 CONFIGURATION FILE MANAGEMENT .................................................................. 24
       3.2.1 Configuration file management architecture .................................................. 26
       3.2.2 CFG_Open.vi ............................................................................................. 26
       3.2.3 CFG_Read.vi ............................................................................................. 27
       3.2.4 CFG_Modify.vi .......................................................................................... 29
       3.2.5 CFG_Close.vi ............................................................................................. 30
       3.2.6 CFG_FindValues.vi ...................................................................................... 30

   3.3 DISCRETE INPUT AND OUTPUT ............................................................................... 31
3.3.1 DIO architecture ................................................................. 31
3.3.2 DIO_Init.vi ................................................................. 34
3.3.3 DIO_SetupSignal.vi ......................................................... 35
3.3.4 DIO_EnableSignal.vi ......................................................... 36
3.3.5 DIO_DisableSignal.vi ......................................................... 37
3.3.6 DIO_ResetSignal.vi ......................................................... 37
3.3.7 DIO_Close.vi ............................................................... 38
3.3.8 DIO_GetAttributeValue.vi ................................................... 39
3.3.9 DIO_SetAttributeValue.vi ................................................... 40
3.3.10 DIO_ReadCfg.vi .......................................................... 41

3.4 ANALOG INPUT AND OUTPUT .............................................. 42
3.4.1 AIO architecture ............................................................. 42
3.4.2 AIO_Init.vi ................................................................. 46
3.4.3 AIO_SetupSignal.vi ......................................................... 47
3.4.4 AIO_EnableSignal.vi ......................................................... 48
3.4.5 AIO_DisableSignal.vi ......................................................... 50
3.4.6 AIO_ResetSignal.vi ......................................................... 50
3.4.7 AIO_Close.vi ............................................................... 51
3.4.8 AIO_GetAttributeValue.vi ................................................... 52
3.4.9 AIO_SetAttributeValue.vi ................................................... 52
3.4.10 AIO_ReadCfg.vi .......................................................... 53
3.4.11 AIO_GetDaqmxSession.vi .................................................. 54

3.5 DC POWER SOURCE .......................................................... 55
3.5.1 DCPWR architecture ......................................................... 55
3.5.2 DCPWR_Init.vi ............................................................. 58
3.5.3 DCPWR_SetupSignal.vi ..................................................... 58
3.5.4 DCPWR_EnableSignal.vi ..................................................... 59
3.5.5 DCPWR_DisableSignal.vi .................................................. 60
3.5.6 DCPWR_ResetSignal.vi ..................................................... 60
3.5.7 DCPWR_Close.vi ........................................................... 61
3.5.8 DCPWR_GetAttributeValue.vi .............................................. 61
3.5.9 DCPWR_SetAttributeValue.vi .............................................. 62
3.5.10 DCPWR_ReadCfg.vi ......................................................... 62
3.5.11 DCPWR_TrippedQuery.vi .................................................. 63
3.5.12 DCPWR_Initiate.vi ........................................................ 64
3.5.13 DCPWR_Abort.vi .......................................................... 64
### 3.5.14 DCPWR_SendSoftwareTrigger.vi

64

### 3.5.15 DCPWR_WriteInstrumentData.vi

65

### 3.5.16 DCPWR_ReadInstrumentData.vi

65

### 3.6 SERIAL INTERFACE

#### 3.6.1 SERIAL architecture

66

#### 3.6.2 SERIAL_Init.vi

71

#### 3.6.3 SERIAL_SetupSignal.vi

71

#### 3.6.4 SERIAL_WriteSignal.vi

72

#### 3.6.5 SERIAL_ReadSignal.vi

72

#### 3.6.6 SERIAL_ResetSignal.vi

73

#### 3.6.7 SERIAL_Close.vi

73

#### 3.6.8 SERIAL_GetAttributeValue.vi

74

#### 3.6.9 SERIAL_SetAttributeValue.vi

75

#### 3.6.10 SERIAL_ReadCfg.vi

76

#### 3.6.11 SERIAL_ConvertStringToUInt8.vi

77

#### 3.6.12 SERIAL_ConvertUInt8ToString.vi

77

#### 3.6.13 SERIAL_GetVISAResource.vi

78

#### 3.6.14 SERIAL_FlushOutQueue.vi

78

#### 3.6.15 SERIAL_FlushInQueue.vi

78

### 3.7 ERROR HANDLING

79

### 3.8 ATTRIBUTE MANIPULATION

#### 3.8.1 GetAttributeValueBoolean.vi

82

#### 3.8.2 GetAttributeValueInteger.vi

82

#### 3.8.3 GetAttributeValueReal.vi

83

#### 3.8.4 GetAttributeValueString.vi

83

#### 3.8.5 SetAttributeValueBoolean.vi

84

#### 3.8.6 SetAttributeValueInteger.vi

84

#### 3.8.7 SetAttributeValueReal.vi

85

#### 3.8.8 SetAttributeValueString.vi

85

### 4. PROOF OF CONCEPT

#### 4.1 STATION SETUP INFORMATION

86

#### 4.2 UUT INFORMATION

87

#### 4.2.1 Signals used

87

#### 4.3 EXAMPLE FUNCTION

89

#### 4.3.1 Initialization

92
Introduction

Nowadays, the majority of measurement instrumentation used in automated solutions doesn't provide the manual controlling or requires remote access. Many ways to program instrument remotely were developed in the past. Some of them have taken form of separate programming languages.

These programming languages often have form of string commands (SCPI, VISA), which are sent through some communication buses (GPIB, LXI, RS-232 etc.). Most instrument vendors are creating their own drivers for their devices in order to facilitate the instrument control to end user.

Graphical user interfaces are being created for easy access to the commands and special development environments were designed to enable the developers to create them easily. The NI LabVIEW is the most used tool for the test station GUI creation in Honeywell Brno. However, coding and design standards weren’t deployed in past and existing design didn’t provide the ability of efficient code reuse amongst projects.

This diploma thesis was dedicated to develop and create the unified set of tools based on the signal abstraction of automated test equipment in the NI LabVIEW environment. This toolset will create coding standards and enable developers to share and reuse control solutions created on different projects. Reuse of previously created solutions will speed up the development process and allow the developers to think of new ideas, rather than focusing on the same tasks again.
1. THEORETICAL ANALYSIS

Sections in this chapter are dedicated to the pre-development preparations. Although they may seem unimportant, these actions ensure maximum efficiency of all the research and development.

1.1 DEVELOPMENT ENVIRONMENT CHOOSING PROCESS

Requirements covered: SYS_0010

Requirements have been derived from the initial preparations of the Labview Instrument Control Toolbox (LICT) project. Each section where applicable is marked with requirements covered by that section. This is done to ensure complete requirement coverage and to ease the requirement tracking at the end of the project. List of requirements and additional information can be found in the section 1.4 Project Requirements. Furthermore, a complete requirement traceability matrix is located in the Appendix B.

This section will describe both development environments, which were selected as best suited for the LICl project. In the last part of this chapter, a final decision of the development environment is presented in a form of a Quality Function Design (QFD) table.
1.1.1 NI LabWindows / CVI

According to the NI LabWindows/CVI description [9], the NI LabWindows/CVI is an ANSI C integrated development environment and engineering toolbox created by the National Instruments Company. It is one of the most used environments for aerospace and military research and development projects.

Figure 1-1: GUI created in CVI

Similarly to the Microsoft’s Visual Studio, it has a capability to create and manage workspaces which can contain many projects. This ability is very useful when dividing SW project into sub-projects, but maintaining management of the whole scope is required. Since this is a text based ANSI C environment, the including of third party dynamic link libraries (DLLs) is simply done through `#include <dll_name.h>` and adding the `.lib` file into the project include paths.
1.1.2 NI LabVIEW

According to the NI LabVIEW description [8], the NI LabVIEW is an environment for graphical programming from the National Instruments Company. It was created to speed up the process of development and debug of measurement and control systems.

In order to utilize third party DLLs, wrapper virtual instruments (VIs) must be made. National instruments implemented wizard, which in few steps guides through the include procedure to facilitate this task.

Figure 1-2: GUI created in LabVIEW

As mentioned in LabView for Engineers [6], a simple VI overloading is present, which will be useful for GUI development on multiple driver layers. Overloading is done through the polymorphic VIs, which are basically wrappers for multiple different VIs. One of the included VIs is selected according to the inputs, which are provided.
1.1.3 Decision

In order to properly select the best suited development environment, a survey inside the Honeywell Brno was made. According to the survey, nearly all projects, in last 3 years, which utilizes graphical user interface for the instrument control, used the NI LabVIEW development environment. This is mainly because of an easy GUI creation and overall high development process speed. However, no coding and design standards were established and zero solution reuse is present across various projects. This is due to the high source complexity, which rises from designing application without proper guidance. Programs aren’t divided into purposed modules and all code is present on a single “main” virtual instrument. Due to this, the program becomes unreadable for unfamiliar programmer. An example can be seen on the Figure 1-3. Note that the white polygon in the middle of the figure is representing an area of two full HD monitors.

Figure 1-3: Standard control program size
The final step of the decision process can be seen on the Table 1-1. Each property, which has an impact on the project, is graded according to the environment ability to provide it. A higher final score means a better capability to provide the properties.

Table 1-1: Quality Function Deployment table

<table>
<thead>
<tr>
<th>#</th>
<th>Customer Needs</th>
<th>Importance Ranking</th>
<th>NI LabWindows / CVI</th>
<th>NI LabVIEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Speed of development process</td>
<td>9</td>
<td>3</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>Code reusability</td>
<td>5</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>GUI creation</td>
<td>9</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Third party DLL include</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>Error handling</td>
<td>3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>Function overloading</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Ease of use</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>Debugging</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>Help and Sample code</td>
<td>3</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>Team skills</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td><strong>218</strong></td>
<td><strong>384</strong></td>
</tr>
</tbody>
</table>

QFD Legend

- Importance Ranking
- Relative Grade
- Absolute Grade
- Final Result

According to the Table 1-1 and after a debate with the Honeywell project leadership, the NI LabVIEW development environment was selected as best suited for this task.
1.2 COMPANY PROJECT ANALYSIS

Requirements covered: SYS_0010, SYS_0040

Recent company instrument control programs had to be analyzed to gain experience from the past projects. These experiences provide the LICT project with best initial data and create the scope of project. Analysis consisted of survey and inspection of each project.

1.2.1 Survey

The head developer of NI LabVIEW GUIs from each team was provided with a survey to determine, which instruments were used in their case. Another part consisted of configuration file management and report generation. Last part was used to determinate which types of buses are used and for what purpose. Instruments, which were creating the LICT modules, were selected and designed based on results of this analysis. Exclusion from the survey results can be found in the Table 1-2. A brief recapitulation is present afterwards.

Table 1-2: Exclusion from survey results

<table>
<thead>
<tr>
<th>Data Generation</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog outputs</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Digital outputs</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Buses</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Were RS232/422/485 buses used?</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Was the ARINC 429 bus used?</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Was the ARINC 761 bus used?</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Was the ARINC 708 bus used?</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Was Ethernet used?</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
### Data Processing

<table>
<thead>
<tr>
<th>Question</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Were array operations used?</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Were point by point operations used?</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Were MathScripts (MatLab) used?</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

### GUI

<table>
<thead>
<tr>
<th>Question</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Were charts used?</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Were graphs used?</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Were 3D graphs used?</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Were 2D arrays indicators/controls used?</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Were custom GUI elements used?</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Were parts of the GUI disabled when unused?</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

According to the survey, discrete input and output devices were used in most of the control programs. However, not all control programs made in NI LabVIEW were purposed to control instrumentation. If the program was made to control instrumentation, it had a discrete I/O signal control. This result was more than sufficient to include DIO devices as part of the LICT. Most of the control programs utilize analog I/O as well as some form of measurement device, mostly digital multimeter or power meter. AIO devices were selected to be part of the LICT for their function as both source of signal and their measurement capabilities. The integration of other measurement devices was postponed in favor of other LICT modules.

The differences were in driver layer selection. Various were present, from vendor provided drivers, across IVI specific, to use of SCPI commands.

Every control program was using some form of a configuration file for additional control of the GUI functions. A report generation was present as well. The diversity was in a form of the configuration and report file format, which were .txt, .xml and .csv.
1.3 INSTRUMENT DRIVER LAYER

Requirements covered: SYS_0020, SYS_0030

This section notes instrument driver layers, which were selected as possible candidates to form a base of the LICT. Their purpose and basic functions are described in each chapter separately.

1.3.1 Standard Commands for Programmable Instrumentation (SCPI)

As mentioned in SCPI section of IVI foundation site [5], SCPI were made as first attempt to unite the instrumentation control and are built on IEEE-488.2 standard. The main advantage of SCPI is that it is hardware independent and can be used with any instrument interface. It works equally with GPIB, LAN, RS-232 or VXIBus.

The ASCII text was used for command syntax. This enables usage of SCPI in every computer test language as well as test application environments, such as Matlab or NI LabVIEW.

**Table 1-3: Basic SCPI commands**

<table>
<thead>
<tr>
<th>Command syntax</th>
<th>Command Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>*CLS</td>
<td>Clear Status</td>
</tr>
<tr>
<td>*ESE?</td>
<td>Event Status Enable Query</td>
</tr>
<tr>
<td>*IDN?</td>
<td>Identify Query</td>
</tr>
<tr>
<td>*OPC?</td>
<td>Operation Complete Query</td>
</tr>
<tr>
<td>*OPT?</td>
<td>Identify Options Query</td>
</tr>
<tr>
<td>*RST</td>
<td>Reset</td>
</tr>
<tr>
<td>*SRE</td>
<td>Service Request Enable</td>
</tr>
</tbody>
</table>

Some of the basic SCPI commands are noted in the Table 1-3. It is apparent that SCPI consist of queries and commands, which differentiate in expected action from the instrument. A query asks for answer, a command is specifying the instrument action.
1.3.2 Vendor provided Drivers

Aside from SCPI commands, instrument vendors often provide their own set of drivers designed specifically for instruments of one type in their catalogue. These mainly come in a form of DLLs, source codes and in case of National Instruments, even virtual instruments. They are free to use to control the instrument remotely or in some complex control solution.

For example National Instruments provides drivers for all of their instrumentation with remote control capabilities, as mentioned at the National Instruments site [7]. They are divided into separate libraries dependent on the purpose of the instrument. Some of them can be found in the Table 1-4

<table>
<thead>
<tr>
<th>Driver Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>NI DAQmx</td>
<td>Data Acquisition driver software</td>
</tr>
<tr>
<td>NI-Serial</td>
<td>Serial Interfaces driver software</td>
</tr>
<tr>
<td>NI VISA</td>
<td>Virtual Instrument Software Architecture</td>
</tr>
<tr>
<td>NI 488.2</td>
<td>Software architecture for developing GPIB applications</td>
</tr>
<tr>
<td>NI Vision Acquisition Software</td>
<td>Software for acquiring, displaying, logging, and monitoring images from a multitude of camera types</td>
</tr>
</tbody>
</table>

Most of instruments from the National Instruments fall into a category of Data Acquisition. Therefore, the NI DAQmx driver is one of the most important drivers from their collection. Main reason why Vendor specific drivers weren’t selected, although the portfolio of function and possibilities to control instruments is far superior to other drivers, is their specialization on a single instrument, which makes the development of unified drivers much more difficult.
1.3.2.1 IVI Compliant drivers

The vendor specific drivers are often based on the IVI standard, which was defined by the IVI foundation [5]. The goal was to create a unified interface for the instrument control. The IVI standard provides a set of defined classes of instruments as well as their control functions. There are currently thirteen classes defined: Digital multimeter (DMM), Oscilloscope, Arbitrary waveform/function generator, DC power supply, AC power supply, Switch, Power meter, Spectrum analyzer, RF signal generator, Upconverter, Downconverter, Digitizer and Counter/timer.

The main advantage of the IVI standard is hardware interchangeability. As long as the vendor and instrument class isn’t changed, the software works regardless of the instrument model used. This is achieved through the use of class and specific drivers, which can be seen on the Figure 1-4.

![Figure 1-4: IVI Architecture – taken from [5]](image)

The class driver, which is called by the user, decides what specific driver to use according to instrument mapped in the IVI configuration file. This basically means that the developer can create software prior the knowledge of the instrumentation used. Another benefit of this architecture is when dealing with hardware obsolescence. The instrument can be changed for a new model with minimal or even without any software changes.
1.3.3 Instrument Abstraction Layer


The Instrument Abstraction Layer (IAL) is a set of instrument drivers based on the IVI standard. IAL drivers abstract the test station hardware from applications allowing interchangeability of hardware between different vendors without changing the software. This isn’t possible with vendor provided drivers, even when based on IVI standard. The IAL is also supported by Simulators for Instrumentation. The IAL is a common set of adaptable interfaces that support resource management, maintenance/debugging interface, data reporting, error handling.

The IAL was designed to support three aspects of test software: 1) Expanding functionality of existing test software by utilizing IAL components. 2) Creating new software for new programs with predefined instruments. 3) Enabling the integration of yet to be defined instrumentation into future products.

The IAL contains a common interface across resource types and has exception handling capabilities required for the safe utilization of the test station resources in numerous testing applications. For multiple unit under test (UUT) (of the same type) communication, the IAL supports thread-safe execution. It can be executed manually or sequentially.
1.3.3.1 **IAL Architecture**

As mentioned in [3], the IAL driver is divided into the Class driver and the Specific driver. The Class driver is the same for all the instruments of one type e.g. DC Power Supply and based on configuration settings decides, which Specific driver to use. The Specific driver is designed to suit the instrument and calls appropriate vendor driver commands, which the function demands. The control architecture for DC Power Supply can be seen on the Figure 1-5.

![IAL DCPS architecture - taken from [3]](image)

**Figure 1-5: IAL DCPS architecture – taken from [3]**

The similarity to the IVI architecture can be seen in a form of Class and Specific drivers. However, it surpasses the vendor drivers in term of the hardware interchangeability. This was one of the main reasons why was the IAL selected as best suited for the LICIT project.
1.3.4  Customer Layer

As mentioned in [1] Customer Layer Library software engineering document, test software developers can control the Automated Test Equipment (ATE) instruments with a standardized set of functions and attributes. The control of instruments is facilitated by use of signals described in the Hardware Property File (HPF), which is a defined form of the configuration file. Signals can be defined for each ATE slot separately, so the test script remains the same for one and multiple channels.

The functions are available in logically separated DLLs. Each instrument type has its own library. They can be called from NI TestStand or other project specific DLLs.

1.3.4.1  CL Architecture

The customer layer test system provides user friendly driver’s API and keeps the IVI architecture advantages (interchangeability, commonality etc.) at once. The customer layer is implemented in the NI CVI/LabWindows (ANSI C) and provides interfaces, which support fully featured usage from the NI TestStand.

![Figure 1-6: Customer layer architecture – taken from [1]](image)

As can be seen on the Figure 1-6, CL drivers are built as a wrapper for the Instrument Abstraction Layer (IAL) drivers and are dependent on them. This is the reason why IAL drivers were selected for the LICT to be built on and not CL drivers. Another reason is their limitation to the NI TestStand and its “ThisContext”. All the error handling is done through the NI TestStand. This limitation would have to be removed first, for the CL to be usable as base for the LICT created in the NI LabVIEW.
1.4 PROJECT REQUIREMENTS

Demands from the customer were gathered before the start of the project to create the initial project requirements. These form the basic structure of work. According to those requirements, final virtual instrument set will be tested. Based on those test results, project success will be evaluated.

Project requirements are noted in the Table 1-5. Requirements marked as “shall” are essential for the project success. Requirements marked as “may” are considered above the scope of work and are meant as improvements, which aren’t essential.

Table 1-5: Project requirements

<table>
<thead>
<tr>
<th>REQ ID</th>
<th>Requirement Text / Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS_0010</td>
<td>LICT shall utilize NI LabVIEW virtual instruments</td>
</tr>
<tr>
<td>SYS_0020</td>
<td>LICT shall utilize Instrument Abstraction Layer</td>
</tr>
<tr>
<td>SYS_0030</td>
<td>LICT may support vendor specific drivers</td>
</tr>
<tr>
<td>SYS_0040</td>
<td>LICT shall be divided into modules. Each module shall represent specific instrument or functionality.</td>
</tr>
<tr>
<td>SYS_0050</td>
<td>LICT shall provide unified interface for instrument control. Description: Each virtual instrument created for instrument control purpose shall follow the same design approach (Signal cluster)</td>
</tr>
<tr>
<td>SYS_0060</td>
<td>LICT shall utilize signal approach to instrument control</td>
</tr>
<tr>
<td>SYS_0070</td>
<td>LICT shall utilize polymorph virtual instruments</td>
</tr>
<tr>
<td>SYS_0080</td>
<td>LICT source code shall be designed to support future custom modifications</td>
</tr>
<tr>
<td>SYS_0090</td>
<td>LICT shall support Discrete Input and Output instruments</td>
</tr>
<tr>
<td>SYS_0100</td>
<td>LICT shall support Analog Input and Output instruments</td>
</tr>
<tr>
<td>SYS_0110</td>
<td>LICT shall support DC power source instruments</td>
</tr>
<tr>
<td>SYS_0120</td>
<td>LICT shall support Serial communication</td>
</tr>
<tr>
<td>SYS_0130</td>
<td>LICT architecture shall allow to add next instruments and capabilities</td>
</tr>
<tr>
<td>SYS_0140</td>
<td>LICT shall cover configuration file management capability</td>
</tr>
<tr>
<td>SYS_0150</td>
<td>LICT architecture shall allow to share control solutions and capabilities between projects</td>
</tr>
<tr>
<td>SYS_0160</td>
<td>LICT file structure shall be designed as a base of common library</td>
</tr>
</tbody>
</table>

Full requirement traceability is present across the thesis. Each chapter or section is marked with requirements that are covered. Additionally, the Appendix B is containing a requirement traceability matrix, which is used to track the requirement coverage by the document.
2. SIGNAL APPROACH

Requirements covered: SYS_0030, SYS_0060

This chapter will describe the signal approach to the instrument control. The signal of an instrument can have defined states, which are dependent on called functions. The description of all signal types and transition functions created for the instrument control can be found in sections below.

2.1 SIGNAL APPROACH ARCHITECTURE

The main goal of the signal abstraction in measurement automation is to facilitate the instrument control to the final user or developer. However, maintaining the level of control, which is present in the lowest driver layers and simplifying the usage to the developer can be proven difficult. In order to achieve this, a compromise was made. The basic instrument control functions were created to handle most of the usage scenarios and test solutions. These VIs are created according to signal definitions shown in sections bellow and function as transitions between signal states. In addition to these basic VIs, advanced instrument control VIs were created to allow the developer to access the instrument settings at lower driver levels (e.g. Vendor Drivers, SCPI).

![Figure 2-1: Signal approach architecture](image)

As can be seen on the Figure 2-1, the signal abstraction was based on the IAL drivers to maintain the IVI architecture and with this, the hardware interchangeability. The configuration file management was created to speed up the process of setting up the signals. However, it isn’t a vital part of the signal abstraction and the signal abstraction layer is fully functional without it.
2.2 SOURCE

All instruments, which provide a stimulus, are marked as “Source” signal type. (E.g. power supply, analog output, signal generator etc.). The state diagram of these instruments is indicated in the Figure 2-2.

![State Diagram](image)

**Figure 2-2: Source Signal State Diagram**

**Init** function changes the state from “Uninitialized” to “Idle”. The function establishes the connection with the instrument, reset its state and set up the driver internal safety parameters as defined in Station Configuration file. (E.g. maximum voltage level what can be set by the library)

**Setup** function changes the state from “Idle” to “Standby”. It configures the instrument as defined by signal parameters but the output stays disabled at this stage.

**Enable** function changes the state from “Standby” to “Generate Output” state. It enables the instrument output.

**Disable** function changes the state from “Generate Output” to “Standby”. It disables the instrument output.

**Reset** function changes the state from “Standby” to “Idle”. It resets the instrument to default state.

**Close** function changes the state from “Idle” to “Uninitialized”. It closes the instrument session.
2.3 MEASUREMENT

All instruments, which measure values, are marked as “Measurement” signal type. (E.g. scope, digital multimeter, discrete input etc.). The state diagram of these instruments is indicated in the Figure 2-3.

![Figure 2-3: Measurement Signal State Diagram](image)

- **Init** function changes the state from “Uninitialized” to “Idle”. The function establishes the connection with the instrument, reset its state.

- **Setup** function changes the state from “Idle” to “Standby”. It configures the instrument as defined by signal parameters.

- **Enable** function changes the state from “Standby” to “Read/Capture” state. It performs the measurement. After successful reading, “Standby” state is restored.

- **Reset** function changes the state from “Standby” to “Idle”. It resets the instrument to default state.

- **Close** function changes the state from “Idle” to “Uninitialized”. It closes the instrument session.
2.4 CONNECTION

All instruments, which provide connection capability, are marked as “Connection” signal type (e.g. Switch, Multiplexer). The state diagram of these instruments is indicated in the Figure 2-4. Note that the connection signal types were not used in the LICT project.

![State Diagram](image)

Figure 2-4: Connection Signal State Diagram

- **Init** function changes the state from “Uninitialized” to “Idle”. The function establishes the connection with the instrument, reset its state and set up the driver internal safety parameters as defined in Station Configuration file.

- **Setup** function changes the state from “Idle” to “Standby”. It configures the connection parameters as defined by signal parameters.

- **Reset** function changes the state from “Standby” to “Idle”. It resets the instrument to default state.

- **Close** function changes the state from “Idle” to “Uninitialized”. It closes the instrument session.
2.5 COMMUNICATION

All instruments, which provide communication capability, are marked as “Communication” signal type (e.g. RS-232, Ethernet). The state diagram of these instruments is indicated in the Figure 2-5.

![Figure 2-5: Communication Signal State Diagram](image)

**Init** function changes the state from “Uninitialized” to “Idle”. The function establishes the communication with the instrument or communication port.

**Setup** function changes the state from “Idle” to “Standby”. It configures the communication parameters as defined by signal parameters.

**Read** function changes the state from “Standby” to “Read” state. It opens the communication and reads the data. After the successful reading, the state is changed back to “Standby”.

**Write** function changes the state from “Standby” to “Write” state. It opens the communication and writes the data. After the successful data write, the state is changed back to “Standby”.

**Reset** function changes the state from “Standby” to “Idle”. It resets the instrument to default state.

**Close** function changes the state from “Idle” to “Uninitialized”. It closes the instrument session.
3. IMPLEMENTATION IN LABVIEW

As mentioned in chapter 2, the signal abstraction will be created through the use of blocks of basic functions. Each function is represented by the virtual instrument consisting of wrapped functions. These virtual instruments (VIs) have unified interface and may work with multiple driver layers in the future.

The main part of the interface is signal cluster as input and output. All information and data are saved inside the signal cluster and VIs themselves will decide, which data to use and alter. These additional data can be defined inside a standardized configuration file. Their retrieval can be done through configuration file management VIs created as part of the LICT project. An example of the Enable virtual instrument for DIO instruments based on IAL drivers can be seen on the Figure 3-1.

![Figure 3-1: DIO “Enable” Virtual Instrument](image)

Note that not all function inputs are used, because not all are required or needed. The goal is to use as input a signal cluster and only the essential additional data needed for the instrument control (e.g. number of samples to read for analog input measurements). Advanced instrument settings will be handled through VIs specialized in the instrument attribute control. Integer, Real, String and Boolean type properties as defined in the IVI standard [5] will be handled.
3.1 PROJECT DETAILS

Requirements covered: SYS_0010, SYS_0020, SYS_0040, SYS_0080, SYS_0130, SYS_0150, SYS_0160.

The NI LabVIEW project was created in order to simplify the LICT development. A repository in the common Brno subversion was created to serve as a project backup and to facilitate data providing to other developers. The NI LabVIEW project contains all the data needed to use and further enhance the LICT project.

The idea behind the LICT project was to create the coding and design standards in the NI LabVIEW and to promote the code reuse amongst developed projects in Honeywell Brno. In order to do so, subtle approach was needed. VIs created for the signal control were discussed with head NI LabVIEW developers and altered to their demands and needs. One of their conditions was to maintain the possibility to control the instrument at the lowest driver layer, aside to prepared VIs.

![Image: NI LabVIEW project tree structure]

Figure 3-2: NI LabVIEW project tree structure
The Figure 3-2 is showing the final NI LabVIEW project tree structure. Each module has its own folder. Basic VIs and signal cluster custom control are located in core. Advanced control VIs are located in the “Utilities” directory. The “TypeDefs” directory contains the attribute type definitions and the “Test” folder contains an integration VI. Integration VIs were created as testing programs for each one of LICT modules. The module selection was based on their usage frequency and adjusted to cover most of the required capabilities. List of the modules and their capabilities is in the Table 3-1.

Table 3-1: Module capabilities

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Capability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIO</td>
<td>Source, Measurement</td>
<td>Discrete input / output</td>
</tr>
<tr>
<td>AIO</td>
<td>Source, Measurement</td>
<td>Analog input / output</td>
</tr>
<tr>
<td>DCPWR</td>
<td>Source</td>
<td>DC power source</td>
</tr>
<tr>
<td>SERIAL</td>
<td>Communication</td>
<td>Serial interface</td>
</tr>
</tbody>
</table>

The “IALWrappers” folder contains wrapped IAL dynamic link library functions, which are used as base for the signal abstraction layer programming. Each wrapper was created with regard to the error handling described in the section 3.7.

A custom help with developer guidance was created for each component along with usage examples in a form of integration VIs.
3.2 CONFIGURATION FILE MANAGEMENT

Requirements covered: SYS_0140

Initial data settings in all programs are done through some sort of a configuration file. In Honeywell Brno, two main types are used – INI and HPF. After consideration, both were chosen for support by the LICIT project. To facilitate this task, FORMAT section in configuration file was added.

The format section is used to help the configuration VI decide how to handle the file correctly. An example of Format section is shown below.

[FORMAT]
Type = INI
Delimiter = /

The “Type” key is used to determine the nature of the configuration files. This value is important because a traditional INI file stores data in 2D perspective. That means one key has one value. However, a HPF utilizes 3D perspective for data storage. This means that one key, which is usually defining a signal, stores multiple properties. Another significant difference is in the HPF header. Each section has its own header, which describes the meaning and the type of value below. This header isn’t created as comment, so it would create a garbage values amongst the retrieved values if handled incorrectly.

The “Delimiter” key is used for a configuration file modifications. The delimiter is used to mark the character, which will separate the section and key in one of VIs parameters.

However, as this section and all the parameters are optional, default values were created. For the “Type” key, an INI file is used as default, because the HPF format is signal specific and used on minority of the projects at the moment. In “Delimiter” key, “::” characters are used if none other is defined.
The Figure 3-3 is showing the part of the HFP used in the LICT project example. Two signals are defined. One signal is defined in the discrete (digital) input section and one in the discrete (digital) output section.

The [FORMAT] section, which was described earlier, can be found at the beginning of the file.

Note that this type of the configuration file is ideal for the instrument control with the signal abstraction layer.
3.2.1 Configuration file management architecture

To keep the configuration file management as simple as possible, file handling was reduced to the absolute basic. Thus only file reading and modification are supported. However, any other action can be done, because the reference to the file can be retrieved and used with the native NI LabVIEW functions.

3.2.2 CFG_Open.vi

Input: path  Configuration file Path
       ErrorCluster  Error in
Output: RefNum  Config refnum out
       ErrorCluster  Error out

Icon: OPEN

Purpose: To open .ini file for read/write purposes.

The document reference number is generated when opening the configuration file located in the path defined as one of the VI inputs. The reference number can be described as a pointer to the opened document and remains valid until the document is closed. If the file isn’t found at the defined location, the function fails and provides appropriate error information.
3.2.3 CFG_Read.vi

Input:
- RefNum  Config refnum
- ErrorCluster Error in

Output:
- string[] Sections
- string[][] Keys
- string[][][] Values
- RefNum Config refnum out
- ErrorCluster Error out

Icon:

Purpose: To retrieve and store all data from the configuration file inside three arrays.

- string[] Sections
- string[][] Keys
- string[][][] Values

First array named “Sections” is a vector containing all sections name. Result of a search for the desired section name is meant to determinate the X dimension in the “Keys” matrix and the X dimension in the “Values” matrix.

The second array named “Keys” is a 2 dimensional matrix containing all the keys of all sections. Search in the “Sections” vector gave us the X dimension to search in this matrix. Finding the correct key will give us the Y dimension value for the “Values” matrix.

Figure 3-5: Values matrix schematic overview

The last array named “Values” contains all the values from all sections and their keys stored in a 3 dimensional matrix. As mentioned above, search in the “Sections” vector gives us the X dimension and search in the “Keys” matrix will provide us with
the $Y$ dimension. The $Z$ dimension in a classic INI file is neglected (the value is always 0), because each key has only one value. However in a HPF, each key has multiple values, which are stored in order and define the $Z$ dimension of the “Values” matrix. The header of a HPF must be consulted in order to choose the $Z$ dimension correctly, because the order and meaning of the values is different in each of the sections. Note that the Figure 3-5 has a different layout of axes. The order was altered to properly demonstrate the configuration data storing.
3.2.4 CFG_Modify.vi

Input:  
- **RefNum**  
- **string[]**  
- **string[]**  
- **string[]**  
- **ErrorCluster**

Output:  
- **RefNum**  
- **ErrorCluster**

Icon:  

Purpose:  
To change the configuration file according to the user input.

The VI for configuration file modification was created to enable the alteration of a configuration file by the program itself. It is used to change the values of keys, but keep all the comments inside the file intact. If the required Key and/or Section aren’t found, the desired section-key connection is created. In case that the section was found but the key wasn’t, new key is created at the end of the section. In case the section wasn’t found, a new section is created at the end of the configuration file.

Main inputs of the Configuration file modification VI are two string arrays and a document reference number.

The array named “Save Names” contains strings created from the connection of the Section and Key name. An example of such connection can be seen below. The items in this array define the target segment of the modification.

Section[delimiter]Key >>> DC_POWER::UUTPower

As mentioned above in section 3.2, the delimiter character or set of characters is configurable by the value of “Delimiter” key in “FORMAT” section. However, if not found, the “::” characters are used as the delimiter by default.

The array named “Save Values” contains the values to be saved in a location corresponding with the “Save Names” array. The previous value of the selected segment is overwritten.
3.2.5   CFG_Close.vi

Input:   RefNum       Config refnum
         ErrorCluster   Error in

Output:  ErrorCluster  Error out

Icon:  

Purpose: To close and save the opened document.

Closing the configuration file will disable all actions connected with the specific document and makes the document reference number invalid. All modifications made in the configuration file will be saved.

3.2.6   CFG_FindValues.vi

Input:   string[]       Sections
         string[][]      Keys
         string[][][]   Values
         string          SearchSection
         string          SearchKey

Output:  string[]       Values out

Icon:  

Purpose: To retrieve the values of a key in defined section.

The function searches the Section array and the Keys matrix as defined in section 3.2.3. Array of attributes found for the key defined by SearchKey in the section defined by the SearchSection is fetched in a form of Values out string array. Note that for a detailed understanding of the values defined in a HPF, its header must be consulted.

Each module has its own wrapper for this VI, which supports the HPF data layout. Use those VIs to transfer the values from configuration arrays to signal clusters.
3.3 DISCRETE INPUT AND OUTPUT

Requirements covered: SYS_0050, SYS_0060, SYS_0090

Discrete Input and Output (DIO) instruments are one of the most used in the measurement automation and can provide both source and measurement capabilities. Additionally, when discrete lines are grouped into channels, a simple output generation and input reading can attain the role of data transfer.

The main role of DIO instruments is a FET control, logical circuit control and triggering of other instruments. The discrete input can be used to monitor logical values in the circuits.

3.3.1 DIO architecture

A signal cluster was developed in order to simplify the instrument control. The signal cluster is used as a medium for main signal values as well as critical signal settings. Due to a similarity between the discrete input and output, only one cluster type is used for both signal types. The signal cluster was created as a Strict Type Def. in the NI LabVIEW and can be seen on the **Figure 3-6**.

![Figure 3-6: DIO signal cluster](image)

```plaintext
enum SignalType
    DO

UInt16 Value

UInt32 InstrumentHandle

string ChannelName

string PhysicalChannel

ErrorCluster SignalErrorInfo
```
**SignalType** enumerator has two states, DO and DI, which represents the Discrete Output and the Discrete Input. This enumerator provides the information for the control VIs on how to handle the signal. Signal types used in the DIO are Source, which is described in section 2.1, and Measurement, which is described in section 2.3.

**ChannelName** string represents the name of the task created for this particular signal and must have a defined structure:

“CH[i]” where \([i] = \{1, 2, 3, 4…\}\)

e.g. “CH1”

Note that the ChannelName must be unique for each signal. If a duplicate ChannelName string is found, appropriate error is created.

**InstrumentHandle** integer represents the number of opened session with the initialized instrument. This number is the same for all signals, which are using the same resource and becomes invalid after the instrument communication closure.

**PhysicalChannel** string is used to define, which physical port and lines to use for this signal. Each signal can consist of one or multiple discrete lines, however only one port can be used in the signal. PhysicalChannel string must have a defined structure:

For single line in port

“[P]/[L]” where \([P]\) …port number

\([L]\) …line number

e.g. “1/0” for port 1 and line 0

For multiple lines in port

“[P]/[L_S]:[L_E]” where \([P]\) …port number

\([L_S]\) …start line number

\([L_E]\) …end line number

e.g. “2/0:7” for port 2 and lines 0 to 7
**Value** unsigned integer is used as a desired hexadecimal value of the discrete output signal and a measured values at the discrete input signal. Each line can attain the value of a logical one or zero. The least significant bit (LSB) is the value of the first line inside the signal. An example is shown on the Figure 3-7.

![Conversion to bit array](image)

**Figure 3-7: DO Value transformation**

**SignalErrorInfo** is a standardized NI LabVIEW error information cluster. It provides information whether the error occurred in a form of a status Boolean, error specific code number in a form of a code integer and error message as a source string.

Controls mentioned above are derived from the HPF and are sufficient for a complete control of the discrete lines.

![DIO signal cluster](image)

**Figure 3-8: DIO control chain**

The Figure 3-8 is showing a DIO control chain and represents the whole code segment needed for a basic DIO instrument control. All functions are described in the upcoming sub-sections.
3.3.2 DIO_Init.vi

Input: string logicalName  
ErrorCluster Error in 

Output: UInt32 InstrumentHandle 
ErrorCluster Error out 

Icon: 

Purpose: To perform initialization of a communication with the DIO instrument.

The VI is used as a transition from the Uninitialized state to the Idle state. Upon entering of the logicalName, which is used in the Measurement & Automation Explorer to map the SW components to the HW assets, the VI will attempt to make a connection with the instrument. If the function is successful, a positive non-zero integer is returned in a form of the Instrument Handle.

The Instrument Handle is unique for each logical name, however it is the same for all the signals operating on this logical name. To differentiate between each signal on same the Instrument handle, a signal ChannelName string is used.

Using this function will reset the device and mark the device as “initialized” so it cannot be initialized again without using the DIO_Close VI first.
3.3.3 DIO_SetupSignal.vi

Input: signal DIO_signal in
Output: signal DIO_signal out

Icon:  

Purpose: To perform the initial setup of a desired input/output port.

The VI is used as transition from the **Idle** state to the **Standby** state. It creates a discrete output channel or discrete input channel based on the enum SignalType. The InstrumentHandle specifies the instrument and the PhysicalChannel ports and lines used. The ChannelName is used as unique identifier for the signal. Note that the output lines retain the previous value after the setup. The Enable or Disable functions must be used in order to achieve the desired output.

The function updates the SignalErrorInfo cluster with information whether the wrapped function failed or succeeded.
3.3.4 DIO_EnableSignal.vi

Input: \textit{signal} DIO\_signal\ in

Output: \textit{signal} DIO\_signal\ out

Purpose: To set the output port to desired logical value.
To read the state of the input port.

3.3.4.1 Discrete Output:

The VI is used as a transition from the \textit{Standby} state to the \textit{Generate Output} state. As mentioned in the \textbf{Figure 3-7}, desired value in hexadecimal form must be transformed into a bit array. Values in array are then distributed across lines used in the signal cluster. The LSB is assigned to the first line in the signal cluster. Only one sample per channel is written. If the output bit array contains more values than the number of lines used in the signal cluster, only the amount equal to the number of lines is written. For example:

\[
A = [1, 0, 1, 1] \text{ on } 0/0:2
\]

\[
0/0 \rightarrow 1
\]

\[
0/1 \rightarrow 1
\]

\[
0/2 \rightarrow 0
\]

\[
0/3 \rightarrow \text{previous state}
\]

3.3.4.2 Discrete Input:

The VI is used as a transition from the \textit{Standby} state through the \textit{Read/Capture} state to the \textit{Standby} state again. By using this VI, each line fetches its logical value, which is saved in a form of a bit array. The array is ordered as: LSB = first line used. In order to provide a single value, transformation to the hexadecimal value is performed. By default 32-bit array is allocated for reading. The unused values remain zero, so that they aren’t affecting the final value.
3.3.5 DIO_DisableSignal.vi

Input: signal DIO_signal in
Output: signal DIO_signal out

Icon:

Purpose: To set the output port to 0.

3.3.5.1 Discrete Output:

The VI is used as a transition from the Generate Output state to the Standby state. Using this function will write logical zeroes at all lines used by signal. Lines outside the specified fragment won’t be affected by this function.

3.3.5.2 Discrete Input:

The VI has no effect on signals with a Measure capability.

3.3.6 DIO_ResetSignal.vi

Input: signal DIO_signal in
Output: signal DIO_signal out

Icon:

Purpose: To reset the defined lines to an unused state.

The VI is used as a transition from the Standby state to the Idle state. Lines must be cleared in order to assign them as a different type. To properly clear the output lines, they must be set to the input state and cleared. Otherwise they will retain a discrete status. This wouldn’t be a problem on a strictly DIO instrument, however most instruments with DIO capabilities share DIO pins with programmable interfaces. These interfaces wouldn’t be accessible as the pin would be marked as occupied by the discrete output by the instrument and their setup would fail.
3.3.7 DIO_Close.vi

Input:  
- UInt32 InstrumentHandle
- ErrorCluster Error in

Output:  
- ErrorCluster Error out

Icon:  

Purpose:  To close the communication with a DIO instrument.

The VI is used as a transition from the **Idle** state to the **Uninitialized** state.
Function closes the session with a DIO instrument with provided InstrumentHandle.
The InstrumentHandle integer will be invalid after the communication disconnection.
Calling DIO_Init.vi will provide a different InstrumentHandle.
3.3.8 DIO_GetAttributeValue.vi

Input:  
- signal DIO_signal in
- TypeDef attributeID

Output:  
- signal DIO_signal out
- VAR attributeValue out

Icon:

Purpose: To retrieve the value of desired attribute.

A polymorphic VI [6] grouping multiple attribute types. The appropriate type is selected by the list box located under the icon. The types grouped in this VI and an example of usage is shown on the Figure 3-9.

![Figure 3-9: DIO GetAttributeValue polymorphic VI](image)

A list of handled attributes is in the section 6.1 of the Appendix A – List of Attributes. Some attributes are device specific (e.g. PXI_MANF_ID, PXI_MODEL_CODE, etc.) and using them with an unsupported device will result in an appropriate error message. The attribute handling and each attribute type is described in detail in the section 3.8 Attribute manipulation.
3.3.9  DIO_SetAttributeValue.vi

Input:  
- signal  DIO_signal in
- typedef  attributeID
- VAR  Value in

Output:  
- signal  DIO_signal out

Icon:

Purpose:  To set the value of desired attribute.

A polymorphic VI [6] grouping multiple attribute types. The appropriate type is selected by the list box located under the icon. The types grouped in this VI and an example of usage is shown on the Figure 3-10.

Figure 3-10: DIO SetAttributeValue polymorphic VI

A list of handled attributes is in the section 6.1 of the Appendix A – List of Attributes. Some attributes are device specific (e.g. PXI_MANF_ID, PXI_MODEL_CODE, etc.) and using them with an unsupported device will result in an appropriate error message. The attribute handling and each attribute type is described in detail in the section 3.8 Attribute manipulation.
3.3.10 DIO_ReadCfg.vi

Input:  

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal</td>
<td>DIO_signal in</td>
</tr>
<tr>
<td>string</td>
<td>SearchKey</td>
</tr>
<tr>
<td>string[]</td>
<td>Sections</td>
</tr>
<tr>
<td>string[][]</td>
<td>Keys</td>
</tr>
<tr>
<td>string[][][]</td>
<td>Values</td>
</tr>
</tbody>
</table>

Output:  

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal</td>
<td>DIO_signal out</td>
</tr>
</tbody>
</table>

Icon:  

Purpose:  

To set the values from the configuration file to the desired signal.

The VI utilizes the FindValues.vi from the configuration sub-section 3.2.6 to search for the SearchKey in “DI” (Discrete Input) section and if no results are found, in the “DO” (Discrete Output) section. If the key isn’t found in either section, an appropriate error message is passed. If the key is found, values are according to the HPF structure distributed to appropriate signal variables.

Figure 3-11: DIO ReadCfg usage example

As can be seen on example shown on the Figure 3-11, the CFG_Open and the CFG_Read VIs must be called first in order to fill the Sections, Keys and Values arrays with a valid data.

Note that the function is designed to work with a HPF data order and will not work if the order isn’t kept. The data order used is noted in the VI as comment.
3.4 ANALOG INPUT AND OUTPUT

Requirements covered: SYS_0050, SYS_0060, SYS_0100

Analog Input and Output (AIO) instruments are used in the measurement automation to provide a source and measurement capabilities. The source capabilities are represented by a current and voltage outputs. The voltage outputs can sustain the role of waveform generator, which can generate various waveforms. Waveforms supported by the LICT are sinusoid, square, triangle and saw tooth.

Analog cards have extensive measurement capabilities, which can in most cases replace the DMM in the ATE. However the AI can only measure current and voltage values. Resistance, frequency and other measurements are not supported.

The main role of AIO instruments is waveform generation, driving of analog values and voltage measurement.

Since analog outputs can drive maximum of approximately 50mA per channel [7], their usage is fairly limited to merely reference signals. Thus analog signals aren’t used as power supplies. This role must be handled by DCPWR signals mentioned in the section 3.5. Even though their use is limited to a reference role, at least one analog signal is used in almost every test solution.

3.4.1 AIO architecture

A signal cluster was developed in order to simplify the instrument control. The signal cluster is used as a medium for main signal values as well as critical signal settings. Only one cluster type is used for both signal types. In order to achieve maximum reusability for both the input and the output channels, only the essential information about the signal is contained. The signal cluster was created as a Strict Type Def. in the NI LabVIEW and can be seen on the Figure 3-12.
**SignalType** enumerator has multiple states, which represents Analog Output and Input types. This enumerator provides information for the control VIs on how to handle the signal. Signal types are categorized as a Source or Measurement, which are described in the sections 2.1 and 2.3. Signal types are defined in the Table 3-2. Note that not all signal types are available for all instruments. If the signal type isn’t supported, an appropriate error message is provided.

**Table 3-2: AIO Signal Types**

<table>
<thead>
<tr>
<th>Signal Type Name</th>
<th>Capability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AO_Voltage_CONST</td>
<td>Source</td>
<td>Constant voltage output</td>
</tr>
<tr>
<td>AO_Voltage_SINE</td>
<td>Source</td>
<td>Sine waveform generation</td>
</tr>
<tr>
<td>AO_Voltage_SQUARE</td>
<td>Source</td>
<td>Square waveform generation</td>
</tr>
<tr>
<td>AO_Voltage_TRIANGLE</td>
<td>Source</td>
<td>Triangle waveform generation</td>
</tr>
<tr>
<td>AO_Voltage_SAWTOOTH</td>
<td>Source</td>
<td>Saw tooth waveform generation</td>
</tr>
<tr>
<td>AO_Current</td>
<td>Source</td>
<td>Constant current output</td>
</tr>
<tr>
<td>Signal Type Name</td>
<td>Capability</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>AI_Current</td>
<td>Measurement</td>
<td>Immediate current value</td>
</tr>
<tr>
<td>AI_Current_RMS</td>
<td>Measurement</td>
<td>Current RMS value</td>
</tr>
<tr>
<td>AI_Voltage</td>
<td>Measurement</td>
<td>Immediate voltage value</td>
</tr>
<tr>
<td>AI_Voltage_RMS</td>
<td>Measurement</td>
<td>Voltage RMS value</td>
</tr>
<tr>
<td>AI_Voltage_PP</td>
<td>Measurement</td>
<td>Voltage Peak-Peak value</td>
</tr>
<tr>
<td>AI_Voltage_BUF</td>
<td>Measurement</td>
<td>Buffered samples per frequency and number of samples</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Currently not supported</td>
</tr>
</tbody>
</table>

**Value** is represented by a double precision number. The value is representing signal amplitude when the signal type with a source capability is selected. When the measurement capability is selected, the Value is representing the measured signal value.

**ChannelName** string represents the name of the task created for this particular signal and must have a defined structure:

“CH[i]” where [i] = \{1, 2, 3, 4,…\}

E.g. “CH1”

Note that the ChannelName must be unique for each signal. If a duplicate ChannelName string is found, appropriate error is created.

**InstrumentHandle** integer represents the number of opened session with the initialized instrument. This number is the same for all signals, which are using the same resource and becomes invalid after the instrument communication closure.

**PhysicalChannel** string is used to define, which physical port and lines to use for this signal. Each signal can consist of one analog input or output. The PhysicalChannel string must have a defined structure:

“[A]” where [A] …analog line number

E.g. “1” for analog line 1

**ChannelMinLimit** double precision is used to define the lower limit of a measured or generated signal. The limit cannot be lower than the instruments capability.
**ChannelMaxLimit** double precision is used to define the upper limit of a measured or generated signal. The limit cannot be higher than the instruments capability. Note that working outside the limits set by the ChannelMinLimit and the ChannelMaxLimit will result in an appropriate error message.

**Frequency** integer is used to set the frequency of a generated waveform. In the analog input mode, the frequency integer is used to set the sampling frequency, which in combination with the NumberOfSamples defines the measurement.

**Offset** double precision defines the output waveform offset as well as the input channel measurement offset. Note that the offset is not supported by all channel types at the moment.

**NumberOfSamples** integer is used to set the number of samples to gather for the measurement. In case of the waveform generation, the NumberOfSamples integer is representing the number of samples generated per cycle. The maximum of 3000 samples is taken if kept zero or asked for more.

**SignalErrorInfo** is a standardized NI LabVIEW error information cluster. It provides information whether the error occurred in a form of a status Boolean, error specific code number in a form of a code integer and error message as a source string.

The controls mentioned above are derived from the HPF and are sufficient for a complete control of analog instruments.

![AIO signal cluster](image)

**Figure 3-13: AIO control chain**

The Figure 3-13 is showing an AIO control chain and represents the whole code segment needed for a basic AIO instrument control. All functions are described in the upcoming sub-sections.
3.4.2 AIO_Init.vi

Input: string logicalName
       ErrorCluster Error in

Output: UInt32 InstrumentHandle
        ErrorCluster Error out

Icon:  

Purpose: To perform initialization of a communication with the AIO instrument.

The VI is used as a transition from the Uninitialized state to the Idle state. Upon entering of the logicalName, which is used in the Measurement & Automation Explorer to map the SW components to the HW assets, the VI will attempt to make connection with the instrument. If the VI is successful, a positive non-zero integer is returned in a form of the Instrument Handle.

The Instrument Handle is unique for each logical name. However it is the same for all the signals operating on this logical name. To differentiate between each signal on the same Instrument handle, a signal ChannelName string is used.

Using this function will reset the device and mark the device as “initialized” so it cannot be initialized again without using the AIO_Close VI first.
3.4.3 AIO_SetupSignal.vi

Input:  
\begin{verbatim}
  signal  AIO_signal in
  enum InputTerminalConfig (optional)
\end{verbatim}

Output:  
\begin{verbatim}
  signal  AIO_signal out
\end{verbatim}

Purpose:  
To perform the initial setup of a desired input/output analog channel.

The VI is used as a transition from the Idle state to the Standby state. It creates an analog output channel (voltage or current) or an analog input channel (voltage or current) based on the enum SignalType. The InstrumentHandle specifies the instrument and the PhysicalChannel the analog channel used. The ChannelName is used as a unique identifier for the signal. The Enable or Disable functions must be used in order to achieve the desired output.

The channel maximal and minimal limits are set according to the signal definition while the input or output channel is created.

Optionally, type of the input channel can be selected with the InputTerminalConfig enumerator. The ReferencedSingleEnded configuration is selected by default. This setting is sufficient in most cases. However, special cases require a different terminal configuration. List of all the terminal configurations and their descriptions can be found in the Table 3-3.

<table>
<thead>
<tr>
<th>Configuration Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Device specific default – device manual must be consulted - this option is not recommended.</td>
</tr>
<tr>
<td>ReferencedSingleEnded</td>
<td>Voltage is measured with respect to the analog signal ground (AIGND pin)</td>
</tr>
<tr>
<td>NonReferencedSignleEnded</td>
<td>Voltage is measured with respect to the analog input sense (AISENSE pin)</td>
</tr>
<tr>
<td>Differential</td>
<td>Voltage is measured between the AI+ and AI-</td>
</tr>
<tr>
<td>PseudoDifferential</td>
<td>Voltage is measured between the AI+ and AI- with respect to the AIGND</td>
</tr>
</tbody>
</table>
Note that even though all the parameters are accessible, not all devices support all the settings. The device manual should be consulted first. An attempt to create a device unsupported channel will result in appropriate error info.

The function updates the SignalErrorInfo cluster with information whether the wrapped function failed or succeeded.

3.4.4 AIO_EnableSignal.vi

Input: \texttt{signal} AIO_signal in
Output: \texttt{signal} AIO_signal out

Icon: \includegraphics[width=0.1\textwidth]{icon.png}

Purpose: To start the signal generation at the analog output port.
To read the analog input port.

Action performed by the VI is selected according to the SignalType enumerator and are divided to two basic groups.

3.4.4.1 Analog output signal types:

The VI is used as a transition from the Standby state to the Generate Output state. The constant voltage and current outputs broadcast a single value, thus \texttt{frequency} and \texttt{NumberOfSamples} integers are neglected.

The function generation uses \texttt{Value} double64 as amplitude of the generated waveform. The \texttt{NumberOfSamples} integer defines the number of samples per cycle. Note that the maximum number of samples is 3000. The VI will check the desired number but won’t set higher than 3000. Frequency of the generated waveform is set by \texttt{frequency} integer.

After the VI execution, the \texttt{NumberOfSamples} integer will contain the number of samples per cycle generated by the function.

Note that the signal generation is continuous and doesn’t stop after a finite number of cycles by default. Use the SetAttributeValueInteger VI to achieve a finite number of cycles.
### 3.4.4.2 Analog input signal types:

The VI is used as a transition from the **Standby** state through the **Read/Capture** state to the **Standby** state again. Simple voltage and current outputs are reading single actual value from the input channel.

Other input channels read the number of samples specified by **NumberOfSamples** integer with the **frequency** of the sample clock defined in the signal cluster.

The RMS values are counted by the equation (1), which can be seen on the **Figure 3-14**.

\[
X_{\text{RMS}} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2}
\]

where: 
- \( n \) …number of samples
- \( x_i \) …sample value

**Figure 3-14**: RMS value count

The peak to peak value is counted as the measured signal maximum value minus the measured signal minimum value.

The **NumberOfSamples** after the VI execution contains the number of samples read from the signal.
3.4.5 AIO_DisableSignal.vi

Input: signal AIO_signal in
Output: signal AIO_signal out

Purpose: To set the output port value to zero.

3.4.5.1 Analog output signal types:

The VI is used as a transition from the Generate Output state to the Standby state. Using this function will write zero value at constant voltage and current signal types. Waveform generation channels are reset and set by the ResetSignal and SetupSignal VIs. This process will disrupt the continuous waveform generation and set the output to 0.

3.4.5.2 Analog input signal types:

The VI has no effect on signals with Measure capabilities.

3.4.6 AIO_ResetSignal.vi

Input: signal AIO_signal in
Output: signal AIO_signal out

Purpose: To reset the defined lines to an unused state.

The VI is used as a transition from the Standby state to the Idle state. Waveform generation signals must be replaced with constant signal channels to stop the continuous signal generation. Function chain can be seen on the Figure 3-15.

ClearChannel >> CreateAnalogOutputChannel >> WriteAnalogScalar >> ClearChannel

Figure 3-15: AO Reset function chain

The analog input channels don’t need any special treatment. A simple ClearChannel function call is sufficient for a signal reset.
3.4.7 AIO_Close.vi

Input:  
- UInt32 InstrumentHandle
- ErrorCluster Error in

Output:  
- ErrorCluster Error out

Icon:  

Purpose:  
To close the communication with an AIO instrument.

The VI is used as a transition from the Idle state to the Uninitialized state. The VI closes the session with the AIO instrument with the provided InstrumentHandle. The InstrumentHandle integer will be invalid after the communication disconnection. Calling AIO_Init.vi again will provide a different, unique InstrumentHandle.
3.4.8  AIO_GetAttributeValue.vi

Input:  

- **signal**  AIO_signal in
- **TypeDef**  attributeID

Output:  

- **signal**  AIO_signal out
- **~VAR~**  attributeValue out

Icon:

**Purpose:**  To retrieve the value of desired attribute.

A polymorphic VI [6] grouping multiple attribute types. The appropriate type is selected by the list box located under the icon. The types grouped in this VI and an example of usage is shown on the **Figure 3-9** in a DIO module section.

A list of handled attributes is in the section 6.2 of the Appendix A – List of Attributes. Some attributes are device specific (e.g. PXI_MANF_ID, PXI_MODEL_CODE, etc.) and using them with an unsupported device will result in an appropriate error message. The attribute handling and each attribute type is described in detail in the section 3.8 Attribute manipulation.

3.4.9  AIO_SetAttributeValue.vi

Input:  

- **signal**  AIO_signal in
- **TypeDef**  attributeID
- **~VAR~**  attributeValue

Output:  

- **signal**  AIO_signal out

Icon:

**Purpose:**  To set the value of desired attribute.

A polymorphic VI [6] grouping multiple attribute types. The appropriate type is selected by the list box located under the icon. The types grouped in this VI and an example of usage is shown on the **Figure 3-10** in DIO module section.

A list of handled attributes is in the section 6.2 of the Appendix A – List of Attributes. Some attributes are device specific (e.g. PXI_MANF_ID, PXI_MODEL_CODE, etc.) and using them with an unsupported device will result in an appropriate error message. The attribute handling and each attribute type is described in detail in the section 3.8 Attribute manipulation.
3.4.10 AIO_ReadCfg.vi

Input:  
- signal  AIO_signal in
- string  SearchKey
- string[]  Sections
- string[][]  Keys
- string[][][]  Values

Output:  
- signal  AIO_signal out

Purpose:  
To set the values from the configuration file to the desired signal.

The VI utilizes the FindValues.vi from the configuration section 3.2.6 to search for the SearchKey in “AI” (Analog Input) section and if no results are found, in the “AO” (Analog Output) section. If the key isn’t found in either section, an appropriate error message is passed. If the key is found, values are according to the HPF structure distributed to appropriate signal variables.

Figure 3-16: AIO_ReadCfg usage example

As can be seen on example shown on the Figure 3-11, the CFG_Open and The CFG_Read VIs must be called first in order to fill the Sections, Keys and Values arrays with a valid data.

Note that the function is designed to work with a HPF data order and will not work if the order isn’t kept. The data order used is noted in the VI as a comment.
3.4.11 AIO_GetDaqmxSession.vi

Input:  
\[ \text{signal} \quad \text{AIO\_signal in} \]

Output:  
\[ \text{signal} \quad \text{AIO\_signal out} \]
\[ \text{DAQmx task} \quad \text{DaqmxSession out} \]

Icon:

Purpose: To get the DAQmx session for advanced signal configuration.

The VI is used to retrieve the DAQmx session handle for advanced channel manipulation. The DaqmxSession is used with native DAQmx VIs for the advanced signal setup.
3.5 DC POWER SOURCE

Requirements covered: SYS_0050, SYS_0060, SYS_0110

DC power sources are necessary in the measurement automation for their high output voltage and current drive capabilities. Various types of the instrument triggering are used. Although immediate mode is sufficient for most cases, TTL or real time triggering is often required as well.

The main role of DCPWR instruments is power supply for the unit under test (UUT) and other necessary components with high power consumption.

3.5.1 DCPWR architecture

A signal cluster was developed in order to simplify the instrument control. The signal cluster is used as a medium for main signal values as well as critical signal settings. The signal cluster was created as a Strict Type Def. in the NI LabVIEW and can be seen on the Figure 3-17.

![Figure 3-17: DCPWR signal cluster](image)

```plaintext
enum SignalType

Double64 VoltageLevel
UInt32 InstrumentHandle

Double64 CurrentLimit
UInt32 ChannelID

enum TriggerSource

ErrorCluster SignalErrorInfo
```
**SignalType** enumerator is for future purposes only. Currently only a DC POWER SOURCE item is available. Future plan is to incorporate AC power supplies as well. The signal type is categorized as Source, which is described in the section 2.1.

**InstrumentHandle** integer represents the number of opened session with the initialized instrument. This number is the same for all the signals, which are using the same resource and becomes invalid after the instrument communication closure.

**VoltageLevel** double precision number is representing the demanded voltage output of the power source. Setting it higher than the instrument capability will result in an appropriate error message when used.

**CurrentLimit** double precision number is used to define the overcurrent protection (OCP) limits. The action upon achieving the current limit can be defined by the appropriate integer attribute (CURRENT_LIMIT_BEHAVIOR). The basic functionality is to trip the OCP and disable the output of power supply. Additionally, regulate mode can be used. The power supply in regulate mode will lower the voltage output to keep the current output at acceptable level (Current Limit).

**ChannelID** unsigned integer is used to define, which output to use at a multi output devices.

**TriggerSource** enumeration is used to determinate the type of triggering, which will be used. Note that not all types are supported by all instruments. Most common are the immediate, software and external triggers. A list of trigger sources can be found in the Table 3-4.

<table>
<thead>
<tr>
<th>Trigger Source Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMMEDIATE</td>
<td>Output is enabled by output enable function</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>Output is controlled by external analog value</td>
</tr>
<tr>
<td>SOFTWARE</td>
<td>Output is enabled by software trigger</td>
</tr>
<tr>
<td>TTL0..7</td>
<td>Output is enabled by TTL lines</td>
</tr>
<tr>
<td>ECL0, ECL1</td>
<td>Output is enabled by ECL lines</td>
</tr>
<tr>
<td>PXI_STAR</td>
<td>Output is enabled PXI specific trigger line</td>
</tr>
<tr>
<td>RTSI_0..5</td>
<td>Output is enabled by real-time line</td>
</tr>
</tbody>
</table>
Note that only signal in the immediate mode is enabled after the OutputEnable function call. Other triggering types require calling of the OutputEnable and Initiate functions. The initiate function will start the waiting for trigger. The output will be enabled only upon the arrival of the trigger start event.

**SignalErrorInfo** is a standardized NI LabVIEW error information cluster. It provides information whether the error occurred in a form of a status Boolean, error specific code number in a form of a code integer and error message as a source string.

The controls mentioned above are derived from a HPF and are sufficient for a complete control of DC power supplies.

**Figure 3-18: DCPWR control chain**

The **Figure 3-18** is showing a DCPWR control chain and represents the whole code segment needed for a basic DC power supply control. All functions are described in the upcoming sub-sections.
3.5.2  DCPWR_Init.vi

Input:  

- *string*  logicalName  
- ErrorCluster  Error in

Output:  

- UInt32  InstrumentHandle out  
- ErrorCluster  Error out

Icon:  

Purpose:  

To perform initialization of the communication with the DC Power source.

The VI is used as a transition from the **Uninitialized** state to the **Idle** state. Upon entering of the logicalName, which is used in the Measurement & Automation Explorer to map the SW components to the HW assets, the VI will attempt to make connection with the instrument. If the attempt is successful, a positive non-zero integer is returned in a form of Instrument Handle.

The Instrument Handle is unique for each logical name. However, it is the same for all the signals operating on this logical name. To differentiate between each signal on a same Instrument handle, the signal ChannelID integer is used. Note that most DC power sources have only one channel. The channel numbering begins at number 1.

Using this function will reset the device and mark the device as “initialized” so it cannot be initialized again without using the DCPWR_Close VI first.

3.5.3  DCPWR_SetupSignal.vi

Input:  

- *signal*  DCPWR_signal in

Output:  

- *signal*  DCPWR_signal out

Icon:  

Purpose:  

To perform the initial setup of the desired DC power source channel.

The VI is used as a transition from the **Idle** state to the **Standby** state. It sets the power source voltage level according to the VoltageLevel double precision number in the signal cluster. The current limit is set according to the definition in the signal and the trigger source to be used is defined. Note that setting any parameter outside the instruments capability range will result in an appropriate error message. The Enable or Disable VIs must be used in order to enable the desired output.
3.5.4 DCPWR_EnableSignal.vi

Input:  signal  DCPWR_signal in
Output: signal  DCPWR_signal out

Icon:  

Purpose:  To enable the output of the power supply.

The VI is used as a transition from the Standby state to the Generate Output state. The overvoltage and overcurrent protection state of the power supply is checked prior to the VI start. The VI will not be executed, if any of these protections were triggered.

When the immediate trigger type is used, the VI is used to close the circuit and start the signal generation at the output terminal.

When other trigger types are used, this VI only enables the power supply to generate at the output. However, the signal generation won’t start until the trigger condition is met. The Initiate and Abort waiting for trigger VIs must be used with different types of triggering.

Note that upon the output disable (e.g. trip action, force output disable etc.), signal generation won’t start even if the trigger condition is met, unless the Enable VI is used again.

Request to enable the output when already enabled, will result in an appropriate warning message.
3.5.5 DCPWR_DisableSignal_vi

Input: \textit{signal} DCPWR\_signal in
Output: \textit{signal} DCPWR\_signal out

Icon: DISABLE SIGNAL

Purpose: To disable the output of the power supply.

The VI is used as a transition from the \textit{Generate Output} state to the \textit{Standby} state. Utilize this VI to disable the output of the power supply. Output won’t be generated even if the trigger condition is met. To resume the signal generation, the Enable VI must be used again.

A request to disable the output when already disabled will result in an appropriate warning message.

3.5.6 DCPWR_ResetSignal_vi

Input: \textit{signal} DCPWR\_signal in
Output: \textit{signal} DCPWR\_signal out

Icon: RESET SIGNAL

Purpose: To reset the power supply to the default state.

The VI is used as a transition from the \textit{Standby} state to the \textit{Idle} state. Resetting the instrument will set the current limit and voltage level attributes to the signal specified values. Any changes made through setting attributes during the run will be disregarded.
3.5.7 DCPWR_Close.vi

Input:          UInt32     InstrumentHandle
                 ErrorCluster Error in

Output:         ErrorCluster Error out

Icon:          

Purpose:       To close the communication with the DC power supply.
                
The VI is used as a transition from the **Idle** state to the **Uninitialized** state. The function
        closes the session with the DCPWR instrument with the provided InstrumentHandle. The
        InstrumentHandle integer will be invalid after the communication disconnection. Calling
        the DCPWR_Init.vi again will provide a different, unique InstrumentHandle.

3.5.8 DCPWR_GetAttributeValue.vi

Input:          signal DCPWR_signal in
                 TypeDef attributeID

Output:         signal DCPWR_signal out
                -VAR- attributeValue out

Icon:          

Purpose:       To retrieve the value of desired attribute.
                
A polymorphic VI [6] grouping multiple attribute types. The appropriate type is
        selected by the list box located under the icon. The types grouped in this VI and an
        example of usage is shown on the **Figure 3-9** in the DIO module section.

A list of handled attributes is in the section 6.3 of the Appendix A – List of
Attributes. Some attributes are device specific and using them with an unsupported
device will result in an appropriate error message. The attribute handling and each
attribute type is described in detail in the section 3.8 Attribute manipulation.
3.5.9  DCPWR_SetAttributeValue.vi

**Input:**
- `signal`  
  DCPWR_signal in
- `TypeDef`  
  attributeID
- `-VAR-`  
  attributeValue

**Output:**
- `signal`  
  DCPWR_signal out

**Icon:**

**Purpose:**
To set the value of desired attribute.

A polymorphic VI [6] grouping multiple attribute types. The appropriate type is selected by the list box located under the icon. The types grouped in this VI and an example of usage is shown on the Figure 3-10 in the DIO module section.

A list of handled attributes is in the section 6.3 of the Appendix A – List of Attributes. Some attributes are device specific and using them with an unsupported device will result in an appropriate error message. The attribute handling and each attribute type is described in detail in the section 3.8 Attribute manipulation.

3.5.10  DCPWR_ReadCfg.vi

**Input:**
- `signal`  
  DCPWR_signal in
- `string`  
  SearchKey
- `string[]`  
  Sections
- `string[][]`  
  Keys
- `string[][][]`  
  Values

**Output:**
- `signal`  
  DCPWR_signal out

**Icon:**

**Purpose:**
To set the values from the configuration file to desired signal.

The VI utilizes the FindValues.vi from the configuration section 3.2.6 to search for the SearchKey in “DC_POWER” (DC power source) section and if no results are found, an appropriate error message is passed. If the key is found, values are according to the HPF structure distributed to appropriate columns.
As can be seen on the example shown on the Figure 3-19, the CFG_Open and The CFG_Read VIs must be called first in order to fill the Sections, Keys and Values arrays with a valid data.

Note that the function is designed to work with a HPF data order and will not work if the order isn’t kept. The data order used is noted in the VI as a comment.

### 3.5.11 DCPWR_TrippedQuery.vi

**Input:**
- signal DCPWR_signal in

**Output:**
- signal DCPWR_signal out

**Purpose:** To read the OVP and OVC status of the power supply.

The VI was created to check the overcurrent and overvoltage protections. The overvoltage protection is checked first, if it’s tripped, an appropriate error message is created and the output is disabled. If the overvoltage protection isn’t tripped, the VI will check if the overcurrent protection isn’t reporting a tripped state. If it is, the check is done again after one second. If the power supply still indicates the overcurrent protection active, an appropriate error message is created and the output is disabled to avoid damage to the instrument or the load connected to the power supply.
3.5.12 DCPWR_Initiate.vi

Input: signal DCPWR_signal in
Output: signal DCPWR_signal out

Icon: ![Initiate Waiting DCPWR]

Purpose: To initiate a waiting for trigger.

The VI tells the instrument to wait for trigger. When the trigger condition arrives, the instrument will start to generate output. The VI has no effect on a power supply in the immediate mode.

3.5.13 DCPWR_Abort.vi

Input: signal DCPWR_signal in
Output: signal DCPWR_signal out

Icon: ![Abort Waiting DCPWR]

Purpose: To abort waiting for trigger.

The VI tells the instrument to stop waiting for a trigger. When the trigger condition arrives, the instrument won’t react anymore. The VI has no effect on a power supply in the immediate mode.

3.5.14 DCPWR_SendSoftwareTrigger.vi

Input: signal DCPWR_signal in
Output: signal DCPWR_signal out

Icon: ![Send SW Trigger DCPWR]

Purpose: To send a software trigger to the power supply.

The VI was created to send a software trigger to the power supply in the software triggering mode. The software trigger has no effect on power supply, which is using other type of triggering.
3.5.15  DCPWR_WriteInstrumentData.vi

Input:  
signal  DCPWR_signal in
string  WriteBuffer

Output:  
signal  DCPWR_signal out

Purpose:  To send SCPI command to the instrument.

The VI will send a command written in the WriteBuffer string. Writing SCPI commands is implemented to allow the developer to access all functions and settings of the power supply instrument.

3.5.16  DCPWR_ReadInstrumentData.vi

Input:  
signal  DCPWR_signal in
UInt32  numberOfBytesToRead (optional)

Output:  
signal  DCPWR_signal out
string  ReadBuffer

Purpose:  To read data sent from the instrument.

The VI will read a defined number of bytes from the instrument. The default number of bytes read is 256, but the value can be changed by the optional numberOfBytesToRead unsigned integer. The data read from instrument are answers to the queries sent by the WriteInstrumentData VI.
3.6 SERIAL INTERFACE

Requirements covered: SYS_0050, SYS_0060, SYS_0120

The serial module is designed to facilitate the communication over the serial interface. For this purpose a variety of utility functions were implemented. A serial communication is an undividable part of all automated testing solutions. The communication with the UUT and custom test equipment is done mainly through the RS-232 or RS-485 interfaces.

3.6.1 SERIAL architecture

The main difference from other modules is in the instrument-signal connection. One instrument is tied with a specific COM port. This means, that only one signal can operate on one instrument. A signal cluster was developed in order to simplify the instrument control. The signal cluster is used as a medium for main signal values as well as critical signal settings. The signal cluster was created as a Strict Type Def. in the NI LabVIEW and can be seen on the Figure 3-20.

![Figure 3-20: SERIAL signal cluster](image)

```plaintext
enum SignalType
UInt32 InstrumentHandle
Int32 Baudrate
enum Parity
UInt16 DataBits
enum StopBits
enum TerminationChar
enum FlowControl
UInt8[] Message
UInt32 Bytes I/O
enum TriggerSource
```

Figure 3-20: SERIAL signal cluster
**SignalType** enumerator is used to determine the type of a communication protocol used by the signal. A list of available signal types and their description can be found in the **Table 3-5**. The signal type is categorized as Communication, which is described in section 2.5.

<table>
<thead>
<tr>
<th>Signal Type Name</th>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS485</td>
<td>40</td>
<td>Standard 4-wire RS-485 communication</td>
</tr>
<tr>
<td>RS485_2</td>
<td>43</td>
<td>2-wire RS-485 communication</td>
</tr>
<tr>
<td>RS485_2_DTR_CTRL</td>
<td>42</td>
<td>2-wire RS-485, flow control: Data terminal control device</td>
</tr>
<tr>
<td>RS485_2_DTR_ECHO</td>
<td>41</td>
<td>2-wire RS-485, flow control: Data terminal slave device</td>
</tr>
<tr>
<td>RS232</td>
<td>130</td>
<td>Standard RS-232 communication</td>
</tr>
<tr>
<td>RS232_DCE</td>
<td>129</td>
<td>RS-232 communication, flow control: Data terminal master</td>
</tr>
<tr>
<td>RS232_DTE</td>
<td>128</td>
<td>RS-232 communication, flow control: Data terminal circuit-terminating device</td>
</tr>
<tr>
<td>NOT_DEFINED</td>
<td>-1</td>
<td>Undefined state</td>
</tr>
</tbody>
</table>

**InstrumentHandle** integer represents the number of opened session with the initialized instrument. This number is the same for all signals, which are using the same resource and becomes invalid after the instrument communication closure.

**Baudrate** integer is used to determine the baud rate used in the communication. Setting the baud rate in a signal cluster control panel is done through a ring control with most used baud rate values. However, the control can accept undefined values for custom control solutions.
**Parity** enumeration is used to set the communication parity. The Table 3-6 is showing available parity types and their description.

**Table 3-6: SERIAL parity types**

<table>
<thead>
<tr>
<th>Parity Name</th>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>10</td>
<td>Parity bit isn’t used</td>
</tr>
<tr>
<td>ODD</td>
<td>11</td>
<td>Parity bit used to alter to the odd number</td>
</tr>
<tr>
<td>EVEN</td>
<td>12</td>
<td>Parity bit used to alter to the even number</td>
</tr>
</tbody>
</table>

**DataBits** unsigned integer is used to mark the number of bits carrying the data in the communication. Supported values are 5, 6, 7 and 8 data bits. Other values are not supported.

**StopBits** enumeration defines the number of stop bits used after the data bits transfer. A list of supported values and their description is in the Table 3-7.

**Table 3-7: SERIAL supported stop bits**

<table>
<thead>
<tr>
<th>Stop Bit Name</th>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>One stop bit</td>
</tr>
<tr>
<td>1.5</td>
<td>22</td>
<td>One and half stop bits</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
<td>Two stop bits</td>
</tr>
</tbody>
</table>

**TerminationChar** enumeration is a list of possible values for the termination character in the serial communication. Values in the enumeration can be seen in the Table 3-8.

**Table 3-8: SERIAL supported termination characters**

<table>
<thead>
<tr>
<th>Termination Char Name</th>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>0</td>
<td>No termination character, send or write requested number of bytes</td>
</tr>
<tr>
<td>CR</td>
<td>13</td>
<td>Carriage return (ASCII - 0x0D)</td>
</tr>
<tr>
<td>LF</td>
<td>10</td>
<td>Line feed (ASCII - 0x0A)</td>
</tr>
</tbody>
</table>
FlowControl enumeration determinates the method of data flow control in the serial communication. The Table 3-9 is showing a list of possible values.

Table 3-9: SERIAL flow control types

<table>
<thead>
<tr>
<th>Flow Control Name</th>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>30</td>
<td>No flow control</td>
</tr>
<tr>
<td>XON – XOFF</td>
<td>31</td>
<td>SW: receiver sends XOFF and XON messages</td>
</tr>
<tr>
<td>RTS – CTS</td>
<td>32</td>
<td>HW: Request to send – Clear to send</td>
</tr>
<tr>
<td>DTR – DSR</td>
<td>33</td>
<td>HW: Data terminal ready – Data set ready</td>
</tr>
</tbody>
</table>

Message array is used as a SW buffer for the data to be sent or the data just received. The array is of 8bit unsigned integer type. The utility VI SERIAL_ConvertStringToUInt8 must be used in order to transfer a string message. The data received are in a UInt8 format as well. In order to convert them to a human readable format, the utility VI SERIAL_ConvertUInt8ToString must be used. An example is shown on the Figure 3-21.

Figure 3-21: SERIAL Message conversion

Bytes I/O integer has a dual purpose. Before the Write and Read signal VIs, it indicates how many bytes should be written or read. After the VIs execution, the number of actually written and read bytes is marked in this signal variable.

SignalErrorInfo is a standardized NI LabVIEW error information cluster. It provides information whether the error occurred in a form of a status Boolean, error specific code number in a form of a code integer and error message as a source string.
The controls mentioned above are derived from a HPF and are sufficient for a complete control over the serial communication port.

Figure 3-22: SERIAL control chain

The Figure 3-22 is showing a SERIAL control chain and represents the whole code segment needed for a basic communication over the serial interface. All functions are described in the upcoming sub-sections.
3.6.2 SERIAL_Init.vi

Input:  
\begin{itemize}
  \item \textbf{string} \hspace{1em} \textbf{logicalName}
  \item \textbf{ErrorCluster} \hspace{1em} \textbf{Error in}
\end{itemize}

Output:  
\begin{itemize}
  \item \textbf{UInt32} \hspace{1em} \textbf{InstrumentHandle out}
  \item \textbf{ErrorCluster} \hspace{1em} \textbf{Error out}
\end{itemize}

Icon:  

Purpose:  
To perform the initialization of the communication port.

The VI is used as a transition from the \textbf{Uninitialized} state to the \textbf{Idle} state. Upon entering of the \textbf{logicalName}, which is used in the Measurement & Automation Explorer to map the SW components to the HW assets, the VI will attempt to make the connection with the communication port. If the attempt is successful, a positive non-zero integer is returned in a form of the \textbf{InstrumentHandle}.

The \textbf{InstrumentHandle} is unique for each logical name. Note that, unlike other LICT modules, only one signal may operate on one logical name. However, one instrument can have multiple communication ports, thus multiple logical names can access various COM ports on a single instrument.

Using this function will reset COM port settings and mark the port as “initialized” so it cannot be initialized again without using the SERIAL_Close VI first.

3.6.3 SERIAL_SetupSignal.vi

Input:  
\begin{itemize}
  \item \textbf{signal} \hspace{1em} \textbf{SERIAL\_signal in}
\end{itemize}

Output:  
\begin{itemize}
  \item \textbf{signal} \hspace{1em} \textbf{SERIAL\_signal out}
\end{itemize}

Icon:  

Purpose:  
To perform the initial setup of the desired communication port.

The VI is used as a transition from the \textbf{Idle} state to the \textbf{Standby} state. All the settings in the serial signal cluster are applied and the communication port is configured accordingly. Not all the settings available are supported by all the instruments and using unsupported settings will result in an appropriate error message.
3.6.4 SERIAL_WriteSignal.vi

Input: signal SERIAL_signal in
Output: signal SERIAL_signal out

Icon:

Purpose: To write a defined number of bytes to the transmit buffer.

The VI is used as a transition from the Standby state to the Write state and to the Standby state again. The VI will attempt to write a number of bytes defined by the Bytes I/O integer but no more than to the location of a termination character (Carriage Return or Line Feed). The write is done to the Transmit (Tx) buffer. The number of bytes written is stored in the Bytes I/O signal variable before the end of the VI execution.

The message written is in a form of UInt8 array, where each number represents a byte to be written. In order to send human readable messages, a string message must be converted by the SERIAL_ConvertStringToUInt8.vi.

3.6.5 SERIAL_ReadSignal.vi

Input: signal SERIAL_signal in
Output: signal SERIAL_signal out

Icon:

Purpose: To read a defined number of bytes from the receive buffer.

The VI is used as a transition from the Standby state to the Read state and to the Standby state again. The VI will attempt to read a number of bytes defined by the Bytes I/O integer. Reading is done from the Receive (Rx) buffer. The number of bytes read is stored in the Bytes I/O signal variable before the end of the VI execution.

The message read is in a form of UInt8 array, where each number represents a byte received. In order to receive human readable messages in a string form, the SERIAL_ConvertUInt8ToString.vi must be used.
3.6.6 SERIAL_ResetSignal.vi

Input: signal SERIAL_signal in
Output: signal SERIAL_signal out

Icon: SERIAL

Purpose: To reset the communication port settings to the default state.

The VI is used as a transition from the Standby state to the Idle state. Resetting the communication port settings will set the COM port to a default state, which is instrument specific. Any changes made through setting attributes during the run will be disregarded.

3.6.7 SERIAL_Close.vi

Input: UInt32 InstrumentHandle
        ErrorCluster Error in
Output: ErrorCluster Error out

Icon: SERIAL

Purpose: To close the session with the communication port.

The VI is used as a transition from the Idle state to the Uninitialized state. The VI closes the session with the communication port with provided InstrumentHandle. The InstrumentHandle integer will be invalid after the session closure. Calling the SERIAL_Init.vi again will provide a different, unique InstrumentHandle.
3.6.8 SERIAL_GetAttributeValue Value.vi

Input:  
- signal SERIAL_signal in
- TypeDef attributeID

Output:  
- signal SERIAL_signal out
- VAR attributeValue out

Icon:

Purpose:  
To retrieve the value of desired attribute.

A polymorphic VI [6] grouping multiple attribute types. The appropriate type is selected by the list box located under the icon. The types grouped in this VI and an example of usage is shown on the Figure 3-23.

Figure 3-23: SERIAL GetAttributeValue Value polymorphic VI

A list of handled attributes is in the section 6.4 of the Appendix A – List of Attributes. Some attributes are device specific and using them with an unsupported device will result in an appropriate error message. The attribute handling and each attribute type is described in detail in the section 3.8 Attribute manipulation.

In addition to the standard attribute types, custom VISA attributes were created to access most common NI VISA specific attributes. Custom type definition was created as well as SERIAL_GetAttributeVISA.vi.
3.6.9 SERIAL_SetAttributeValue.vi

Input: 
- signal SERIAL_signal in
- TypeDef attributeID
- ~VAR~ attributeValue

Output: 
- signal SERIAL_signal out

Icon:

Purpose: To set the value of desired attribute.

A polymorphic VI [6] grouping multiple attribute types. The appropriate type is selected by the list box located under the icon. The types grouped in this VI and an example of usage is shown on the Figure 3-24.

Figure 3-24: SERIAL SetAttributeValue polymorphic VI

A list of handled attributes is in the section 6.4 of the Appendix A – List of Attributes. Some attributes are device specific and using them with an unsupported device will result in an appropriate error message. The attribute handling and each attribute type is described in detail in the section 3.8 Attribute manipulation.

In addition to the standard attribute types, custom VISA attributes were created to access most common NI VISA specific attributes. Custom type definition was created as well as SERIAL_SetAttributeVISA.vi.
3.6.10 SERIAL_ReadCfg.vi

Input:  
- `signal`  
- `string`  
- `string[]`  
- `string[][]`  
- `string[][][]`

Output:  
- `signal`  
- `string`  
- `string[]`  
- `string[][]`  
- `string[][][]`

Purpose: 
To set values from the configuration file to the desired signal.

The VI utilizes the FindValues.vi from the configuration section 3.2.6 to search for the SearchKey in “Serial” (Serial communication) section and if no results are found, an appropriate error message is passed. If the key is found, values are according to the HPF structure distributed to appropriate columns.

![Diagram](read cfg v.png)

**Figure 3-25: SERIAL ReadCfg usage example**

As can be seen on the example shown on the Figure 3-25, the CFG_Open and CFG_Read VIs must be called first in order to fill the Sections, Keys and Values arrays with a valid data.

Note that the function is designed to work with a HPF data order and will not work if the order isn’t kept. The data order used is noted in the VI as a comment.
3.6.11 SERIAL_ConvertStringToUInt8.vi

**Input:**
- `signal` SERIAL_signal in
- `string` StringInput

**Output:**
- `signal` SERIAL_signal out

**Icon:**

**Purpose:** To convert the input string to the UInt8 array.

The VI was created to convert a string input to its ASCII number representation. Each character as its ASCII number representation is stored in own field. The array item with index zero is representing the first character in the string. An example of the conversion can be found on the *Figure 3-26*.

![Figure 3-26](image)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>72</td>
<td>101</td>
<td>108</td>
<td>108</td>
<td>111</td>
<td>32</td>
<td>119</td>
<td>111</td>
<td>114</td>
<td>108</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

*Figure 3-26: SERIAL String to array conversion*

An example of the VI usage along with the Read and Write VIs can be found on the *Figure 3-21*.

3.6.12 SERIAL_ConvertUInt8ToString.vi

**Input:**
- `signal` SERIAL_signal in
- `UInt32` outStrLen (optional)

**Output:**
- `signal` SERIAL_signal out
- `string` StringOutput

**Icon:**

**Purpose:** To convert the input UInt8 array to the string representation.

The VI was created to convert the received UInt8 message to a human readable string. The array item with index zero is representing the first character (ASCII form) in a final string. The conversion found on the *Figure 3-26* represents the same process but in a reverse direction. An example of the VI usage along with the Read and Write VIs can be found on the *Figure 3-21*. 

77
3.6.13 SERIAL_GetVISAResource.vi

Input: signal SERIAL_signal in
Output: signal SERIAL_signal out
        VISARef VISAResource out

Icon: SERIAL

Purpose: To pass the VISA resource session for the advanced instrument control.

The VISA session handle have to be retrieved in order to allow the access to advanced VISA communication settings. This VI allows developer to create solutions outside the pre-made VIs, while maintaining the signal architecture.

3.6.14 SERIAL_FlushOutQueue.vi

Input: signal SERIAL_signal in
Output: signal SERIAL_signal out

Icon: SERIAL

Purpose: To clear the transmit buffer.

The VI was created to clear the entire transmit buffer of the signal defined communication port. Note that all the data are dismissed (not sent).

3.6.15 SERIAL_FlushInQueue.vi

Input: signal SERIAL_signal in
Output: signal SERIAL_signal out

Icon: SERIAL

Purpose: To clear the receive buffer.

The VI was created to clear the entire receive buffer of the signal defined communication port. Note that all the data are dismissed (not saved).
3.7 ERROR HANDLING

In order to properly manipulate with the errors in program, error handling based on the function return values was created. Each module signal cluster was equipped with the SignalErrorInfo cluster. This approach was selected to extract and separate the errors of each signal. E.g. an error in the DIO module will have no effect on the DCPWR module and vice versa.

![SignalErrorInfo cluster](image)

**Figure 3-27: SignalErrorInfo cluster**

**Status** Boolean variable indicates whether the error occurred or not. Based on this variable, the error state is detected in all functions and VIs. The TRUE value indicates an “error” and the value FALSE a “no error” state.

**Code** integer stores the error code passed from the function. Values below zero indicate errors and values above zero indicate warnings.

**Message** string contains the error message passed by the ErrorMessage function, which was created in the IAL, to assign the error message to the appropriate error code.

![SignalErrorInfo cluster with values](image)

**Figure 3-28: Error occurred**

Two basic approaches to error were discussed:

1. An error doesn’t disrupt the execution of VIs after the occurrence
2. An error forces to skip all VIs (or functions) after the occurrence
The second approach was selected due to its better effect on the debugging. E.g. if the instrument cannot be initialized, it is no point in trying to create any channels on it.

Note that the exception to this approach was created in the reset and close functions, which are executed even after an error occurrence. Otherwise the instrument wouldn’t be able to leave the error state.

An error is evaluated from the return value of each function wrapper called. The error handling sub-VI was created to call the IAL ErrorMessage function. In case of error detection, the sub-VI bundles the appropriate error info into the SignalErrorInfo cluster. An example can be seen at the Figure 3-29.

![Figure 3-29: Error handling sub-VI](image)

Note that the sub-VI has a case structure, which in case of an error disables the sub-VI in order to conserve the error info from previous function call. The reason not to evaluate error is the fact that the function wrapper isn’t called and the data from its output are zeroes, which in the case of function return indicates an OK status.

An error cluster can be unbundled from the signal at any time to pass the error info to other components or the user. An example can be seen on the Figure 3-30.

![Figure 3-30: SignalErrorInfo unbundling](image)
3.8 ATTRIBUTE MANIPULATION

Requirements covered: SYS_0070

This section is dedicated to the description of VIs created for the advanced instrument control and settings. Each instrument has set of attributes defined by the IVI standard [5] and a set of specific attributes, which were created additionally. They are divided by the input and output value types into the categories as shown in the Table 3-10. A clarification of numeric data types is present in the Appendix C.

<table>
<thead>
<tr>
<th>Attribute Data type</th>
<th>Attribute Name</th>
<th>Terminal Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>UInt8</td>
<td>Boolean</td>
<td></td>
</tr>
<tr>
<td>Int32</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>Double64</td>
<td>Real</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>

A different VI was created for attribute setting and retrieval. Some attributes can have a read-only value. However, all attribute IDs were kept in both VIs. If the user tries to write in a read-only attribute, the VI will fail and an appropriate error message is transferred into the SignalErrorInfo cluster.

Attribute IDs are created as type definitions to facilitate adding of new attributes (in case of a driver update) and removing an obsolete or unsupported attributes. Each module has its own set of type definitions, because each module has unique capabilities and the set of attributes is significantly different.

To facilitate the attribute handling even further, all VIs with data setting capabilities were grouped into the SetAttributeValue polymorphic VI and the user can select the desired attribute type with a list box. VIs with data retrieving capabilities were grouped into the GetAttributeValue polymorphic VI. The main advantage of a polymorphic VI lies in a unified interface for multiple sub-VIs. The input or output type of the attribute value changes according to the selected attribute type. The attribute type affects also what type definition will be provided. Each function has appropriate help, which lists attribute IDs and if necessary their values and purpose.
3.8.1 GetAttributeValue Value Boolean.vi

Input:  
signal Prefix_signal in
TypeDef attributeID

Output:  
signal Prefix_signal out
UInt8 attributeValue out

Icon:  
(module name is for illustration purpose only)

Purpose:  
To retrieve the value of Boolean type attributes.

The VI is getting the value of the IVI specific attribute, which is defined by the attributeID. To facilitate the pass of a correct attributeID, instrument specific type definitions were made. A boolean attribute value is represented by the integer. A value 0 stands for false and 1 for true.

3.8.2 GetAttributeValue Value Integer.vi

Input:  
signal Prefix_signal in
TypeDef attributeID

Output:  
signal Prefix_signal out
Int32 attributeValue out

Icon:  
(module name is for illustration purpose only)

Purpose:  
To retrieve the value of integer type attributes.

The VI is getting the value of the IVI specific attribute, which is defined by the attributeID. To facilitate the pass of a correct attributeID, instrument specific type definitions were made. Unless the integer attribute value defines the value itself (e.g. number of bytes in queue, etc.), it is tied with an enumeration, which is defined in the function help.
3.8.3 GetAttributeValue Real.vi

Input:  

- **signal**  Prefix_signal in
- **TypeDef** attributeID

Output:  

- **signal**  Prefix_signal out
- **Double64** attributeValue out

Icon:  

(module name is for illustration purpose only)

Purpose:  

To retrieve the value of real type attributes.

The VI is getting the value of the IVI specific attribute, which is defined by the attributeID. To facilitate the pass of a correct attributeID, instrument specific type definitions were made. The real attribute values are passed in a form of a double precision number and they are tied with the value directly (e.g. current limit, timeout, etc.)

3.8.4 GetAttributeValue String.vi

Input:  

- **signal**  Prefix_signal in
- **TypeDef** attributeID

Output:  

- **signal**  Prefix_signal out
- **string** attributeValue out

Icon:  

(module name is for illustration purpose only)

Purpose:  

To retrieve the value of string type attributes.

The VI is getting the value of the IVI specific attribute, which is defined by the attributeID. To facilitate the pass of a correct attributeID, instrument specific type definitions were made. The string attributes are used to store data and descriptions, which cannot be stored in an enumeration or any other form. Note that a 1024 byte size array is allocated for a response message.
### 3.8.5 SetAttributeValueBoolean.vi

**Input:**
- `signal` Prefix_signal in
- `TypeDef` attributeID
- `UInt8` attributeValue

**Output:**
- `signal` Prefix_signal out

**Icon:**
![module name is for illustration purpose only](image)

**Purpose:**
To set the value of Boolean type attributes.

The VI is setting the value of the IVI specific attribute, which is defined by the attributeID. To facilitate the pass of a correct attributeID, instrument specific type definitions were made. The boolean attribute value is represented by the integer. The value 0 stands for false and 1 for true.

### 3.8.6 SetAttributeValueInteger.vi

**Input:**
- `signal` Prefix_signal in
- `TypeDef` attributeID
- `Int32` attributeValue

**Output:**
- `signal` Prefix_signal out

**Icon:**
![module name is for illustration purpose only](image)

**Purpose:**
To set the value of integer type attributes.

The VI is setting the value of the IVI specific attribute, which is defined by the attributeID. To facilitate the pass of a correct attributeID, instrument specific type definitions were made. Unless the integer attribute value defines the value itself (e.g. number of bytes in queue, etc.), it is tied with an enumeration, which is defined in the function help.
3.8.7  SetAttributeValue Real.vi

Input:
- **signal**  Prefix_signal in
- **TypeDef**  attributeID
- **Double64**  attributeValue

Output:
- **signal**  Prefix_signal out

Icon:  

Purpose: To set the value of real type attributes.

The VI is setting the value of the IVI specific attribute, which is defined by the attributeID. To facilitate the pass of a correct attributeID, instrument specific type definitions were made. The real attribute values are passed in a form of a double precision number and they are tied with the value directly (e.g. current limit, timeout, etc.)

3.8.8  SetAttributeValue String.vi

Input:
- **signal**  Prefix_signal in
- **TypeDef**  attributeID
- **string**  attributeValue

Output:
- **signal**  Prefix_signal out

Icon:  

Purpose: To set the value of string type attribute.

The VI is setting the value of the IVI specific attribute, which is defined by the attributeID. To facilitate the pass of a correct attributeID, instrument specific type definitions were made. The string attributes are used to store data and descriptions, which cannot be stored in an enumeration or any other form.
4. PROOF OF CONCEPT

Requirements covered: SYS_0040, SYS_0080, SYS_0150

An example project was created in order to prove the LIC&T function. This example was created to support all coding standards in the NI LabVIEW and may function as base for all other projects created with the LIC&T.

4.1 STATION SETUP INFORMATION

The example was created on fully equipped test station. Instruments and their logical names are noted in Table 4-1.

<table>
<thead>
<tr>
<th>Instrument Type</th>
<th>Logical Name</th>
<th>Supported Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>NI PXI-6229</td>
<td>MIO_1A</td>
<td>AIO</td>
</tr>
<tr>
<td></td>
<td>MIO_2A</td>
<td></td>
</tr>
<tr>
<td>NI PXI-8431/8</td>
<td>SERIAL_1A</td>
<td>SERIAL</td>
</tr>
<tr>
<td></td>
<td>SERIAL_2A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SERIAL_3A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SERIAL_4A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SERIAL_5A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SERIAL_6A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SERIAL_7A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SERIAL_8A</td>
<td></td>
</tr>
<tr>
<td>AMETEK RFP-D1065-5A1</td>
<td>DCPS_3A</td>
<td>DCPWR</td>
</tr>
</tbody>
</table>

Note that multiple logical names can be used for a single instrument. In case of a multi-purpose card as NI PXI-6229, logical names are dual to support all modules on a single card. In case of the serial communication card NI PXI-8431/8, each of the eight COM ports is mapped to a single logical name.
4.2 UUT INFORMATION

Real testing unit was borrowed for the example in order to achieve best testing results. The Enhanced Ground Proximity Warning System was placed as a formal UUT, which was communicating with the test panel created as an example.

As mentioned in [4] EGPWS product specification, the purpose of the EGPWS is to help prevent accidents caused by controlled flight into terrain or by severe wind shear. The system achieves this objective by accepting a variety of aircraft analog and arinc signals as input, applying algorithms and warning the crew by both visual and aural alert messages.

4.2.1 Signals used

The example was using some of UUT input signals [4], provided them with an appropriate stimulus and retrieved the status information from unit by the serial communication port. The selection of signals used in the example is noted in the sub-sections below along with their characteristics and module, which was used.

4.2.1.1 Primary Power

Module: DCPWR
Purpose: To supply UUT with power.
Signal characteristics:
   Nominal Input: 28 VDC
   Normal Voltage Range: 22.0 to 30.3 VDC
   Power Requirements: 9 Watts

4.2.1.2 Engine Torque

Module: DIO
Purpose: To determinate the plane engine torque.
Signal Characteristics:
   Input Voltage Range: ±5 VDC
   Input type: differential
Note: Since DIO module is used for this input – only 0 VDC and +5 VDC values are available for testing.
4.2.1.3 Barometric Altitude

Module: AIO
Purpose: To determinate device barometric altitude

Signal Characteristics:

- Input Voltage Range: -0.18 VDC to +15 VDC
- Input type: differential

Note: +15 VDC is outside the range provided by the AIO station instrumentation. Maximum Voltage is limited to +10 VDC.

4.2.1.4 Radio Altitude

Module: AIO
Purpose: To determinate device radio altitude

Signal Characteristics:

- Input Voltage Range: -0.18 VDC to +37.6 VDC
- Input type: differential

Note: +37.6 VDC is outside the range provided by the AIO station instrumentation. Maximum Voltage is limited to +10 VDC.

4.2.1.5 RS-232 Maintenance port

Module: SERIAL
Purpose: To handle communication with the EGPWS unit maintenance port.

Characteristics:

- Baud rate: 19,200
- Parity: None
- Data bits: 8
- Stop bits: 1
4.3 EXAMPLE FUNCTION

The main function of the example was to create a modular control program as guidance to the future use of the LICT and prove that the signal abstraction is a viable method of instrument control.

The UUT and signals used to control it are specifically selected to match all the LICT modules and to provide real life use case. Even though all modules were properly integrated while developed, extensive testing of simultaneous module use was needed. Additionally, many useful utility functions were created along with the example implementation to facilitate development of future projects.

Figure 4-1: Example GUI

A simple GUI was created for the instrument control in order to provide the UUT with a correct stimulus. In addition, communication with the UUT maintenance port
was established and allowed user to read the status of each stimulated input port and to send the UUT commands and queries.

Each signal generated by the test station is looped back to the input port for self-test purposes. An error info cluster is located at the bottom of the screen and provide user with actual information about occurred errors.

The example is working with the LICT configuration file management module to prove the design functionality. A HPF with the definition of each signal was created. Using the HPF along the with configuration management VIs greatly simplifies the setting of all signals. All basic and most of the advanced signal settings are derived from the HPF. Using this layout of the configuration file is recommended for the best results.

The example itself is divided into 3 modules: Initialization, Main and Close. The structure can be seen on the Figure 4-2.

![Example folder structure]

**Figure 4-2: Example folder structure**

All modules and their purpose are described in the Table 4-2. The “Documentation” folder contains the UUT documentation and the “Configuration” folder a HPF with signal descriptions.
Table 4-2: Example module description

<table>
<thead>
<tr>
<th>Module name</th>
<th>Actions</th>
</tr>
</thead>
</table>
| Initialization.vi | Read configuration file  
                   | Insert data from configuration file to signal clusters  
                   | Initialize required instruments  
                   | Setup all signals |
| Main.vi           | Allow user to Enable and Disable signals  
                   | Allow user to change attributes of each signal |
| Close.vi          | Disable all signals  
                   | Reset all signals  
                   | Close communication with all instruments |

Each Module is described in detail in the sub-sections below. All actions are clarified and examples of use of the basic VIs are shown.
4.3.1 Initialization

Reading of the configuration file and the signal setup is done in the initialization VI. All the steps are done sequentially in the stacked sequence control. The Figure 4-3 and Figure 4-4 are showing the setting of the signals with the data from the HPF.

**Figure 4-3: Configuration file reading**

The configuration file is opened only to read the data and then it’s immediately closed. The data from the configuration file are stored in the appropriate local variables.

**Figure 4-4: Setting signals with configuration data**

Data from the configuration file are sent to all signals, which are set by the appropriate ReadCfg VI.
The initialization of instruments along with the setup of each signal is done next. Each used logical name has to be initialized, but multiple signals can be active on one logical name. This can be seen on the Figure 4-5.

**Figure 4-5: Instrument initialization and signal setup**

DIO signals EngineTorque and EngineTorque_LB are active on a single logical name MIO_2A. This means that they have the same InstrumentHandle. Also the error information from the instrument initialization is sent to both signals. Signal Setup is done after a successful instrument initialization.

The Figure 4-5 also shows the initialization and setup of the serial interface used in this example for the communication with the UUT. The setup incorporated a SERIAL_SetAttributeValue polymorphic VI, which was used to set the END_MODES_FOR_READS VISA attribute value to 0 – none. This attribute allows the program to read a whole input buffer, rather than just one line of text.

All the signal clusters are sent to the VI output at the end of the stacked sequence. Note that this example isn’t using global variables to store signal clusters, but the use of global variables in other, more complex, solutions is advised.
4.3.2 Main

This module is created to support user interaction. All the actions done in this module are processed by an event structure. Control buttons are created in order to allow the user to perform enable and disable actions on the signals.

![Discrete Input/Output Controls Diagram]

**Figure 4-6: Discrete IO controls**

A change of value in signal switches is processed by the event structure and depending on the control new value, a specific action (Signal Enable or Disable) is done. An example can be seen on the **Figure 4-7**.

![Event Handling Diagram]

**Figure 4-7: Event handling**

The **Figure 4-7** also shows the unbundling of the error information, which is then passed to the main error window. These solutions were taken from integration VIs created for each module. The reusability of the solution was therefore confirmed.
The initialization and close VIs are called from this VI. The switch, which is controlling the action, can be seen on the Figure 4-8.

![Initialization/Close switch](image)

**Figure 4-8: Initialization/Close switch**

All the AIO and DIO signals have a loopback installed in the test solution. This loopback is used for self test purposes. The data read from the barometric altitude loopback are inserted into the plot.

### 4.3.3 Close

This VI is used to reset all signals and close the communication with all instruments, which were used in this example. The whole process can be seen on the Figure 4-9. This VI is called from the main VI either by the Init/Close button or upon the application closure.

![Signal reset and instrument close](image)

**Figure 4-9: Signal reset and instrument close**

The case structure is used to determine whether the signals are operating on the same instrument. If they are, only one close is needed, if they aren’t, both instruments are closed. The Figure 4-9 is showing the case, in which both the instruments will be closed.
5. CONCLUSION

The beginning of the thesis is dedicated to the project preparations. Each aspect of the development had to be carefully investigated prior to the main implementation. National Instruments LabVIEW was chosen as the development environment by technical leadership review board based on the quality function deployment (QFD) table. The choice of the driver layer was based on the requirement to achieve a compromise between the hardware interchangeability and the number of control possibilities. This compromise was found in the Honeywell Instrument Abstraction Driver layer. This driver layer is based on the IVI standard and offered a possibility to develop a single solution for a variety of instrumentation.

Project requirements were established at the end of the theoretical part of the thesis. These requirements were derived from the company survey and were discussed with the Honeywell project leadership. All requirements are covered by this thesis and full requirement traceability is provided by the requirement traceability matrix, which is included in the appendix B.

Next chapter contains the description of the signal abstraction layer and its architecture. The signal abstraction was created to simplify the instrument control. It allows the developer to create a software control solution, which is tied directly with a signal flow diagram describing the hardware solution. The signals are divided, by the capabilities of the instruments they’re using, into: Source, Measurement, Connection, and Communication.

The main chapter was focusing on the development of the LICT in the NI LabVIEW. Each module has its own section with an architecture and detailed description of every virtual instrument. Integration VIs were created in order to integrate all the implemented software. Those VIs verified the function of each module and all its parts. A section, which is describing the error handling, is present at the end of the chapter along with the section dedicated to manipulation with the signal attributes.
The purpose of the last chapter was to demonstrate the project capabilities and to verify its function in a real life use case. The Enhanced Ground Proximity Warning System unit was wired to the test station and a simple software control solution was created. This example has proved the project success because the control GUI was created by utilizing all the LICT modules at once.

The LICT software is designed as open. This design enables future updates and enhancements. These enhancements may be based upon a formal review, which was conducted by the NI LabVIEW developers inside Honeywell Brno.
# Table of Figures

Figure 1-1: GUI created in CVI ................................................................................. 3
Figure 1-2: GUI created in LabVIEW ................................................................. 4
Figure 1-3: Standard control program size ....................................................... 5
Figure 1-4: IVI Architecture – taken from [5] .................................................. 11
Figure 1-5: IAL DCPS architecture – taken from [3] ....................................... 13
Figure 1-6: Customer layer architecture – taken from [1] ............................... 14
Figure 2-1: Signal approach architecture .......................................................... 16
Figure 2-2: Source Signal State Diagram ......................................................... 17
Figure 2-3: Measurement Signal State Diagram ............................................ 18
Figure 2-4: Connection Signal State Diagram ............................................... 19
Figure 2-5: Communication Signal State Diagram ......................................... 20
Figure 3-1: DIO “Enable” Virtual Instrument .................................................. 21
Figure 3-2: NI LabVIEW project tree structure ............................................ 22
Figure 3-3: HPF example .................................................................................. 25
Figure 3-4: Configuration management chain ............................................... 26
Figure 3-5: Values matrix schematic overview .............................................. 27
Figure 3-6: DIO signal cluster ......................................................................... 31
Figure 3-7: DO Value transformation ................................................................ 33
Figure 3-8: DIO control chain ....................................................................... 33
Figure 3-9: DIO GetAttributeValue polymorphic VI .................................... 39
Figure 3-10: DIO SetAttributeValue polymorphic VI .................................... 40
Figure 3-11: DIO ReadCfg usage example ..................................................... 41
Figure 3-12: AIO signal cluster ....................................................................... 43
Figure 3-13: AIO control chain ..................................................................... 45
Figure 3-14: RMS value count ....................................................................... 49
Figure 3-15: AO Reset function chain ............................................................. 50
Figure 3-16: AIO ReadCfg usage example ..................................................... 53
Figure 3-17: DCPWR signal cluster ................................................................. 55
Figure 3-18: DCPWR control chain ............................................................... 57
Table of Tables

Table 1-1: Quality Function Deployment table ............................................................... 6
Table 1-2: Exclusion from survey results ........................................................................... 7
Table 1-3: Basic SCPI commands .................................................................................... 9
Table 1-4: National Instruments drivers .......................................................................... 10
Table 1-5: Project requirements ..................................................................................... 15
Table 3-1: Module capabilities ....................................................................................... 23
Table 3-2: AIO Signal Types .......................................................................................... 43
Table 3-3: AI Terminal configurations ............................................................................ 47
Table 3-4: DCPWR trigger sources ................................................................................ 56
Table 3-5: SERIAL signal types ..................................................................................... 67
Table 3-6: SERIAL parity types ...................................................................................... 68
Table 3-7: SERIAL supported stop bits .......................................................................... 68
Table 3-8: SERIAL supported termination characters .................................................... 68
Table 3-9: SERIAL flow control types ......................................................................... 69
Table 3-10: Attribute categories .................................................................................. 81
Table 4-1: Station instrument setup ............................................................................... 86
Table 4-2: Example module description ...................................................................... 91
Literature


101
### List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIO</td>
<td>Analog Input / Output</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ATE</td>
<td>Automated Test Equipment</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>CFG</td>
<td>Configuration</td>
</tr>
<tr>
<td>CL</td>
<td>Customer Layer</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial off the Shelf</td>
</tr>
<tr>
<td>DC</td>
<td>Direct Current</td>
</tr>
<tr>
<td>DIO</td>
<td>Digital Input / Output</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Link Library</td>
</tr>
<tr>
<td>DMM</td>
<td>Digital Multimeter</td>
</tr>
<tr>
<td>GPIB</td>
<td>General Purpose Interface Bus</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Abstraction Layer</td>
</tr>
<tr>
<td>HPF</td>
<td>Hardware Property File</td>
</tr>
<tr>
<td>IAL</td>
<td>Instrument Abstraction Layer</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>I/O</td>
<td>Input / Output</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LICIT</td>
<td>LabVIEW Instrument Control Toolbox</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>NI</td>
<td>National Instruments</td>
</tr>
<tr>
<td>PS</td>
<td>Power Supply (Source)</td>
</tr>
<tr>
<td>PXI</td>
<td>PCI eXtensions for Instruments</td>
</tr>
<tr>
<td>SCPI</td>
<td>Standard Commands for Programmable Instrumentation</td>
</tr>
<tr>
<td>UUT</td>
<td>Unit Under Test</td>
</tr>
<tr>
<td>QFD</td>
<td>Quality Function Deployment</td>
</tr>
<tr>
<td>VI</td>
<td>Virtual Instrument</td>
</tr>
<tr>
<td>VXI</td>
<td>VMI eXtension for Instrumentation</td>
</tr>
</tbody>
</table>
6. APPENDIX A – LIST OF ATTRIBUTES

6.1 DIO ATTRIBUTES:

6.1.1 Boolean Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE</td>
<td></td>
</tr>
<tr>
<td>RANGE_CHECK</td>
<td></td>
</tr>
<tr>
<td>QUERY_INSTRUMENT_STATUS</td>
<td>(read only)</td>
</tr>
<tr>
<td>RECORD_COERCIONS</td>
<td></td>
</tr>
<tr>
<td>SIMULATE</td>
<td>(read only)</td>
</tr>
<tr>
<td>INTERCHANGE_CHECK</td>
<td></td>
</tr>
<tr>
<td>SPY</td>
<td></td>
</tr>
<tr>
<td>USE_SPECIFIC_SIMULATION</td>
<td>(read only)</td>
</tr>
</tbody>
</table>

6.1.2 Integer Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS_DRIVER_CLASS_SPEC_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_CLASS_SPEC_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_CLASS_SPEC_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_CLASS_SPEC_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>PXI_MANF_ID</td>
<td></td>
</tr>
<tr>
<td>PXI_MODEL_CODE</td>
<td></td>
</tr>
<tr>
<td>IDLE_STATE</td>
<td>1 – High</td>
</tr>
<tr>
<td></td>
<td>2 – Low</td>
</tr>
<tr>
<td>SAMPLE_MODE</td>
<td>1 – Finite samples</td>
</tr>
<tr>
<td></td>
<td>2 – Continuous samples</td>
</tr>
</tbody>
</table>
### Parameter Name

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 – HW trigger single</td>
</tr>
</tbody>
</table>

### DATA_LAYOUT

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – GRPCHNL</td>
</tr>
<tr>
<td>1 – GRPSCNNUM</td>
</tr>
</tbody>
</table>

### LINE_GROUP

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – Channel per line</td>
</tr>
<tr>
<td>1 – Channel for all lines</td>
</tr>
</tbody>
</table>

### START_TRIGGER_TYPE

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 – Analog edge</td>
</tr>
<tr>
<td>2 – Digital edge</td>
</tr>
<tr>
<td>3 – Digital pattern</td>
</tr>
</tbody>
</table>

### ARM_START_TRIGGER_TYPE

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 – Analog window</td>
</tr>
<tr>
<td>5 – None</td>
</tr>
</tbody>
</table>

### START_TRIGGER_EDGE

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 – Rising</td>
</tr>
</tbody>
</table>

### ARM_START_DIGITALEdge_EDGE

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 – Falling</td>
</tr>
</tbody>
</table>

### SAMP_TIMING_TYPE

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 – Sample clock</td>
</tr>
<tr>
<td>2 – Burst handshake</td>
</tr>
<tr>
<td>3 – Handshake</td>
</tr>
<tr>
<td>4 – Implicit</td>
</tr>
<tr>
<td>5 – On demand</td>
</tr>
<tr>
<td>6 – Change detection</td>
</tr>
<tr>
<td>7 – Pipelined sample clock</td>
</tr>
</tbody>
</table>

### CHANNEL_TYPE

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 – Analog input</td>
</tr>
<tr>
<td>2 – Analog output</td>
</tr>
<tr>
<td>3 – Digital input</td>
</tr>
<tr>
<td>4 – Digital output</td>
</tr>
<tr>
<td>5 – Counter input</td>
</tr>
<tr>
<td>6 – Counter output</td>
</tr>
</tbody>
</table>

### 6.1.3 Real Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>

| START_TRIGGER_DIGEDGE_DIGFLTR_MINPULSE_WIDTH |

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>
### 6.1.4 String Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS_DRIVER_PREFIX</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_VENDOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_DESCRIPTION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_PREFIX</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_LOCATOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>IO_RESOURCE_DESCRIPTOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>LOGICAL_NAME</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_VENDOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_DESCRIPTION</td>
<td>(read only)</td>
</tr>
<tr>
<td>INSTRUMENT_FIRMWARE_REVISION</td>
<td>(read only)</td>
</tr>
<tr>
<td>INSTRUMENT_MANUFACTURER</td>
<td>(read only)</td>
</tr>
<tr>
<td>INSTRUMENT_MODEL</td>
<td>(read only)</td>
</tr>
<tr>
<td>SUPPORTED_INSTRUMENT_MODELS</td>
<td>(read only)</td>
</tr>
<tr>
<td>DRIVER_SETUP</td>
<td></td>
</tr>
<tr>
<td>ANALOG_INPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>ANALOG_OUTPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>DIGITAL_INPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>COUNTER_INPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>DIGITAL_OUTPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>COUNTER_OUTPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>START_TRIGGER_DIGITAL_EDGE_SOURCE</td>
<td></td>
</tr>
<tr>
<td>ARM_START_DIGEDGE_TRIGGER_SOURCE</td>
<td></td>
</tr>
<tr>
<td>VENDOR_DRIVER_REVISION</td>
<td>(read only)</td>
</tr>
</tbody>
</table>
### 6.2 AIO ATTRIBUTES:

#### 6.2.1 Boolean Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE</td>
<td></td>
</tr>
<tr>
<td>RANGE_CHECK</td>
<td></td>
</tr>
<tr>
<td>QUERY_INSTRUMENT_STATUS</td>
<td>(read only)</td>
</tr>
<tr>
<td>RECORD_COERCIONS</td>
<td></td>
</tr>
<tr>
<td>SIMULATE</td>
<td>(read only)</td>
</tr>
<tr>
<td>INTERCHANGE_CHECK</td>
<td></td>
</tr>
<tr>
<td>SPY</td>
<td></td>
</tr>
<tr>
<td>USE_SPECIFIC_SIMULATION</td>
<td>(read only)</td>
</tr>
</tbody>
</table>

#### 6.2.2 Integer Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS_DRIVER_CLASS_SPEC_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_CLASS_SPEC_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_CLASS_SPEC_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_CLASS_SPEC_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>PXI_MANF_ID</td>
<td></td>
</tr>
<tr>
<td>PXI_MODEL_CODE</td>
<td></td>
</tr>
<tr>
<td>VOLT_UNITS</td>
<td>1 – Volts</td>
</tr>
<tr>
<td>CURRENT_UNITS</td>
<td>1 – Amps</td>
</tr>
<tr>
<td></td>
<td>2 – TEDs</td>
</tr>
<tr>
<td>TERMINAL_CONFIG</td>
<td>2 – R Single Ended</td>
</tr>
<tr>
<td></td>
<td>3 – NR Single Ended</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SHUNTRES_LOC</td>
<td>1 – Internal</td>
</tr>
<tr>
<td></td>
<td>2 – External</td>
</tr>
<tr>
<td>IDLE_STATE</td>
<td>1 – High</td>
</tr>
<tr>
<td></td>
<td>2 – Low</td>
</tr>
<tr>
<td>SAMPLE_MODE</td>
<td>1 – Finite samples</td>
</tr>
<tr>
<td></td>
<td>2 – Continuous samples</td>
</tr>
<tr>
<td></td>
<td>3 – HW trigger single</td>
</tr>
<tr>
<td>FUNC_TYPE</td>
<td>1 – Sine</td>
</tr>
<tr>
<td></td>
<td>2 – Triangle</td>
</tr>
<tr>
<td></td>
<td>3 – Square</td>
</tr>
<tr>
<td></td>
<td>4 – Saw tooth</td>
</tr>
<tr>
<td>DATA_LAYOUT</td>
<td>0 – GRPCHNL</td>
</tr>
<tr>
<td></td>
<td>1 – GRPSCNNUM</td>
</tr>
<tr>
<td>LINE_GROUP</td>
<td>0 – Channel per line</td>
</tr>
<tr>
<td></td>
<td>1 – Channel for all lines</td>
</tr>
<tr>
<td>START_TRIGGER_TYPE</td>
<td>1 – Analog edge</td>
</tr>
<tr>
<td></td>
<td>2 – Digital edge</td>
</tr>
<tr>
<td></td>
<td>3 – Digital pattern</td>
</tr>
<tr>
<td>ARM_START_TRIGGER_TYPE</td>
<td>4 – Analog window</td>
</tr>
<tr>
<td></td>
<td>5 – None</td>
</tr>
<tr>
<td>START_TRIGGER_EDGE</td>
<td>1 – Rising</td>
</tr>
<tr>
<td>ARM_START_DIGITALEdge_EDGE</td>
<td>2 – Falling</td>
</tr>
<tr>
<td>SAMP_TIMING_TYPE</td>
<td>1 – Sample clock</td>
</tr>
<tr>
<td></td>
<td>2 – Burst handshake</td>
</tr>
<tr>
<td></td>
<td>3 – Handshake</td>
</tr>
<tr>
<td></td>
<td>4 – Implicit</td>
</tr>
<tr>
<td></td>
<td>5 – On demand</td>
</tr>
<tr>
<td></td>
<td>6 – Change detection</td>
</tr>
<tr>
<td></td>
<td>7 – Pipelined sample clock</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CHANNEL_TYPE</td>
<td>1 – Analog input</td>
</tr>
<tr>
<td>(read only)</td>
<td>2 – Analog output</td>
</tr>
<tr>
<td></td>
<td>3 – Digital input</td>
</tr>
<tr>
<td></td>
<td>4 – Digital output</td>
</tr>
<tr>
<td></td>
<td>5 – Counter input</td>
</tr>
<tr>
<td></td>
<td>6 – Counter output</td>
</tr>
<tr>
<td>AO_OUTPUT_TYPE</td>
<td>1 – Voltage</td>
</tr>
<tr>
<td>(read only)</td>
<td>2 – Current</td>
</tr>
<tr>
<td></td>
<td>3 – Waveform</td>
</tr>
<tr>
<td>AI_MEAS_TYPE</td>
<td>1 – Voltage</td>
</tr>
<tr>
<td>(read only)</td>
<td>2 – Current</td>
</tr>
<tr>
<td></td>
<td>3 – Frequency (CNTR)</td>
</tr>
<tr>
<td>DATA_XFER_REQ_COND</td>
<td>1 – Memory empty</td>
</tr>
<tr>
<td></td>
<td>2 – Half full or less</td>
</tr>
<tr>
<td></td>
<td>3 – Memory not full</td>
</tr>
</tbody>
</table>

### 6.2.3 Real Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>START_TRIGGER_DIGEDGE_DIGFLTR_MINPULSE_WIDTH</td>
<td></td>
</tr>
<tr>
<td>AI_MAX</td>
<td></td>
</tr>
<tr>
<td>AI_MIN</td>
<td></td>
</tr>
<tr>
<td>AO_MAX</td>
<td></td>
</tr>
<tr>
<td>AO_MIN</td>
<td></td>
</tr>
<tr>
<td>TIMING_SAMPLE_CLK_RATE</td>
<td></td>
</tr>
</tbody>
</table>

### 6.2.4 String Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS_DRIVER_PREFIX</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_VENDOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>CLASS_DRIVER_DESCRIPTION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_PREFIX</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_LOCATOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>IORESOURCE_DESCRIPTOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>LOGICAL_NAME</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_VENDOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_DESCRIPTION</td>
<td>(read only)</td>
</tr>
<tr>
<td>INSTRUMENT_FIRMWARE_REVISION</td>
<td>(read only)</td>
</tr>
<tr>
<td>INSTRUMENT_MANUFACTURER</td>
<td>(read only)</td>
</tr>
<tr>
<td>INSTRUMENT_MODEL</td>
<td>(read only)</td>
</tr>
<tr>
<td>SUPPORTED_INSTRUMENT_MODELS</td>
<td>(read only)</td>
</tr>
<tr>
<td>DRIVER_SETUP</td>
<td></td>
</tr>
<tr>
<td>ANALOG_INPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>ANALOG_OUTPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>DIGITAL_INPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>COUNTER_INPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>DIGITAL_OUTPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>COUNTER_OUTPUT_PHYCHNL</td>
<td>(read only)</td>
</tr>
<tr>
<td>START_TRIGGER_DIGITAL_EDGE_SOURCE</td>
<td></td>
</tr>
<tr>
<td>ARM_START_DIGEDGE_TRIGGER_SOURCE</td>
<td></td>
</tr>
<tr>
<td>VENDOR_DRIVER_REVISION</td>
<td>(read only)</td>
</tr>
<tr>
<td>PULSEWIDTH_TERM</td>
<td></td>
</tr>
</tbody>
</table>
### 6.3 DCPWR ATTRIBUTES:

#### 6.3.1 Boolean Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE</td>
<td></td>
</tr>
<tr>
<td>RANGE_CHECK</td>
<td></td>
</tr>
<tr>
<td>QUERY_INSTRUMENT_STATUS</td>
<td>(read only)</td>
</tr>
<tr>
<td>RECORD_COERCIONS</td>
<td></td>
</tr>
<tr>
<td>SIMULATE</td>
<td>(read only)</td>
</tr>
<tr>
<td>INTERCHANGE_CHECK</td>
<td></td>
</tr>
<tr>
<td>SPY</td>
<td></td>
</tr>
<tr>
<td>USE_SPECIFIC_SIMULATION</td>
<td>(read only)</td>
</tr>
<tr>
<td>OVP_ENABLED</td>
<td>(multi-channel)</td>
</tr>
<tr>
<td>OUTPUT_ENABLED</td>
<td>(multi-channel)</td>
</tr>
<tr>
<td>INHIBIT</td>
<td>(multi-channel)</td>
</tr>
</tbody>
</table>

#### 6.3.2 Integer Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS_DRIVER_CLASS_SPEC_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_CLASS_SPEC_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_CLASS_SPEC_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_CLASS_SPEC_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_MAJOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_MINOR_VERSION</td>
<td>(read only)</td>
</tr>
<tr>
<td>CHANNEL_COUNT</td>
<td>(read only)</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>CURRENT_LIMIT_BEHAVIOR (multi-channel)</td>
<td>0 – Regulate</td>
</tr>
<tr>
<td></td>
<td>1 – Trip</td>
</tr>
<tr>
<td>TRIGGER_SOURCE (multi-channel)</td>
<td>0 – Immediate</td>
</tr>
<tr>
<td></td>
<td>1 - External</td>
</tr>
<tr>
<td></td>
<td>2 - Software</td>
</tr>
<tr>
<td></td>
<td>3..10 - TTL0..7</td>
</tr>
<tr>
<td></td>
<td>11..12 - ECL0..1</td>
</tr>
<tr>
<td></td>
<td>13 - PXI_STAR</td>
</tr>
<tr>
<td></td>
<td>14..20 - RTSI0..6</td>
</tr>
</tbody>
</table>

### 6.3.3 Real Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOLTAGE_LEVEL (multi-channel) (volts)</td>
<td></td>
</tr>
<tr>
<td>OVP_LIMIT (multi-channel) (volts)</td>
<td></td>
</tr>
<tr>
<td>CURRENT_LIMIT (multi-channel) (amps)</td>
<td></td>
</tr>
<tr>
<td>TRIGGERED_CURRENT_LIMIT (multi-channel) (amps)</td>
<td></td>
</tr>
<tr>
<td>TRIGGERED_VOLTAGE_LEVEL (multi-channel) (volts)</td>
<td></td>
</tr>
</tbody>
</table>

### 6.3.4 String Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP_CAPABILITIES</td>
<td>(read only)</td>
</tr>
<tr>
<td>FUNCTION_CAPABILITIES</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_PREFIX</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_VENDOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_DESCRIPTION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_PREFIX</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_LOCATOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>IO_RESOURCE_DESCRIPTOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>LOGICAL_NAME</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_VENDOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_DESCRIPTION</td>
<td>(read only)</td>
</tr>
<tr>
<td>INSTRUMENT_FIRMWARE_REVISION</td>
<td>(read only)</td>
</tr>
<tr>
<td>INSTRUMENT_MANUFACTURER</td>
<td>(read only)</td>
</tr>
<tr>
<td>INSTRUMENT_MODEL</td>
<td>(read only)</td>
</tr>
<tr>
<td>SUPPORTED_INSTRUMENT_MODELS</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_REVISION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_REVISION</td>
<td>(read only)</td>
</tr>
<tr>
<td>INSTRUMENT_SERIAL_NUMBER</td>
<td>(read only)</td>
</tr>
<tr>
<td>DRIVER_SETUP</td>
<td></td>
</tr>
</tbody>
</table>
6.4 SERIAL ATTRIBUTES:

6.4.1 Boolean Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE</td>
<td></td>
</tr>
<tr>
<td>RANGE_CHECK</td>
<td></td>
</tr>
<tr>
<td>QUERY_INSTRUMENT_STATUS</td>
<td>(read only)</td>
</tr>
<tr>
<td>RECORD_COERCIONS</td>
<td></td>
</tr>
<tr>
<td>SIMULATE</td>
<td>(read only)</td>
</tr>
<tr>
<td>INTERCHANGE_CHECK</td>
<td></td>
</tr>
<tr>
<td>SPY</td>
<td></td>
</tr>
<tr>
<td>USE_SPECIFIC_SIMULATION</td>
<td>(read only)</td>
</tr>
</tbody>
</table>

6.4.2 Integer Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAUDRATE</td>
<td></td>
</tr>
<tr>
<td>DATABITS</td>
<td>From 5 to 8</td>
</tr>
<tr>
<td>INPUTQSIZE (Read Only)</td>
<td></td>
</tr>
<tr>
<td>OUTPUTQSIZE</td>
<td></td>
</tr>
<tr>
<td>PARITY</td>
<td>10 – None</td>
</tr>
<tr>
<td></td>
<td>11 – Odd</td>
</tr>
<tr>
<td></td>
<td>12 – Even</td>
</tr>
<tr>
<td>STOPBITS</td>
<td>21 – One (1)</td>
</tr>
<tr>
<td></td>
<td>22 – One, half (1.5)</td>
</tr>
<tr>
<td></td>
<td>23 – Two (2)</td>
</tr>
<tr>
<td>FLOWCONTROL</td>
<td>30 – None</td>
</tr>
<tr>
<td></td>
<td>31 – Xon-Xoff</td>
</tr>
<tr>
<td></td>
<td>32 – RTS-CTS</td>
</tr>
<tr>
<td></td>
<td>33 – DTR-DSR</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>TERMINATE_CHAR_TX</td>
<td>0 – None 10 - LF 13 – CR</td>
</tr>
</tbody>
</table>

### 6.4.3 Real Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMEOUT</td>
<td>milliseconds</td>
</tr>
</tbody>
</table>

### 6.4.4 String Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP_CAPABILITIES</td>
<td>(read only)</td>
</tr>
<tr>
<td>FUNCTION_CAPABILITIES</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_PREFIX</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_VENDOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>CLASS_DRIVER_DESCRIPTION</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_PREFIX</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER LOCATOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>IO_RESOURCE_DESCRIPTOR</td>
<td>(read only)</td>
</tr>
<tr>
<td>LOGICAL_NAME</td>
<td>(read only)</td>
</tr>
<tr>
<td>SPECIFIC_DRIVER_VENDOR</td>
<td>(read only)</td>
</tr>
</tbody>
</table>
### 6.4.5 VISA Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>END_MODES_FOR_WRITES</td>
<td>0 – None</td>
</tr>
<tr>
<td></td>
<td>1 – Last Bit</td>
</tr>
<tr>
<td></td>
<td>2 – TermChar</td>
</tr>
<tr>
<td></td>
<td>3 – Break</td>
</tr>
<tr>
<td>END_MODES_FOR_READS</td>
<td>0 – None</td>
</tr>
<tr>
<td></td>
<td>1 – Last Bit</td>
</tr>
<tr>
<td></td>
<td>2 – TermChar</td>
</tr>
<tr>
<td>ALLOW_TRANSMIT</td>
<td>0 – False</td>
</tr>
<tr>
<td></td>
<td>1 – True</td>
</tr>
<tr>
<td>NUMBER_OF_BYTES_AT_PORT</td>
<td>(read only)</td>
</tr>
</tbody>
</table>
7. APPENDIX B – REQUIREMENT TRACEABILITY MATRIX

According to the requirement traceability matrix, all requirements are covered by the thesis. Requirements with “Analysis” verification method are covered by the thesis text itself. Requirements with “Test Case” verification method are covered by the software. E.g. the integration VIs or the example made with the EGPWS.

<table>
<thead>
<tr>
<th>REQ ID</th>
<th>Requirement Text / Input</th>
<th>Verification Method</th>
<th>Paragraph</th>
<th>Paragraph Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS_0010</td>
<td>LIC shall utilize NI LabVIEW virtual instruments</td>
<td>Analysis</td>
<td>1.1</td>
<td>DEVELOPMENT ENVIRONMENT CHOISING PROCESS</td>
</tr>
<tr>
<td>SYS_0010</td>
<td>LIC shall utilize NI LabVIEW virtual instruments</td>
<td>Analysis</td>
<td>1.2</td>
<td>COMPANY PROJECT ANALYSIS</td>
</tr>
<tr>
<td>SYS_0010</td>
<td>LIC shall utilize NI LabVIEW virtual instruments</td>
<td>Analysis</td>
<td>3.1</td>
<td>PROJECT DETAILS</td>
</tr>
<tr>
<td>SYS_0020</td>
<td>LIC shall utilize Instrument Abstraction Layer</td>
<td>Analysis</td>
<td>1.3</td>
<td>INSTRUMENT DRIVER LAYER</td>
</tr>
<tr>
<td>SYS_0030</td>
<td>LIC may support vendor specific drivers</td>
<td>Analysis</td>
<td>1.3</td>
<td>INSTRUMENT DRIVER LAYER</td>
</tr>
<tr>
<td>SYS_0030</td>
<td>LIC may support vendor specific drivers</td>
<td>Analysis</td>
<td>2</td>
<td>SIGNAL APPROACH</td>
</tr>
<tr>
<td>SYS_0040</td>
<td>LIC shall be divided into modules. Each module shall represent specific instrument or functionality.</td>
<td>Analysis</td>
<td>1.2</td>
<td>COMPANY PROJECT ANALYSIS</td>
</tr>
<tr>
<td>SYS_0040</td>
<td>LIC shall be divided into modules. Each module shall represent specific instrument or functionality.</td>
<td>Analysis</td>
<td>3.1</td>
<td>PROJECT DETAILS</td>
</tr>
<tr>
<td>SYS_0040</td>
<td>LIC shall be divided into modules. Each module shall represent specific instrument or functionality.</td>
<td>Analysis</td>
<td>4</td>
<td>PROOF OF CONCEPT</td>
</tr>
<tr>
<td>SYS_0050</td>
<td>LIC shall provide unified interface for instrument control. Description: Each virtual instrument created for instrument control purpose shall follow the same design approach (Signal cluster)</td>
<td>Analysis</td>
<td>3.3</td>
<td>DISCRETE INPUT AND OUTPUT</td>
</tr>
<tr>
<td>SYS_0050</td>
<td>LIC shall provide unified interface for instrument control. Description: Each virtual instrument created for instrument control purpose shall follow the same design approach (Signal cluster)</td>
<td>Analysis</td>
<td>3.4</td>
<td>ANALOG INPUT AND OUTPUT</td>
</tr>
<tr>
<td>SYS_0050</td>
<td>LIC shall provide unified interface for instrument control. Description: Each virtual instrument created for instrument control purpose shall follow the same design approach (Signal cluster)</td>
<td>Analysis</td>
<td>3.5</td>
<td>DC POWER SOURCE</td>
</tr>
<tr>
<td>SYS_0050</td>
<td>LIC shall provide unified interface for instrument control. Description: Each virtual instrument created for instrument control purpose shall follow the same design approach (Signal cluster)</td>
<td>Analysis</td>
<td>3.6</td>
<td>SERIAL INTERFACE</td>
</tr>
<tr>
<td>SYS_0060</td>
<td>LIC shall utilize signal approach to</td>
<td>Analysis</td>
<td>2</td>
<td>SIGNAL</td>
</tr>
<tr>
<td>REQ ID</td>
<td>Requirement Text / Input</td>
<td>Verification Method</td>
<td>Paragraph</td>
<td>Paragraph Name</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------------------------</td>
<td>---------------------</td>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>SYS_0060</td>
<td>LIC shall utilize signal approach to instrument control</td>
<td>Analysis</td>
<td>3.3</td>
<td>DISCRETE INPUT AND OUTPUT</td>
</tr>
<tr>
<td>SYS_0060</td>
<td>LIC shall utilize signal approach to instrument control</td>
<td>Analysis</td>
<td>3.4</td>
<td>ANALOG INPUT AND OUTPUT</td>
</tr>
<tr>
<td>SYS_0060</td>
<td>LIC shall utilize signal approach to instrument control</td>
<td>Analysis</td>
<td>3.5</td>
<td>DC POWER SOURCE</td>
</tr>
<tr>
<td>SYS_0060</td>
<td>LIC shall utilize signal approach to instrument control</td>
<td>Analysis</td>
<td>3.6</td>
<td>SERIAL INTERFACE</td>
</tr>
<tr>
<td>SYS_0070</td>
<td>LIC shall utilize polymorph virtual instruments</td>
<td>Analysis</td>
<td>3.8</td>
<td>ATTRIBUTE MANIPULATION</td>
</tr>
<tr>
<td>SYS_0080</td>
<td>LIC source code shall be designed to support future custom modifications</td>
<td>Analysis</td>
<td>3.1</td>
<td>PROJECT DETAILS</td>
</tr>
<tr>
<td>SYS_0080</td>
<td>LIC source code shall be designed to support future custom modifications</td>
<td>Analysis</td>
<td>4</td>
<td>PROOF OF CONCEPT</td>
</tr>
<tr>
<td>SYS_0090</td>
<td>LIC shall support Discrete Input and Output instruments</td>
<td>Test Case</td>
<td>3.3</td>
<td>DISCRETE INPUT AND OUTPUT</td>
</tr>
<tr>
<td>SYS_0100</td>
<td>LIC shall support Analog Input and Output instruments</td>
<td>Test Case</td>
<td>3.4</td>
<td>ANALOG INPUT AND OUTPUT</td>
</tr>
<tr>
<td>SYS_0110</td>
<td>LIC shall support DC power source instruments</td>
<td>Test Case</td>
<td>3.5</td>
<td>DC POWER SOURCE</td>
</tr>
<tr>
<td>SYS_0120</td>
<td>LIC shall support Serial communication</td>
<td>Test Case</td>
<td>3.6</td>
<td>SERIAL INTERFACE</td>
</tr>
<tr>
<td>SYS_0130</td>
<td>LIC architecture shall allow to add next instruments and capabilities</td>
<td>Analysis</td>
<td>3.1</td>
<td>PROJECT DETAILS</td>
</tr>
<tr>
<td>SYS_0140</td>
<td>LIC shall cover configuration file management capability</td>
<td>Test Case</td>
<td>3.2</td>
<td>CONFIGURATION FILE MANAGEMENT</td>
</tr>
<tr>
<td>SYS_0150</td>
<td>LIC architecture shall allow to share control solutions and capabilities between projects</td>
<td>Test Case</td>
<td>3.1</td>
<td>PROJECT DETAILS</td>
</tr>
<tr>
<td>SYS_0150</td>
<td>LIC architecture shall allow to share control solutions and capabilities between projects</td>
<td>Test Case</td>
<td>4</td>
<td>PROOF OF CONCEPT</td>
</tr>
<tr>
<td>SYS_0160</td>
<td>LIC file structure shall be designed as a base of common library</td>
<td>Analysis</td>
<td>3.1</td>
<td>PROJECT DETAILS</td>
</tr>
</tbody>
</table>
8. APPENDIX C – NUMERIC DATA TYPES

This appendix was created to clarify the numeric data types used in this diploma thesis. Each numeric data type used is derived from the NI LabVIEW numeric data types [10], which are noted in this table.

<table>
<thead>
<tr>
<th>Used name</th>
<th>Numeric Data Type</th>
<th>Bits of Storage on Disk</th>
<th>Approximate Number of Decimal Digits</th>
<th>Approximate Range</th>
</tr>
</thead>
</table>
| Double64  | Double-precision, floating-point        | 64                      | 15                                  | Minimum positive number: 4.94e–324  
Maximum positive number: 1.79e+308  
Minimum negative number: −4.94e–324  
Maximum negative number: −1.79e+308 |
| Int8      | Byte signed integer                    | 8                       | 2                                   | −128 to 127                                                                      |
| Int16     | Word signed integer                    | 16                      | 4                                   | −32,768 to 32,767                                                                |
| Int32     | Long signed integer                    | 32                      | 9                                   | −2,147,483,648 to 2,147,483,647                                                   |
| UInt8     | Byte unsigned integer                  | 8                       | 2                                   | 0 to 255                                                                         |
| UInt16    | Word unsigned integer                  | 16                      | 4                                   | 0 to 65,535                                                                      |
| UInt32    | Long unsigned integer                  | 32                      | 9                                   | 0 to 4,294,967,295                                                               |

This table was taken from the NI LabVIEW online manual and is cited as [10] NI LabVIEW Numeric Data Types.