CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Mechanical Engineering

Department of Instrumentation and Control Engineering

MASTER THESIS

# Design and Implementation of Automatic Tests for Siemens PROFINET IO Development Kit

2017 Bc. Alexandr Osadcii

# Annotation List

| | |
|---|---|
| Authors Name: | Bc. Alexandr Osadcii |
| Name of Master's Thesis: | Design and Implementation of Automatic Tests for Siemens PROFINET IO Development Kit |
| Year: | 2017 |
| Field of study: | Instrumentation and Control Engineering |
| Department: | Department of Instrumentation and Control Engineering |
| Supervisor: | doc. Ing. Ivo Bukovský, Ph.D. |

| | | |
|---|---|---|
| Bibliographical data: | Number of pages | 79 |
| | Number of figures | 12 |
| | Number of tables | 11 |
| | Number of attachments | 2 |

Keywords: PROFINET, Seimens PROFINET IO Development Kit, Python Test Automat, automation, testing, Python, TShark, Lua

## Statement

## Acknowledgments

Firstly I would like to express great thanks to my parents and my girlfriend for their support. Also I would like to express great thanks to my coordinator and supervisor doc. Ing. Ivo Bukovský, Ph.D. for the advices and the remarks, to Siemens s.r.o. Company, as the thesis ordering party, and to its employees personally: Miroslav Dušek, Josef Vopička, Jan Roth, Jan Slováček and Jakub Kováč.

## Abstract

This master thesis refers to design and implementation of automated testing for Siemens PROFINET IO Development Kit. The design of the automated testing is based on the PROFINET communication standard and the Siemens development kit functionality. The implementation is done by means of the Python programming language, the Python Test Automat test tool (which manages the test procedure), a TIA Portal project for S7-1500 CPU 1511-1 and TShark analyzer with integrated Lua scripts. The functionality of the automated test implementation is being verified on the real application by a complete test run.

# Contents

# Abbreviations

| | |
|---|---|
| AR | Application Relationship |
| ARP | Address Resolution Protocol |
| ASIC | Application-Specific Integrated Circuit |
| CM | Context Manager |
| CR | Communication Relationship |
| DevKit | Development Kit |
| DBAI | Direct Buffer Access Interface |
| DCP | Discovery and basic Configuration Protocol |
| DHCP | Dynamic Host Configuration Protocol |
| DHT | Data Hold Time |
| EMC | Electromagnetic Compatibility |
| GPIO | General-Purpose Input/Output |
| GSD | General Station Description |
| ID | Identifier |
| IDE | Integrated Development Environment |
| IO | Input/Output |
| IOC | IO Controller |
| IOD | IO Device |
| IOS | IO Supervisor |
| IP | Internet Protocol |
| IRT | Isochronous Real Time Protocol |
| ISO | International Organization for Standardization |
| I&M | Identification and Maintenance Profile |
| MAC | Media Access Control |
| MRP | Media Redundancy Protocol |
| OPC | Open Platform Communications |

| | |
|---|---|
| OSI | Open Systems Interconnection |
| PLC | Programmable Logic Controller |
| PN | PROFINET |
| PTCP | Precision Transparent Clock Protocol |
| PyTeMat | Python Test Automat |
| RPC | Remote Procedure Call |
| RTA | Real Time Protocol Acyclic |
| RTC | Real Time Protocol Cyclic |
| TBL | Test Box Language |
| TCP | Transmission Control Protocol |
| UART | Universal Asynchronous Receiver/Transmitter |
| UDP | User Datagram Protocol |
| USB | Universal Serial Bus |
| UUID | Universally Unique Identifier |
| XML | Extensible Markup Language |

# 1 Introduction

Nowadays our community can’t imagine modern life without all the technologies that are involved and actively used in almost all spheres of the society. This improvement of our life and comfort in comparison to our ancestors plays crucial role in the development of our society and expanding of our knowledge of the world around us.  All this could be possible only after massive industrialization processes over the world and the following automation of the industries.

This automation is realized by so called PLCs (Programmable Logic Controllers) that, after their invention, were installed in the main factories and now are widely used for control of sequential and combinatorial logic in industrial processes [1].

The production of one type of the technology device (and usually one factory may produce more types of the devices) expects controlling of large amount of production units. Moreover these units are located around large areas of the factories. That’s why one PLC with output and input channels is usually not enough to handle the whole production and several control nodes are installed in the factory around the area. As the production processes are highly complicated and detailed the synchronization between the control nodes is a necessity. This requirement is fulfilled by the means of the distributed control system and fieldbus protocols [2].

Up to date, great amount of the fieldbus protocols exist. The differences between them depend on the application, on the type of the devices, companies designed the protocol and even on the continents, where the protocols are spread, among them are PROFIBUS, Modbus, AS-Interface, CAN, HART, IOLINK, PROFINET, EtherCAT, EtherNet/IP and so on [2].

Working with new Siemens devices from Digital Factory division you can find more factory devices functioning on PROFINET protocol and more of them appears on the market since this communication protocol was invented. As the Siemens Company tries to extend the functionality of the PROFINET devices according to the real requirements from the consumers and to equip by them plants all around the world, more devices are currently being developed based on the PROFINET fieldbus.

The development of such equipment is a highly complicated process and a lot of problems may take place during it. In order to reduce the problems and provide completely functional device to the customer, each new device development requires complex and intensive testing before it can be released to the market. The defects in some devices may lead to large problems of the plants, to decreasing of productivity or even to threatening the life and health of the employees. That's why verification and validation (or simply testing) of the functionality according to the standards is an obligatory procedure, which expects high attention and responsibility from the testing person. As the result, the producer of the developed devices guarantees the customer, that the plant devices are fully functional and complies with all the requirements.

One of the aims of recent engineers involved in testing is including of as much automation in the procedure as it is possible. Not only the automation saves human and time resources, but also provides more reliable results.

That's why design and implementation of automatic tests is the topic concerned within the framework of the thesis. This automation is done for testing Siemens development kit based on ERTEC 200P for PROFINET IO.

The objective of this thesis was to design an automated method and to implement automated testing for the development kit functionality based on PROFINET standard communication. At the beginning, I investigated the basis of PROFINET IO standard. Then I familiarized with the development kit and its main tasks and features. In the next step I studied open source program PyTeMat (Python Test Automat), created in Prague Siemens inside Corporate Technology division and based on object-oriented programming with Python language. Based on this knowledge, I designed test cases that compose part of regression test process required. During the design of some test cases, I expanded the core of PyTeMat with two modules. One of them manages communication via serial port with the development kit. Another handles capturing Ethernet network communication by means of network protocol analyzer - "TShark". For solid testing, I needed some additional hardware equipment, including PLC supporting current version of PROFINET IO standard. As the development kit under the test is considered as a unit of a PROFINET station, I created a project for the PLC, which covers the needs of the test cases. At the end I implemented the whole test process designed by Python programming language in PyTeMat. The results from the performed tests were analyzed and some retests were done as needed.

# 2 PROFINET Standard

According to source [3], “PROFINET (pitch acronym for Process Field Net) is an industry technical standard for data communication over Industrial Ethernet, designed for collecting data from, and controlling, equipment in the industrial systems, with a particular strength in delivering data under tight time constraints”.

This standard is created and currently being maintained by Profibus & Profinet International Organization. Primarily PROFINET was created as a real time Ethernet application for automation. As the working group declares in source [4], this application “can be implemented for production and process automation, safety applications, and the entire range of drive technology up to and including isochronous motion control applications.”

As the main advantages of this communication standard they offer:

- Highly scalable architectures.
- Access to field devices over the network.
- Maintenance and servicing from anywhere (even over the internet).
- Lower costs for production/quality data monitoring. [5]

This standard is still being developed and new functions are being added to it. And due to all this we can say, that PROFINET can be optimally used in all spheres of automation engineering.

## 2.1 PROFINET Architecture

The whole PROFINET concept is based on Industrial Ethernet, as this standard provides robust cabling and EMC immunity, and Ethernet standard according to IEEE 802 in IEC 61158 at the fieldbus level, which enables transfer of process data in real-time communication [2].

Referencing to the ISO/OSI layer model, the PROFINET protocol can be determined by physical (Layer 1), data link (Layer 2), network (Layer 3), transport (Layer 4) and application (Layer 7) layers (see Figure 1).

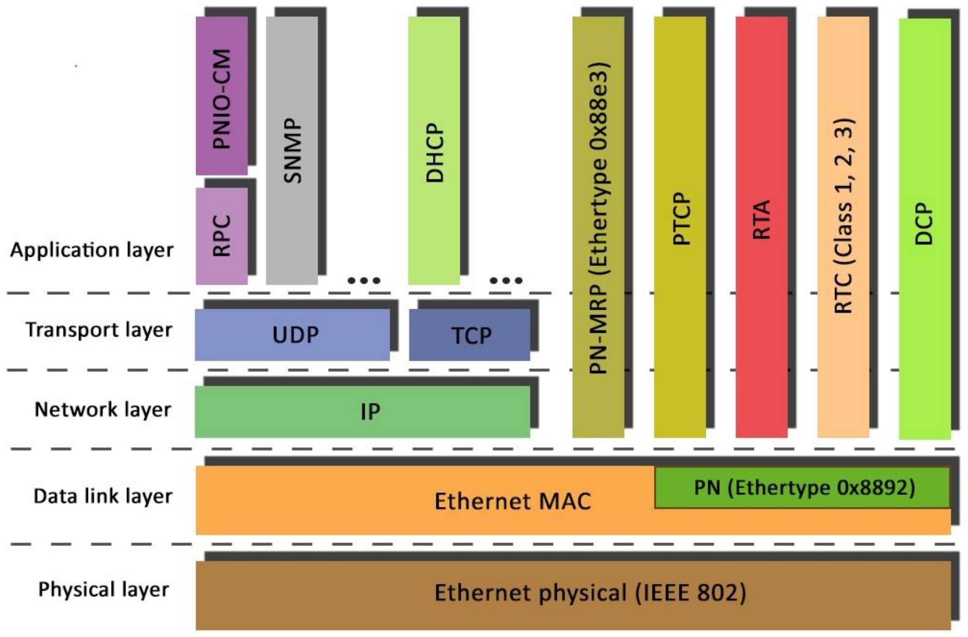


*Figure 1: PROFINET ISO/OSI model*

Physical layer and data link layers for PROFINET are specified by Ethernet standard with some concrete definitions [6].

Every PROFINET device has integrated switch technology inside. That enables switch-based network infrastructure with all types of topologies (star, ring, tree and line). Nevertheless, it establishes some mandatory requirements for the ports on the devices. Among them are those as follows:

1. Twisted pair STP and UTP 5e+ are allowed as copper cables.
2. RJ45 and M12 connectors on the copper cables.
3. The switch ports shall support 100BASE-TX form of Fast Ethernet with full duplicity and automatic crossover functionality.
4. The ports shall support auto negotiation for 10 Mbps speed.
5. Delay time of the port shall be under 10 µs.

As PROFINET supports various applications, which will be described later, the standard includes various protocols. These protocols use different methods of addressing: some of them are MAC-based addressing (Layer 2 communication) and others are IP-based addressing (Layer 3 communication).

These applications are distributed as follows:

1. MAC or Layer 2 addressing:
   a. PROFINET IO communication (in other words, process data), for example RT and IRT (RTC class 1 and 3)
   b. Some PROFINET services, for example DCP (for setting and requesting devices' names and IP addresses)
2. IP or Layer 3 addressing:
   a. Other PROFINET services, for instance SNMP protocol for providing information about the device, RPC protocol allowing read/write services for devices' parameterization  or reading and writing other data objects (PNIO-CM), or RTA protocol used for alarm handling

The definition of the application layer with its services for PROFINET is more complicated and requires plenty of details. Therefore main and crucial part of the application services are described in the following chapters.

## 2.2 PROFINET Application Services

### 2.2.1 Devices Types

Information from this subchapter is adapted from source [6]. The typical plant automation system is usually based on one or several PLCs that are connected to the process units by means of the IO system. This IO system is defined as a version of a hierarchical decentralized system and may consists of smaller subsystems. At the same time this system and its subsystems are composed from 3 types of devices: IO controllers (IOC), IO supervisors (IOS) and IO devices (IOD). Therefore the whole concept is called PROFINET IO. Each type of the device has its own features, functionalities and associations with other devices. Some of these features may be common for several types of devices. In PROFINET IO there are defined cyclic, acyclic and general functionalities, which may be obtained by the devices.

a) IO controller (IOC) defines a device that controls the main automation processes performing all three types of functionalities. This device can be associated with IO devices either one or several. The number of associated devices is usually limited by the PROFINET application. The main functionalities of IOC are:

  - Cyclic – exchange of process data (or IO data) with connected IODs;
  - Acyclic – parameterization and configuration of IODs, read and write access to IOD's record data, sending alarms to and treatment of alarms from IODs;
  - General – manages different PROFINET applications such as system redundancy or isochronous operation, which require for example clock synchronization.

b) IO supervisor (IOS) defines a device that manages reading and providing of configuration, parameterization and diagnosis data from IOCs and IODs that are associated with the IOS, and for them.

c) IO device (IOD) defines a field device that is able to perform activities based on functionalities of all three types:

  - Cyclic – exchange of process data (or IO data) with connected IOC, either one or several depending of the class of the IOD;
  - Acyclic – processing parameterization and configuration requests from IOC, providing record and diagnosis data to IOC and IOS, sending alarms to IOC;
  - General – manages different PROFINET applications such as system redundancy or isochronous operation, which require for example clock synchronization.

As an easy definition, IOC is typically a PLC which runs the automation program inside itself and can be compared to PROFIBUS master of class1. IOC provides output data to the IODs and consumes input data from IODs. IOS can be compared to PROFIBUS master of class 2 and be represented as programming, personal computer, human machine interface. And IOD is represented by a field device connected to IOCs and can be compared to PROFIBUS slave. IOD provides input process data and consumes output data from the IOCs. [4]

### 2.2.2 Device Model of an IOD

The IOD itself is composed of different units that represent real hardware or virtual functional components. Together they create hierarchical internal model of the device that can be described as follows:

- IOD contains

- one or more modules, which contain
- one or more submodules, which may contain
- quantity of channels [6]

Each level of this model is represented with an address which allows designation of service endpoints, according to the following address attributes:

- one IOD instance with application process identifier, which contain
- one or more slots, which contain
- one or more subslots, which may contain
- quantity of channel numbers [6]

Every instance is identified by an instance number in form of UUID and obtains one or more application process identifiers (API). Each Module is addressed by a slot number in range of 0 to 0x7FFF. Each submodule is addresses by a subslot number in range of 0 to 0x9FFF. Starting from this level of the structure record data (parameterization, identification and maintenance, diagnosis data), IO data (summered from channels) and alarms are handled. This means, that objects related to acyclic communication shall be addressed at least by UUID, API, slot number and subslot number on the application layer. In different cases some objects may be specified with additional channel number, when the object relates to a concrete channel. So each channel is addressed by a channel number in range of 0 to 0x7FFF and may provide some record data, IO data and diagnosis. [6], [7]

Cyclic communication that handles IO data exchange relates to a separate protocol and addressing in this case is realized in a different way.

In order to make this entire concept clear, I provide an example of a typical IOD describing its internal structure with addressing numbers. For this purpose I’ve chosen an IOD, which is used in my testing station as a partner device to the development kit under the test. This device is represented by ET200SP station combined of IM155-6 PN HF interface module, one output module, one input module and one terminate or server module (Figure 2).

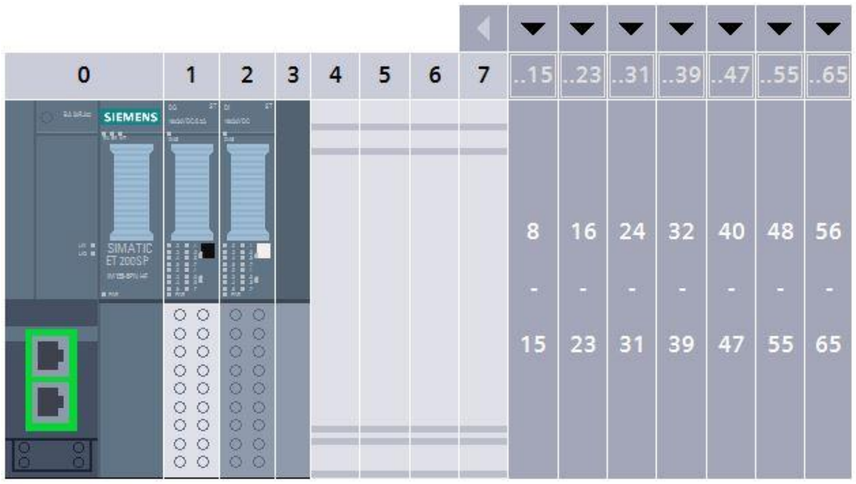


*Figure 2: ET200SP station picture from TIA portal engineering tool*

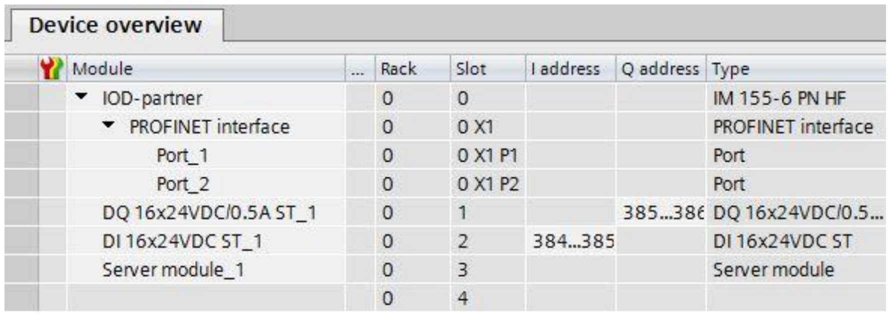
In Figure 2, we can see a picture of the ET200SP station configured in TIA portal engineering tool. The numbers above the blue device represent slot numbers starting from 0 and up to 65. This means, that this interface module supports 64 additional modules plugged into this single station. The interface module itself is placed in slot number 0. As we can see, it has 2 Ethernet ports. Then a module with output channels follows placed on slot 1. One module with input channels is placed in slot 1. And the station is terminated by a server module without any channels in slot 3. As the result, Figure 2 shows the module structure of the IOD with addresses of the slots.

According to the definition above, each module shall contain at least one submodule and slot shall contain at least one subslot. That can be partially seen in Figure 3, where a device overview of the station is provided by the TIA portal tool.

The first module called “IOD-partner”, which is the project name of the device, is then divided into 4 lines with the same slot number 0. These lines represent submodules.  In the first line, there is placed the device access point (DAP) submodule with subslot number address that is usually equal to 0x1. In the second line, there is a PROFINET interface submodule X1 with subslot number address that is usually equal to 0x8000. After that, two port submodules (P1 and P2) are placed in the following subslots. As they follow after the interface submodule, their subslot numbers are usually equal to 0x8001 and 0x8002 respectively.

Output, input and server modules have only one submodule each, and therefore each slot has one subslot and the subslot has usually number 0x1.

The real slot and subslot numbers are defined by device's GSD file, which is a XML file with complete description of the device, designed to make complete parameterization and configuration possible by any engineering tool [7].



Device overview

| Module | ... | Rack | Slot | I address | Q address | Type |
|---|---|---|---|---|---|---|
| ▼ IOD-partner | | 0 | 0 | | | IM 155-6 PN HF |
| ▼ PROFINET interface | | 0 | 0 X1 | | | PROFINET interface |
| Port_1 | | 0 | 0 X1 P1 | | | Port |
| Port_2 | | 0 | 0 X1 P2 | | | Port |
| DQ 16x24VDC/0.5A ST_1 | | 0 | 1 | | 385...386 | DQ 16x24VDC/0.5... |
| DI 16x24VDC ST_1 | | 0 | 2 | 384...385 | | DI 16x24VDC ST |
| Server module_1 | | 0 | 3 | | | Server module |
| | | 0 | 4 | | | |

*Figure 3: ET200SP device overview from TIA portal engineering tool*

In our case, for example, if we need to read some parameters from the interface submodule, the addressing of this application service would include UUID and API of the application relationship (this term will be explained in the following chapters), slot 0x0, subslot 0x8000. The address within the parameters read request for the server module would consist of the following elements: UUID and API of the application relationship, slot 0x3, subslot 0x1.

As the result instead of one endpoint for addressing while performing any action supported by the application functionality, we have a plenty of end points. At the first sight it may seems as a superfluous complication, but in the real application this structure makes handling and maintenance of the PROFINET station much easier.

### 2.2.3 Application and Communication Relationships

In order to establish PROFINET based communication between IOC, IOS and IOD an application relationship (AR) should be established. PROFINET standard distinguishes four types of AR: IO AR, Supervisor AR, Device access AR, Implicit AR. [6]

IO AR provides all the capabilities for regular communication between IOC and IOD. These capabilities are similar to channels for cyclic IO data exchange, acyclic data exchange for reading and writing record data explicitly and transmission of alarm data. Each of these channels can be considered as a separate communication relationship and they are set up simultaneously. The entire concept of AR and CRs is schematically figured in Figure 4.

Other types of AR are not essential for the tests described in this thesis, therefore their description is omitted and can be found in [6].

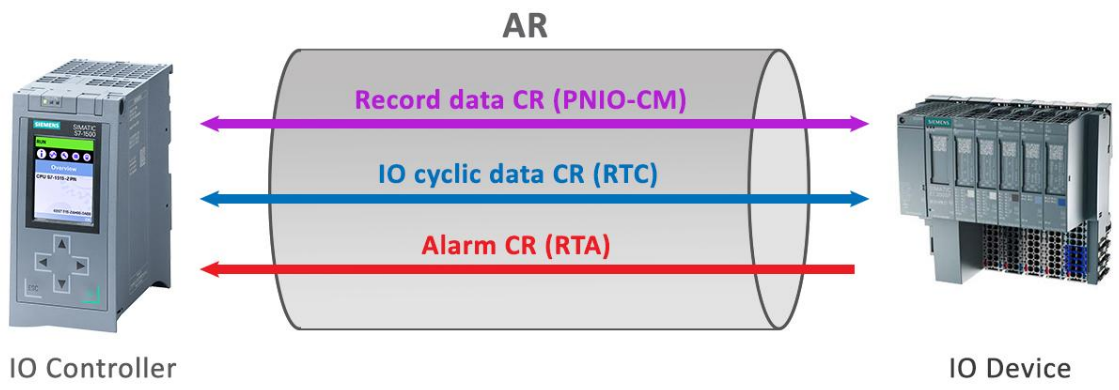


*Figure 4: Application and communication relationships (adapted and modified from* [4]*)*

An IO controller is able to establish AR with a number of IO devices running simultaneously.

At the same time PROFINET supports multiple IO controller systems. For this case there are some special applications for accessing the same data in the IO device by several IO controllers. These applications are called shared device and shared inputs.

### 2.2.4 AR Establishment and Startup of Cyclic Communication

One of the most significant parts of the processes involved in AR and CR is their establishment. This procedure then is followed by startup of cyclic communication, which is the main point of the PROFINET communication. Successfully established AR with CRs and startup mean that the IO controller and IO device are configured correctly and the parameters correspond to the application supported by the devices. That's why testing the AR establishment and startup sequence is one of the basic test cases composing the whole test scenario.

The startup sequence is schematically shown in the sequence graph on Figure A.1 from **Annex A**.

Several types of protocols that define main phases of the process are involved. Firstly the IO controller or the IO supervisor searches the target device by means of device name or NameOfStation. This name shall be preset to the device manually and correspond to the name in the project uploaded to the IOC/IOS. Setting the NameOfStation and correct configuration in the project should be enough for the

IOC/IOS to specify configured devices on the network and complete AR establishment and startup procedure with them.

Setting or getting the NameOfStation is handled by Discovery and basic Configuration Protocol (DCP). This protocol is MAC address (Layer 2) based. So in order to find a specified device IOC/IOS sends an identification request to the predefined multicast address with the target NameOfStation. The device with appropriate NameOfStation shall respond to the request. The response contains current IP suite (IP address, Subnet mask and Gateway) of the device, its VendorID, DeviceID, which are crucial for the AR establishment, and some other informative data. If the VendorID and DeviceID correspond to the configured in the project, IOC/IOS stores the MAC address from the response and assigns it to the existing NameOfStation. Then it compares IP suite from the project with IP suite from the response. In case of some differences, IOC/IOS sends a Set request with the IP suite configured in the project and the IO device shall accept it by a response with "Set Ok" acknowledgement. From this point the IO device is ready for the start of AR establishment.

By the way, the setting of NameOfStation is handled by the DCP protocol as well. For that so called DCP browsers are used, which are usually integrated into engineering tools and allow manual setting not only of NameOfStation but of IP suite too.

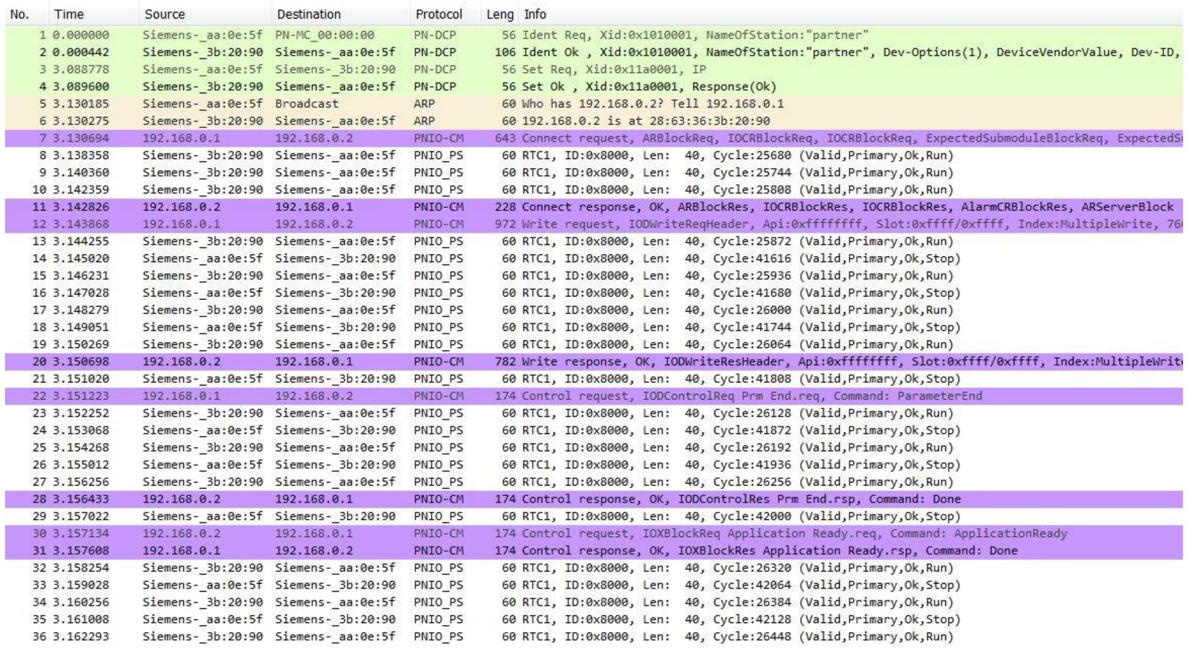
This phase can be observed in Figure 5 in frames no. 1 - 4. On the Figure is shown a screenshot from Wireshark program designed for decoding and displaying the packets (aka frames) on the Ethernet network.

According to source [5], communication that is based on IP suite requires support of address resolution protocol (ARP). This is necessary for translation between IP and MAC addresses. ARP uses its cache to minimize number of requests for address translation on the network. This cache is divided into static (with unlimited lifetime) and dynamic (with "aged" type) entries. As IOD and IOC can't change their addresses in the framework of one established AR, static type of ARP cache is used.

Therefore after the phase of DCP, IOC/IOS sends ARP request with the same IP address that is used in DCP. The IOD shall respond to the request (see Figure 5, frames no. 5, 6). From this moment IOC/IOS is ready for AR establishment too.

The static entry related to the AR shall be kept while the AR is active and shall be removed after AR abortion or release.

The AR establishment itself is handled by configuration and parameterization of the configured modules and submodules. This procedure is realized by acyclic requests and responses via remote procedure call protocol (RPC), which is IP address (Layer 3) based.



| No. | Time | Source | Destination | Protocol | Leng | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | Siemens-_aa:0e:5f | PN-MC_00:00:00 | PN-DCP | 56 | Ident Req, Xid:0x1010001, NameOfStation:"partner" |
| 2 | 0.000442 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PN-DCP | 106 | Ident Ok , Xid:0x1010001, NameOfStation:"partner", Dev-Options(1), DeviceVendorValue, Dev-ID, |
| 3 | 3.088778 | Siemens-_aa:0e:5f | Siemens-_3b:20:90 | PN-DCP | 56 | Set Req, Xid:0x11a0001, IP |
| 4 | 3.089600 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PN-DCP | 56 | Set Ok , Xid:0x11a0001, Response(Ok) |
| 5 | 3.130185 | Siemens-_aa:0e:5f | Broadcast | ARP | 60 | Who has 192.168.0.2? Tell 192.168.0.1 |
| 6 | 3.130275 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | ARP | 60 | 192.168.0.2 is at 28:63:36:3b:20:90 |
| 7 | 3.130694 | 192.168.0.1 | 192.168.0.2 | PNIO-CM | 643 | Connect request, ARBlockReq, IOCRBlockReq, IOCRBlockReq, ExpectedSubmoduleBlockReq, ExpectedS |
| 8 | 3.138358 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:25680 (Valid,Primary,Ok,Run) |
| 9 | 3.140360 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:25744 (Valid,Primary,Ok,Run) |
| 10 | 3.142359 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:25808 (Valid,Primary,Ok,Run) |
| 11 | 3.142826 | 192.168.0.2 | 192.168.0.1 | PNIO-CM | 228 | Connect response, OK, ARBlockRes, IOCRBlockRes, IOCRBlockRes, AlarmCRBlockRes, ARServerBlock |
| 12 | 3.143868 | 192.168.0.1 | 192.168.0.2 | PNIO-CM | 972 | Write request, IODWriteReqHeader, Api:0xffffffff, Slot:0xffff/0xffff, Index:MultipleWrite, 76 |
| 13 | 3.144255 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:25872 (Valid,Primary,Ok,Run) |
| 14 | 3.145020 | Siemens-_aa:0e:5f | Siemens-_3b:20:90 | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:41616 (Valid,Primary,Ok,Stop) |
| 15 | 3.146231 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:25936 (Valid,Primary,Ok,Run) |
| 16 | 3.147028 | Siemens-_aa:0e:5f | Siemens-_3b:20:90 | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:41680 (Valid,Primary,Ok,Stop) |
| 17 | 3.148279 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:26000 (Valid,Primary,Ok,Run) |
| 18 | 3.149051 | Siemens-_aa:0e:5f | Siemens-_3b:20:90 | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:41744 (Valid,Primary,Ok,Stop) |
| 19 | 3.150269 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:26064 (Valid,Primary,Ok,Run) |
| 20 | 3.150698 | 192.168.0.2 | 192.168.0.1 | PNIO-CM | 782 | Write response, OK, IODWriteResHeader, Api:0xffffffff, Slot:0xffff/0xffff, Index:MultipleWrit |
| 21 | 3.151020 | Siemens-_aa:0e:5f | Siemens-_3b:20:90 | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:41808 (Valid,Primary,Ok,Stop) |
| 22 | 3.151223 | 192.168.0.1 | 192.168.0.2 | PNIO-CM | 174 | Control request, IODControlReq Prm End.req, Command: ParameterEnd |
| 23 | 3.152252 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:26128 (Valid,Primary,Ok,Run) |
| 24 | 3.153068 | Siemens-_aa:0e:5f | Siemens-_3b:20:90 | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:41872 (Valid,Primary,Ok,Stop) |
| 25 | 3.154268 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:26192 (Valid,Primary,Ok,Run) |
| 26 | 3.155012 | Siemens-_aa:0e:5f | Siemens-_3b:20:90 | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:41936 (Valid,Primary,Ok,Stop) |
| 27 | 3.156256 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:26256 (Valid,Primary,Ok,Run) |
| 28 | 3.156433 | 192.168.0.2 | 192.168.0.1 | PNIO-CM | 174 | Control response, OK, IODControlRes Prm End.rsp, Command: Done |
| 29 | 3.157022 | Siemens-_aa:0e:5f | Siemens-_3b:20:90 | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:42000 (Valid,Primary,Ok,Stop) |
| 30 | 3.157134 | 192.168.0.2 | 192.168.0.1 | PNIO-CM | 174 | Control request, IOXBlockReq Application Ready.req, Command: ApplicationReady |
| 31 | 3.157608 | 192.168.0.1 | 192.168.0.2 | PNIO-CM | 174 | Control response, OK, IOXBlockRes Application Ready.rsp, Command: Done |
| 32 | 3.158254 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:26320 (Valid,Primary,Ok,Run) |
| 33 | 3.159028 | Siemens-_aa:0e:5f | Siemens-_3b:20:90 | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:42064 (Valid,Primary,Ok,Stop) |
| 34 | 3.160256 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:26384 (Valid,Primary,Ok,Run) |
| 35 | 3.161008 | Siemens-_aa:0e:5f | Siemens-_3b:20:90 | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:42128 (Valid,Primary,Ok,Stop) |
| 36 | 3.162293 | Siemens-_3b:20:90 | Siemens-_aa:0e:5f | PNIO_PS | 60 | RTC1, ID:0x8000, Len: 40, Cycle:26448 (Valid,Primary,Ok,Run) |

*Figure 5: Startup sequence displayed in Wireshark*

The AR establishment itself starts with Connect request from IOC/IOS to IOD (see Figure 5, frame no. 7). The request carries several blocks with configuration of the future AR and CRs. The first one is AR block, which contains for example AR UUID used for acyclic record data handling, AR properties that define mode and type of the AR and other information.

Then IO CR blocks follow, usually one for Input data and one for Output. In these IO CR blocks there are defined length of the data, unique frame ID for the CR, IO cycle time interval and layout of the IO data itself in the data packets.

Expected Submodule blocks follow after that. Each block represents one module including its submodules and definition of their types, identification numbers and slot/subslot numbers.

In our case the Connect request block is finished with Alarm CR block that defines some configuration of alarm handling between the IOC/IOS and IOD.

All the data used in the Connect request is taken over from the configuration in the project that is uploaded to the IOC/IOS.

According to source [8], Connect request frame may contain other blocks with additional information according to the configuration and application used for the station.

After sending the request the IO devices related to the AR may start sending IO data, but without processing the data of the provider (see Figure 5, frames no. 8 – 10, 13 – 19 and so on).

Meanwhile the IO device receives the request and starts to process it. If the configuration data within the request is valid, corresponds to the real configuration and supported application, the IOD sends the response (see Figure 5, frame no. 11). This response contains some additional information required from IOD for AR and CR establishment, for example Frame ID for Input CR. In case, if there are some discrepancy between the expected configuration and the real one, the device adds ModuleDiffBlock to the response with the specification of the problem inside. For example the software version of IOD may be outdated and it will be mentioned in the ModuleDiffBlock. This discrepancy isn’t crucial for the AR and CR, as some versions are compatible and may substitute each other.

From this moment we can consider the AR and the CRs established.

In the next step, IOC/IOS starts parameterization of the IO device by sending Write requests containing the data (see Figure 5, frame no. 12). The parameterization is done based on special parameters taken from the GSD file of the device, common parameters in the project related to the specific type of submodules and applications supported by PROFINET. The acyclic Write requests (and Read requests as well) are handled by RPC protocol. The parameterization or other data are packed into blocks according to its type. The type is specified by Index number according to PROFINET application protocol. Each of the data blocks relates to the specific submodule and is addressed according to the model described in 2.2.2. Writing can be done to a several submodules in one frame, but reading shall always address to a single submodule.

For example, we can deactivate one of the ports on the device in the engineering tool; let’s say it would be Port 1. After uploading the project to IO controller and observing the parameterization of IO device, we will be able to find *PDPortDataAdjust* block addressed to Slot 0x0, Subslot 0x8001 with the deactivation parameter. Parameterization of Port 2 will be sent in a separate *PDPortDataAdjust* block addressed to Slot 0x0, Subslot 0x8002.

In order to realize the complete parameterization, these blocks can be sent in one Write request frame, several frames, or separately – one for each parameterization block.

The IO device shall respond to the Write requests with Write responses and acknowledge every parameter block within it (see Figure 5, frame no. 20). In case, if some of the parameter blocks are wrong or not supported, the device rejects this block with an appropriate Error entry related to the concrete parameterization block.

The end of parameterization phase is signaled by IOC/IOS. When all the Write responses to all requests are received, the IOC/IOS sends Control request with *ParameterEnd* control command (see Figure 5, frame no. 22). The IO device responds to request after the received parameterization is processed inside the device (see Figure 5, frame no. 28).

The final phase of the startup is executed by the IO device. When the device is completely prepared for the configured application, it sends Control request with ApplicationReady control command (see Figure 5, frame no. 30). And after the IOC responds to the request (see Figure 5, frame no. 31). Only from this moment both devices shall switch over IO cyclic data frames from idle data to real data.

But if some mismatches in the configuration of AR and CRs or parameterization of the device took place, the Control request with ApplicationReady will contain ModuleDiffBlock with the description of the mismatch. In case of problems crucial for common IO cyclic data exchange, the data will set its status as “Problem” and the exchange of real data is impossible on these terms until the problem is fixed. But in some cases the mismatch is not a problem for the data exchange, for example the firmware version mismatch or some diagnosis occurred during parameterization that has no influence on the IO cyclic data exchange. Anyway, this block informs about some either small or big problem in the configuration or parameterization that is good to be fixed in order to insure complete functionality of the application as expected.

The acceptation of the Connect request and the presence of the ModuleDiffBlock in Control request play the key role in the test cases that I’ve designed for testing the AR establishment.

### 2.2.5 IO Cyclic Data and Data Hold Time

As it was mentioned in the previous chapter, after the startup procedure is completed the cyclic IO data exchange between IOC and IOD in the framework of IO CR established.

This data exchange is handled by Real Time Cyclic (RTC) protocol. In case of simple real-time (RT) communication this protocol is related to class 1 RTC protocol (RTC1). The PROFINET standard defines RTC class 3 (RTC3) as well, which handles isochronous real-time communication (IRT). In this thesis RTC class 1 will be discussed only.

RTC protocol itself is MAC address (Layer 2) based. In order to provide as fast communication as possible the frames of this protocol contain minimum information, most needed only. With this approach the minimal size of the frames is reached for transporting the data.

Besides the source and destination MAC addresses, the RTC frame consists of:

- FrameID according to the Connect request and response
- so called Provider and Consumer status one byte for each submodule
- the IO data itself
- internal cycle counter
- data status
- and transfer status

The data status is represented by one byte. Bits within the byte define the validity of the data, state of the provider and state of the application. Transfer status represented by one byte too informs about the transferring problems, such as wrong length of the frame, buffer overflow or wrong checksum.

Therefore checking the value of the data and transfer status bytes is essential for verification of correct communication process.

Another important PROFINET standard feature related to the cyclic data exchange is the time intervals between the provider’s data frames and so call data hold timer.

According to the standard each data frame shall be sent in a fixed time intervals (some insignificant errors in these intervals are allowed according to the certification test case description for the PROFINET IO devices [9]).

The intervals or send cycles are defined in IO CR blocks of Connect request by SendClockFactor and RedactionRatio values. These values shall correspond to the supported values issued in the GSD file of the device and can be configured in the engineering tool for each device.

Then the send cycle is calculated according to the equation 1:

$$SendCycle = SendClock \times RedactionRatio \times 31.25\mu s \qquad (1)$$

So called Data Hold Time (DHT) timer is related to the SendCycle value directly. This timer is responsible for detecting defects in the cycles of data exchange (some kind of watchdog). The DHT timer is executed with each provided data frame. If there are some problems on the network or in the devices and the last data frame is left without the response from the consumer (in the form of consumers data frame with the valid data), the provider shall resend his last data frame with no changes inside. In this case the DHT timer is not reset. As the result this repetition of the last frame shall last until the timer expires. After that the AR shall be aborted and the appropriate alarm is sent to the consumer informing about the abortion by means of the acyclic communication. [8]

The value of the timer can be set while configuration in the engineering tool by DataHoldFactor. This configuration value is sent to the IO device during AR establishment within the IO CR blocks.

The DHT timer itself is calculated according to the equation 2:

$$DataHoldTime = DataHoldFactor \times SendClock \qquad (2)$$

Testing of DHT timer in the framework of regression tests is important for ensuring the capability of the device under the test to abort the AR, watch the cycles of data exchange and providing of the errors.

# 3 PROFINET IO Development Kit

In order to simplify the developing and testing processes for PROFINET devices Siemens AG provides a development kit (DevKit). This kit is based on ERTEC 200P ASIC designed for PROFINET communication performance and assists in developing of both hardware and software of the PROFINET devices. [10]

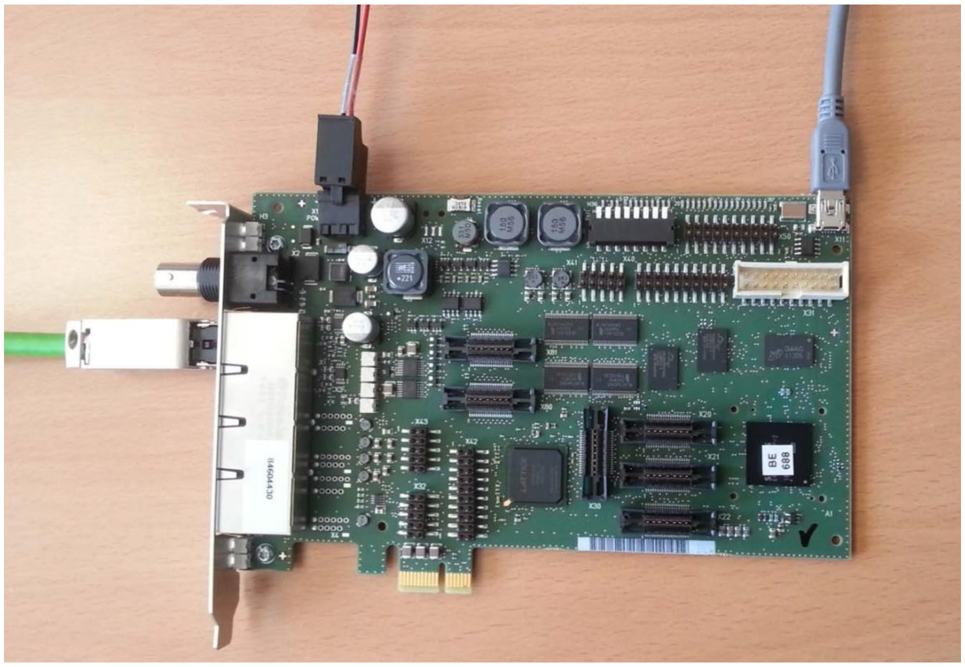


*Figure 6: Development kit board with connections RJ45, Power supply and Terminal console* [10]

Together with the hardware in the form of evaluation board (see Figure 6), the PROFINET IO stack with user examples are provided to the customer. This stack contains not only basic PROFINET IO functionalities but some special functionalities of the standard as well, such as Fast Startup, Shared Device and Media Redundancy (the description of the functionalities can be found in source [6]).

A user may choose one of the examples by changing the values of some variables from the stack source code. After compiling the source code and downloading it to the board, the development kit starts to behave as a PROFINET IO device. As it has no real slots for Input and Output modules, they are simulated and can be configured in the engineering tool. After the AR establishment with IO controller, the development kit as an IO device performs regular data exchange according to the configuration and other supported applications.

The GSD file with prepared user examples is provided as well.

## 3.1 Connectors Layout

The development kit has a number of connectors of different types designed for various purposes. The overview of their designation and location on the board is in Figure 7.

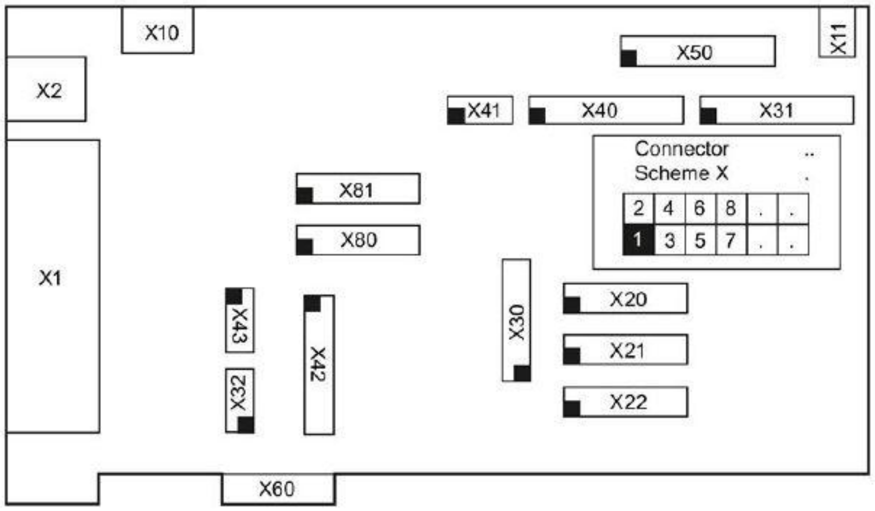


*Figure 7: Overview of DevKit connectors* [10]

The connectors which are significant for test performance are mentioned in Table 1.

Connector X1 has 4 Ethernet RJ45 ports, two of them are designed for PROFINET communication, two others have integrated TAP functionality but only for outgoing packets [11]. The development kit also exists in another hardware assembly with two Ethernet plastic fiber optic connectors instead of four RJ45 ports.

Connector X10 is a 2-pin industry plug-in for external 24VDC power supply. The kit can be connected to a PCI slot and doesn't require external power supply.

**Table 1: DevKit connectors description**

| **Name** | **Type** | **Role** |
|---|---|---|
| X1 | RJ45 socket | Ethernet and TAP interface |
| X10 | 2-pin industry plug-in | External DC power supply |
| X11 | USB Mini-B connector | USB UART |
| X20 | 38-pin Mictor connectors | GPIO |
| X40 | 2x10 pin plug connector | Configuration pins for ERTEC 200P |
| X42 | 2x10 pin plug connector | Configuration pins for onboard circuits |

Connector X11 is a USB Mini-B connector for UART communication with the user interface program. This program manages some operations for simulating the IO device, for example triggering alarms, or some system functions, for example firmware update via TCP communication or changing MAC address. The user interface is operated via terminal console.

Connector X20 is used for plugging in external flash memory of SPI type.

The activating of different flash memory types is handled by connecting appropriate pins on X40 connector. The firmware is booted from the selected flash memory after the device's reset.

Connector X42 manages configuration of the boot mode. There are the following the modes available: enabling of synchronization signal for IRT communication, enabling of trace, enabling of user GPIOs, of UART port is significant for the tests and some others. Enabling of user GPIOs and UART port I consider as a default boot mode and the following tests are performed with this configuration. [11]

Other connectors that are displayed in Figure 7 are not used during the tests.

## 3.2 Flash Types

The DevKit can boot firmware from different flash memories:

- NOR flash 16 bits
- NOR flash 32 bits
- SPI flash [10]

The selection of the source memory is managed by the pins of connector X40. For use of NOR 32 bits flash all the pins shall be open, for use of SPI 32 flash pins 5-6 shall be closed (connected with the jumper). [11]

16 bits and 32 bits flash memory types require different firmware compilation according to the bits number. For booting from SPI flash the appropriate flash memory shall be plugged in connector socket X20.

In the framework of this thesis, these two types of flash memory are tested only, NOR 16 bits is omitted.

## 3.3 Application Examples

The software part for the DevKit consists of several application examples. These examples are implemented into provided PROFINET stack and are used for adaptation of the PROFINET to a wider range of requirements during the development. To use them, firstly the source code from the stack should be compiled with the required application example selected. The selecting is done by changing the value of the appropriate variable.

There exist three applications as templates for improving according to the requirement:

- Application 1 Standard
- Application 2 DBAI
- Application 3 IsoApp

Each of them supports basic PROFINET application for IO device, such as cyclic IO data exchange, acyclic services for startup, reading/writing records and alarm handling. All application examples support RT and IRT communication functionality. The differences of the examples lie in the properties of some application service and system performances.

Application 1 Standard can be used as a simple and universal example for fast implementation of standard PROFINET interface. This application example uses module/submodule-oriented view onto the cyclic IO data and may not know about the ARs and IOCRs.

Application 2 DBAI (Direct Buffer Access Interface) has some advantages compared to Application 1 in case if large number of modules and submodules is configured. The application example uses IOCR-oriented view onto the cyclic IO data and the application is capable to manage the ARs and IOCRs.

Application 3 IsoAppl uses the same access method to cyclic IO data as Application 1, but in addition this example requires input/output modules supporting IRT application.

## 3.4 Simulated Devices Types

Not only application examples can be selected by the changes in values of the source code variables, types of simulated IO devices can be selected too. The current GSD file (GSDML-V2.32-Siemens-ERTEC200pEvalkit-20161027.xml) presents six types of the IO devices. Four of them relate to the RJ45 connector assembly and two of them relate to the POF connector assembly. Their IDs and key functional differences issued in GSD file are described in Table 2.

DAP1 example simulates device with no Interface and Port submodules available to be involved in an AR. Therefore the functions are not specified. But Input and Output modules can be configured and support cyclic IO data exchange. As my tests are focus on testing of Interface and Port submodules as well, test configuration with this type of the device is omitted.

**Table 2: Simulated device DAPs**

| **ID** | **Ports** | **MRP functionality**[1] |
|---|---|---|
| DAP1 | Not specified | Not supported |
| DAP2 | 2 RJ45 | Not supported |
| DAP3 | 2 RJ45 | Supported |
| DAP4 | 1 RJ45 | Not supported |
| DAP5 | 1 POF | Not supported |
| DAP6 | 2 POF | Not supported |

Also my regression tests in the framework of the thesis are focused on the DevKit with RJ45 connector assembly only. Therefore testing of DAP5 and 6 is omitted too.

## 3.5 Project Builds and Firmware Test Options

As was mentioned in the previous chapters, the intervention into PROFINET stack source code is needed. It’s possible to select one of the options “application + device type” by changing values of some variables. But selecting of required flash type (16 bits or 32bits) is done by choosing appropriate project build.

Eclipse IDE is provided with the PROFINET stack and ready project builds can be imported into the IDE. There are four types of the project builds:

- 16 bits on native operating system
- 16 bits on Posix operating system
- 32 bits on native operating system

[1] MRP (Media Redundancy Protocol) functionality – functionality suitable to most Industrial Ethernet applications. [12] As ring topology in Ethernet network may cause so called broadcast storm and drop the communication, this protocol was designed to prevent this error by virtual disabling of one the ports on MRP manager device. This functionality is supported by PROFINET standard as well.

- 32 bits on Posix operating system

Each of the flash types shall support booting the firmware with both operating systems available (native, Posix).

As the focus of test scenarios is on 32 bits flash memories only, the 16 bits project builds of the firmware are omitted.

After the building the project and setting the values of the appropriate variables according to the DAP device type required the firmware can be compiled. (Detailed guideline of selecting, building of the project and setting the application example is available in source [10].)

The successfully compiled firmware can be downloaded to the appropriate flash of the DevKit. After resetting the device, the DevKit shall boot the firmware and simulate the device that was configured in the source code before the compilation.

Downloading the firmware is possible by two methods. The first one by means of Olimex JTAG debugger (details are in [10]), the second one is via Ethernet connection directly (by means of TCP protocol). The second method is much faster and requires the use of user interface. Therefore I focus on this method while test design, as it reduces the time of test performance and involves testing of user interface communication simultaneously.

# 4 PyTeMat

PyTeMat (acronym for Python Test Automat) is an open source program designed in by employers from Siemens s.r.o Corporate Technology division located in Prague. Currently this program is in active development and new functionalities are being added. The program has no graphical user interface and is executed from the command line.

PyTeMat is a Python programming language based and object-oriented methodology of programming. Currently is used for test automation of PROFINET and Ethern/IP devices in Siemens Corporate Technology in Prague.

This program is designed to process special TBL (Test Box Language) syntax based on Python and designed for PyTeMat. This language allows easy implementation of test sequences without repeating the code responsible for step performances and result analysis. Also this program provides automatic collecting and filtering of the results in form of ***.txt** file.

## 4.1 PyTeMat Core

The core of PyTeMat consists of two folders.

The first one is *System* folder. System folder contains codes responsible for basic PyTeMat functionality. The entry point to the PyTeMat itself is *PyTeMat.py* script, which should be executed for launching the program. Other significant script is *Core.py*. This script handles the execution of preprocessing of the test script written TBL and merging the parts of the processed code final executable script. *Preprocessor.py* Python module located in the *System* folder is responsible for checking the TBL syntax and replacing the key words with prepared Python code. *LogStream.py* Python module together with *Loggers.py*, *Enums.py* and *Filter.py* modules handles the logging of the test step results.

The second folder is *PyModules* folder. This folder contains Python modules for handling the structure of devices involved in the test. For some classes of the devices managing of some their functions is implemented. The basic properties of the device instance are defined by calling *Dev_Struct.py* module. Each device separately can be defined inside the *Dev_Struct.py* by selecting device class. In this case appropriate module is called for adding some specific properties and attributes distinctive for the class, for example *PN_Devices.py* for PROFINET devices or *EIP_Devices.py* for Ethernet/IP devices. Among the modules managing the functionality of the specified devices and the system itself, there are:

- *DCP.py* module for setting/getting NameOfStation and IP suite to/from the devices defined in the PyTeMat project;
- *OPC.py* module as an interface between PyTeMat application and OPC server;
- *RPC.py* module for sending Implicit acyclic Read/Write record requests directly by the PyTeMat application;
- *Step7.py* module for managing Simatic Controller modes and uploading and downloading projects to it;
- And other modules.

For my test scenario only two functional Python modules are used: *DCP.py* and *OPCs.py*.

### 4.1.1 DCP.py Module

While creating a project and configuring any devices in PyTeMat an instance of *Interface* class from *DCP.py* module is automatically created. The name of network interface on the PC running PyTeMat is defined by the *NetworkConnectionName* attribute in *LocalSttings.py* module. The DCP packets with selected function are handled by this network interface.

The *Interface* class from *DCP.py* module defines several methods corresponding to the DCP protocol functionality. First method is called *Browse*. Calling this method sends multicast DCP Identification request to all devices on the network. The device shall respond to it according to the specification. The method returns a collection of the devices responded to the request. The collection includes NameOfStation, IP suites and MAC addresses of the devices.

Two other methods are used in the test implementation from the *Interface* class.

*SetDeviceName* method is one of them. This method handles setting of the NameOfStation to the device addressed by MAC. The mandatory input parameters of the method are: *mac* – the destination MAC address in string format and *devName* – NameOfStation to be set on the device in string format. Optional input parameter is *permanent*, which is of Boolean type and is set to *True* by the default. This parameter defines, if the required NameOfStation should be set as permanent name or temporary one (in case *permanent* is *False*). If the permanent value is *True*, then the NameOfStation will be kept even after resetting the device until new NameOfStation set request is received. Otherwise, after resetting the device by switching the power supply off and on, the NameOfStation is reset to empty string. The *SetDeviceName* method returns *None* value.

Another method is called *SetIP*. This method handles setting of IP suite to the addressed device by means of DCP set request. The mandatory input parameters are: *mac* – the destination MAC address; *ipAddr* – the IP address that should be set; *mask* – the Subnet mask that should be set; *gate* – the Gateway that should be set. All these parameters are in the string format. The same as *SetDeviceName* method, the *SetIP* method has optional input parameter *permanent*. This parameter has the same format and purpose as in the *SetDeviceName* method. The *SetIP* method returns *None* value.

The Interface class of DCP.py module contains other methods that correspond to the DCP protocol functionality, but they are not used in my tests, therefore their description is omitted.

### 4.1.2 OPCs.py Module

The *OPCs.py* module is designed for interconnecting of the OPC server tool and the PyTeMat application. The module contains classes, which handles different OPC servers according to the device connected with the server. There are integrated classes that represent OPC for Allen-Bradley ControlLogix System, for Siemens Sinamics class of devices and for Siemens Simatic class. As the devices involved in the tests are Simatic class only, the OPC for Simatic devices is used only.

The class for OPC managing Simatic devices in PyTeMat is called *SimaticOPC*. The class is a descendant of class *client* from *OpenOPC.py* module that is freely shared in the internet.

Definition of a *SimaticOPC* instance shall be called explicitly. After instance definition, *OPCServerName* and *OPCTopicName* parameters should be specified according to the real OPC server.

*SimaticOPC* class contains four methods that handle basic functionality related to an OPC server.

The connection to the specified server can be established by calling *ProjectOnline* method. No input parameters are required for the method. The method returns either *True* value, if the connection secedes, or *None*, if the connection fails.

The disconnection of the OPC server is handled by *ProjectOffline* method. Similar to *ProjectOnline*, the *ProjectOffline* method requires no input parameters and returns *None* value in case of failed disconnection. If the disconnection is done without errors, the method returns integer value *1*.

For reading the controller addresses *OPCRead* method is used. The input parameter to the method is the symbolic name of the required address (*Symbol*). The method returns the value of the *Symbol* in the format according to the value stored in the controller address. If some problem occurred while reading, the method returns empty string.

Writing to the controller addresses is handled by OPCWrite method. This method requires two input parameters: *Symbol* – symbolic name of the address; *Value* – the value that should be written. The method does not return any values.

## 4.2 Project with Test Case Definition

The test cases itself are stored separately, in my case in directory *Tests*. For each project it's better to create separate folder in the directory, where number of test cases can be kept. The folder of a single project contains directory *HWconfigs* for hardware configuration files for PyTeMat application and directory *TC* for test case design.

The *HWconfig* directory contains Python source files for description of the hardware configuration of the test environment. Each file describes one hardware configuration with instances of the devices that are involved in the test process and can be managed or checked by PyTeMat application. The IP suite, MAC addresses, topology of the network, PGPC settings and other parameters are defined in the source file as needed.

The test case description within *TC* directory is divided into three directories: *prm*, *run* and *src*.

The *prm* directory contains files with ***.ptm** suffix defining test parameters. The test parameters can be represented by global variables that define for example time intervals for execution of separate test steps or number of tries of the steps.

The *run* directory contains files with ***.ptc** suffix defining the complete test scenario each one. The test scenarios within the run file has a specified structure that shall be kept. Firstly the hardware configuration file used for the test scenario is selected. *HWconfig* keyword foregoes the path with the hardware configuration file name. Then definition of the directories with test case source files and parameter files follows. The keywords for them are *PrefixTCS* and *PrefixPRM* respectively. And at the end the list of test case source files executed with the parameter files are specified in form of: *Testcase <source file> PRM <parameter file>*.

In my case of implementation of test for the development kit the content of the run file (*DevKit_Automat.ptc*) look as follows:

```
HWconfig Tests\DP_DEVKIT_TESTS\HWconfigs\DevKit_Automat_rack.py

PrefixTCS Tests\DP_DEVKIT_TESTS\TC\src\
PrefixPRM Tests\DP_DEVKIT_TESTS\TC\prm\

Testcase TC_DevKit_Automat.ptm PRM PRM_DevKit_Automat.ptm
```

The last *src* directory contains the source files defining the test sequence in the framework of one test case. The files have ***.ptm** suffix and are written in TBL. The TBL detailed description is in chapter 4.2.1.

The launching of the test case is described in chapter 4.3.

### 4.2.1 TBL Syntax

The Test Box Language (TBL) is the language designed for PyTeMat and consists of special keywords that represent structure of test case and test performance processing.

Each test case file in *src* directory shall contain at least four keywords. They are:

- *TESTCASE("<test case name>")*:
    - introduces new test case
    - can be use only once in a test case file
    - the following block contains complete test case procedure build from *STEP* blocks
- *STEP("<step name>"):*
    - introduces new step within the test case
    - next keyword shall be *CRITERION*
- *CRITERION(<criterion>):*

    - as *criterion* there should be a variable that python evaluate as condition
    - keyword has to be followed by *UNTIL*
- *UNTIL(reached_in = <number> remains_for = <number>):*
    - terminates the step
    - code in forgoing *CRITERION* block is executed cyclically while the criterion isn't evaluated as *True*
    - argument *readched_in* specifies time for reaching the *criterion* in seconds
    - argument *remains_for* specifies time in sec for how long the *criterion* have to be fulfilled for passing the step

Besides mandatory keywords TBL supports some optional words as well. The optional keywords used in my test implementation are:

- *INITIALIZATION:*
    - is used as test case initialization - the following block of code is executed before test case section
    - is the keyword is used, it should be the first keyword in the source file
- *FINALLY:*
    - the following block of code is executed after the test case section
    - this block is always executed even if the test case fails or crashes by exception
    - if the keyword is used, it should be the last keyword in the source file
- *SECTION("<section name>"):*
    - introduces new section, can be used for logical separating of code into sections
- *PASS():*
    - The following block of code is executed if the foregoing criterion has passed
    - this keyword is allowed only between *UNTIL* and the following *STEP* keywords or at the end of the test case
- *FAULT():*
    - he following block of code is executed if the foregoing criterion has failed
    - this keyword is allowed only between UNTIL and the following STEP keywords or at the end of the test case

An example of the test case file content can look like this (red keywords are mandatory, blue are optional):

```
INITIALIZATION:
    Device.Init()

TESTCASE("new testcase"):

    STEP("Power on device - poor step"):
        Device.PowerON()

    CRITERION(Device.getAR()):
        pass
    UNTIL(reached_in = 10 remains_for = 1)

    SECTION("Logical section"):
        STEP("Do nothing"):
            pass
        CRITERION(crit):
            crit = A + B
            crit += C
        UNTIL(reached_in=TP.watchdogConstant remains_for=TP.RFconstant)
         FAULT:
               A = B
        PASS:
               B = C
FINALLY:
    Device.CloseConnection()
```

## 4.3 Launching the PyTeMat Test Case

For launching the PyTeMat project, the structure of the project and its content shall correspond to the requirements described in the previous chapters. When PyTeMat is installed and the project is ready, the launching is done by running command in command line in the form: "pytemat <run file>"[2].

PyTeMat supports four runtime options. They are selected by adding to the existing command line and can be combined. The options are:

- "**-d**" or **Debug mode**: In this mode the test performance is not executed, the test cases defined in the run file are processed into final executable python files and the files is saved in the directory defined in *LocalSettings.py*. Later the files can be executed in debug mode in IDE software.
- "**-f**" or **Force mode**: In this mode the test performance is executed and is forced to continue even if s criterion fails.

[2] Note: it's important to open the run directory via command line in advance to have the access of the command promt to the run file.

- "**-s**" or **Save mode**: In this mode the test performance is executed and the processed executable python file is saved in the directory defined in *LocalSettings.py*.
- "**-t**" or **Terminate mode**: In this mode the test performance is executed and, if some of the criterion fails, the test performance is terminated without execution of *FINALLY* block if present.

The execution of the PyTeMat application for a test case passes several passes after launching it:

1. Setting the arguments (setting the modes) entered from the command entered to the command line.
2. Checking the *run* file content for the keywords, directories and files.
3. Checking the content of the source files for correct TBL syntax.
4. Processing the source file into executable python code.
5. Merging the header file, hardware configuration file, parameter file, processed source file and footer file into single executable python file for each test case. The order of the parts composing the executable file is exactly as it was specified in the previous sentence. Header and footer files contain the logging manager. The files are static, are independent from the PyTeMat project configuration and are added to each executable python file unchanged.
6. If the command doesn't include the Debug mode the test is performed by executing the executable python file.

As we can see the PyTeMat program meet the requirement for automatic test performance.

First of all the structure of the PyTeMat project corresponds to the usual test structure (see Figure 8):

- The test sets are represented by a set of the PyTeMat *run* files.
- The test scenarios are represented by a *run* file.
- The test scenarios are represented by source (or test case) files defined in the *run* file.
- The test cases are represented by the source files.
- The test steps with the evaluation criteria are represented by the TBL syntax within a test case.

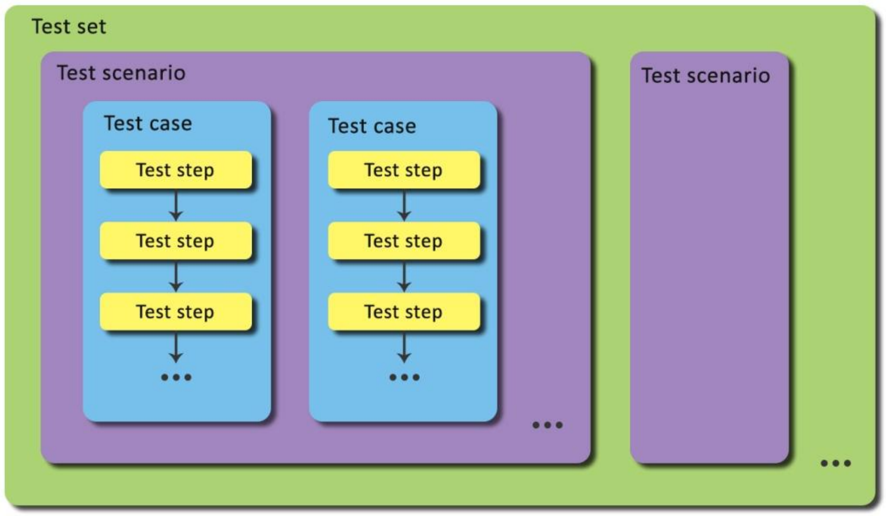


*Figure 8: Test structure*

The second main advantage is TBL, which simplifies the definition of the test steps and criteria. And integrated Python modules for managing result logs and some of the PROFINET functionalities are the last but not the least main advantage of the PyTeMat.

# 5 Test Design

Taking into consideration the fact, that a great number of variations of firmware and flash types should be tested with every version released during the development, the regression test requires great effort, just for verifying basic functionality. In that case the automation of the tests is extremely necessary.

## 5.1 Firmware Update Automation

One of the main tasks for the automating the regression test is the automation of the firmware update.

The DevKit user interface and software tools that are issued together with the board allow running the update in a relatively easy way. One of the possible methods is via TCP protocol. This method requires the connection to the DevKit's active Ethernet port (connector X1) and connection to the DevKit's serial port (connector X11) for running the Terminal console.

Preconditions for the method are:

- The firmware image is compiled and prepared in the as a file
- The user PC is connected to the DevKit's active Ethernet port (connector X1)
- The DevKit has valid IP suite set
- The user PC is connected to the DevKit's serial port (connector X11)
- The terminal console is running and configured correctly for entering the commands

The procedure of firmware update has the following steps:

a. The user enters command "*f*" to the console for preparing the DevKit for the update via TCP
b. "TCP interface wait on connection ..." output line notifies the user about the DevKit's readiness
c. The user runs command line from the directory, where the firmware file with and the *TcpFwLoader.exe* are located

d. User enters the following command to the command line "*tcpFwLoader <firmware file name> <DevKit IP address> 999*"
e. The DevKit notifies the user about the connection via TCP in the console and after receiving the firmware image asks the user for the flash type, where the uploaded firmware should be booted from.
f. The user select the flash type (NOR or SPI) by the number related to the flash.
g. The DevKit handles the firmware update and notifies the user about the end of the process by printing "*OK, Flashing firmware finished*" line to the console output.
h. The user shall reset the DevKit in order to boot the new firmware.

As it was described in chapter 4.4, I focused on selected types DAP simulation, application examples, flash types and project builds. The complete list of all options are represented in the form of tables (Table 3 and 4), where the tested and not tested ones are marked.

The switching between the firmware upload from NOR32 or SPI to NOR 16 flash can't be automated and require the user intervention. Therefore the automated test combinations with NOR 16 have to be done separately from NOR 32 and SPI. As the thesis is focused on implementation of the test cases and not the performance of the complete test combination sets, only several combinations where selected. The selection was done with a view to cover all test parameters' use in the implementation, and they are: firmware upload with different operation systems and automatic switching between the flash types.

**Table 3: Firmware options (DAP-Application)**

| | DAP1 | DAP2 | DAP3 | DAP4 | DAP5 | DAP6 |
|---|---|---|---|---|---|---|
| Application 1 | Not tested | Tested | Tested | Tested | Not tested | Not tested |
| Application 2 | Not tested | Tested | Tested | Tested | Not tested | Not tested |
| Application 3 | Not tested | Tested | Tested | Tested | Not tested | Not tested |

Any option from Table 3 has combination with each option from Table 4. The same works with the Table 4 relating to the Table 3. For example, I choose option DAP2-Application 2 and this means that only for this one option I have 6 further option that can be tested according to the Table 4 (they are: NOR 16-Native, NOR 16-Posix, NOR 32-Navite, NOR 32-Posix, SPI-Native, SPI-Posix). And this is only for one option DAP2-Application 2 selected.

**Table 4: Firmware options (Build-Flash)**

|  | Native | Posix |
|---|---|---|
| NOR 16 | Not tested | Not tested |
| NOR 32 | Tested | Tested |
| SPI | Not tested | Tested |

This implies that the possible firmware test options are equal to the number of options from Table 3 multiplying to the number of options from Table 4. And the total number is 108. For the thesis I limited the number to 27, selecting 3 options from Table 4 and applying them to selected 9 options from Table 4.

From this chapter we can see that for automation of this step we need to manage the serial communication between the DevKit and the PC with PyTeMat application, manage the execution of the command line with following command entering to it and we need to handle convenient selection and switching to an appropriate firmware.

## 5.2 Startup and IO Data Exchange Verification

As we know from the PROFINET description in chapter 2, one of the main functionality related to a PROFINET IO device is AR establishment and the startup of IO data exchange between the IO device and IO controller.

In the framework of the regression tests it is a crucial point of the functional firmware in the DevKit. Therefore the testing of the functionality should be automatized.

For minimal verification of the successful startup and AR establishment we need to capture the communication frames directly from the network. For this purposes special devices called TAPs exist on the market. For the communication based on Ethernet I use Profishark 100M from Profitap Company. The most used and free software capable handling and decoding the Ethernet communication is Wireshark, which is based on Winpcap library and TShark application. A part of the window with frame decoding is shown in Figure 5. With installing the Wireshark software, Winpcap library and TShark application are installed automatically.

In order to trace the successful startup two frames will be enough to analyze.  The frames are: Connect response and ApplicationReady request from the DevKit. If the Connect response is present in the trace and its *Status* value equals 0, then the foregoing Connect request is accepted and the AR is established but the startup is not finished yet. If the ApplicationReady request is present in the trace and it doesn't contain ModuleDiffBlock, this means that the startup procedure was completed without any errors.

For checking the correct IO data exchange is enough to analyze several RTC1 frames provided by the DevKit after the startup sequence. The frames should contain DataStatus equal to 0x35, which means that the parameterization was accepted and the data are valid, and TransferStatus equal to 0x0.

From the automation of this step we need to create a project for IO controller, automatize capturing the trace and analysis of the specified frames including some of their content.

## 5.3 Properties of the Simulated Device Verification

For verifying the correct firmware and its correct boot to the DevKit's processor it's crucial to check the correct properties of the simulated IO device according to the DevKit's manual.

The main differences in the properties lie in the DAP properties and in Application examples.

Application 1 and 2 have no differences from the communication point of view. The only difference with the Application 3 is in the input and output modules that can be configured (verification of IRT communication and application is omitted in the test design).

There are two possible differences related to the DAP properties and they are the number of ports available (1 or 2) and MRP functionality support.

The verification of port numbers can be done by connecting a partner IO device to the second port of the DevKit. In case if the DevKit should simulate one port IO device, the AR establishment and startup of the partner shall not occur and no frames sent by the partner shall be present on the network. Otherwise, if the DevKit simulate two port IO device, the Connect response and ApplicationReady request sent by the partner shall be present on the network. This verifies that the DevKit forwards the frames to the partner and from the partner without any errors.

The verification of the MRP functionality supported is realized by reading the parameters of the device by means of RPC and PNIO-CM read request. If the DevKit supports MRP functionality, the Read request of PDInterfaceMrpDataReal (index 0x8050) record addressed to the Interface submodule shall be accepted. Otherwise, the Read request shall be rejected. The acceptation is verified by checking the Status value of the response. If the value is 0x0, then the request is accepted and the response with some required data follows. If the Status has not zero value, the request is rejected and the value corresponds to some appropriate error, which causes the rejection.

Besides the trace capture of the communication, automation of this step requires handling of various simulated devices configuration by the project for IO controller and triggering the Read request for PDInterfaceMrpDataReal addressed to the DevKit.

## 5.4 Acyclic Read or Write Responses Handling Verification

Verification of correct responses to the acyclic read or write requests with addressing to each submodule is another step of regression test that can be automatized. During this test step we are able to verify that the DevKit processes the read/write requests to each of the submodules configured and is able to accept and reject the request according to its GSD file and the PROFINET standard.

For this purpose I select so called I&M datasets. There are six I&M datasets types and they are enumerated from 0 till 5. I&M0 is a read-only mandatory dataset that shall be supported by every submodule of a PROFINET IO device. The submodules issued in the GSD file may have parameter "*Writeable_IM_Records*". The value of the parameter is a list of numbers of supported I&M datasets, excluding 0, that is supported by default. If the parameter is missing, the submodule doesn't support I&M datasets 1 to 5. [7], [8]

According to source [8] and the GSD file of the DevKit, the responses to the read/write requests for the datasets shall be as it's shown in the Table 5.

The lines correspond to the submodules of the DevKit station according to chapter 2.2.2 and the configuration of the station in the project. The columns represent the dataset type and the request type (write or read). The intersection represents the response type that is expected of the DevKit: *X* stands for rejecting the request with an error, *OK* stands for accepting the request.

**Table 5: I&M datasets responses of each DevKit submodule**

| | I&M0 (Index 0xAFF0) | | I&M1 (Index 0xAFF1) | | I&M2 (Index 0xAFF2) | | I&M3 (Index 0xAFF3) | | I&M4 (Index 0xAFF4) | |
|---|---|---|---|---|---|---|---|---|---|---|
| | write | read | write | read | write | read | write | read | write | read |
| DAP | X | OK | OK | OK | OK | OK | OK | OK | OK | OK |
| Interface | X | OK | X | OK | X | OK | X | OK | X | OK |
| Port 1 | X | OK | X | OK | X | OK | X | OK | X | OK |
| Port 2 | X | OK | X | OK | X | OK | X | OK | X | OK |
| Input | X | OK | X | X | X | X | X | X | X | X |
| Output | X | OK | X | X | X | X | X | X | X | X |

Besides the trace capture of the communication, automation of this step requires handling of triggering write and read requests for different indexes and for different destination submodules.

## 5.5 IO Cycle and DHT Timer Verification

The last step for verification of the functionality of the DevKit firmware is verification of IO cycle time and DHT timer verification.

As it was described in chapter 2.2.5, the IO cycle time is a time interval between the data frames sent by the provider. For verification of correct functionality we need to find the difference between the time stamps of two neighboring data frames sent by the DevKit. As the time measurement can be caused by some errors, tolerance of 10% is accepted. The verification of IO cycle times will be done on a number of randomly selected frames from the captured trace.

DHT timer verification covers several functions. First of all, the test verifies the timer itself, then capability of producing alarms (RTA communication) and managing the abortion of AR according to the PROFINET standard.

As the alarm processing takes more time then data providing processing, there could be some internal errors in precise timing of sending the alarm and aborting the AR. Therefore the tolerance in 3 to 6 data frames sent by the DevKit since the last IO controller's data frame is accepted.

Besides the trace capture of the communication, automation of this step requires breaking of the communication between the DevKit and the IO controller while they provide data exchange.

## 5.6 Regression Test Sequence

The complete regression sequence can be represented as a flow chart in Figure 9.

For switching the power supply on the DevKit, for selecting the pin configuration on the connector X40 according to the flash type and for breaking the communication I use output signals from the output module connected directly to the IO controller. The output channels on the module are set by use of OPC server on the PC.

Triggering the read and write requests required for the testing is done by managing the memory bits in the IO controller internal memory.

The process blocks' description starts from abbreviations, which specify the automated application that should process the step:

- [OPC] – OPC server commands
- [TSH] – TShark software application for capturing specified frame types
- [CON] – terminal console communication via serial port
- [DCP] – PyTeMat integrated DCP protocol handling module (*DCP.py*)

Applications for capturing the communication on the network and managing the communication via serial port with the DevKit were not implemented in the original version of PyTeMat. Therefore I implemented python modules managing the applications.

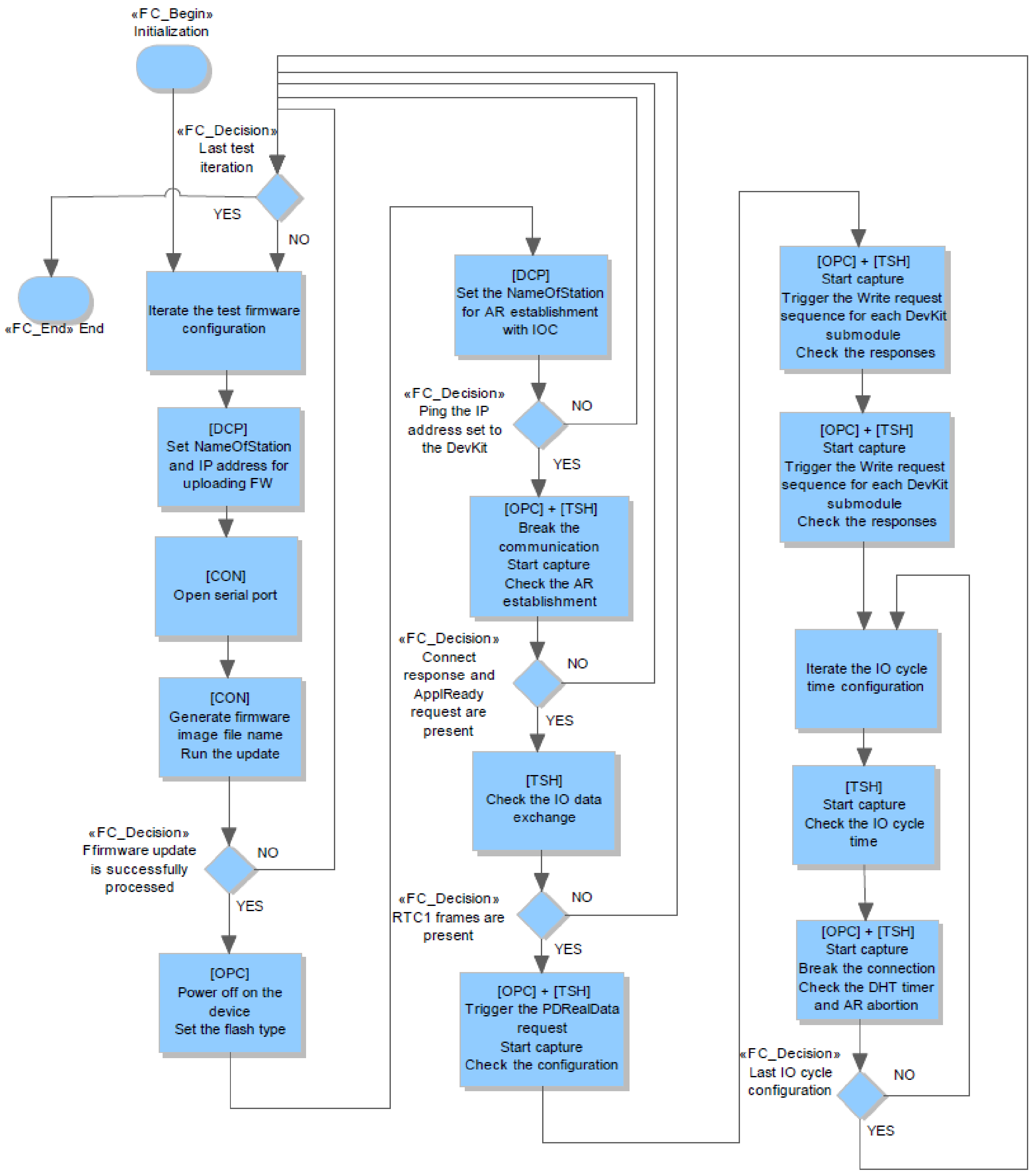

«FC_Begin» Initialization
«FC_Decision» Last test iteration
YES
NO
«FC_End» End
Iterate the test firmware configuration
[DCP] Set NameOfStation and IP address for uploading FW
[CON] Open serial port
[CON] Generate firmware image file name Run the update
«FC_Decision» Ffirmware update is successfully processed
NO
YES
[OPC] Power off on the device Set the flash type
[DCP] Set the NameOfStation for AR establishment with IOC
«FC_Decision» Ping the IP address set to the DevKit
NO
YES
[OPC] + [TSH] Break the communication Start capture Check the AR establishment
«FC_Decision» Connect response and ApplReady request are present
NO
YES
[TSH] Check the IO data exchange
«FC_Decision» RTC1 frames are present
NO
YES
[OPC] + [TSH] Trigger the PDRealData request Start capture Check the configuration
[OPC] + [TSH] Start capture Trigger the Write request sequence for each DevKit submodule Check the responses
[OPC] + [TSH] Start capture Trigger the Write request sequence for each DevKit submodule Check the responses
Iterate the IO cycle time configuration
[TSH] Start capture Check the IO cycle time
[OPC] + [TSH] Start capture Break the connection Check the DHT timer and AR abortion
«FC_Decision» Last IO cycle configuration
NO
YES

*Figure 9: Flow chart diagram for automated regression tests on DevKit*

# 6 Expanding the PyTeMat Core

For realizing the designed test automation I had to expand the core of the PyTeMat with two Python modules. One of them managed the communication via serial port and the firmware update for the DevKit (called *DevKit.py*), another managed the capturing of the communication of the network (called *pywireshark.py*).

## 6.1 DevKit.py Module

The functionality of *DevKit.py* was designed to fulfill missing applications in the PyTeMat in order to make the designed test of the DevKit real. The structure of the module content including classes and their methods is shown in Table 6.

### 6.1.1 NameGenerator Class

As there is a great amount of types of the firmware and the automation requires some adaptation of the substitution of the firmware images on a single DevKit, I decided to resolve the issue by configuring the DevKits with all the firmware and test types in the project for IO controller with unique parameters of *NameOfStation* for each of them (for the configuration see chapter 7.2). Each DevKit configured in the project has a unique IP address too. For handling this configuration I included the class *NameGenerator* to the *DevKit.py* module.

The class handles generation of names for NameOfStation parameter according to the firmware image being tested, NameOfStation according to the IO cycle being tested, IP address according to the firmware and IO cycle, and the name of the firmware image that should be uploaded to the DevKit.

As the result the *NameGenerator* class has four methods for each type of name generation.

The first one is *NameOfStation* method, which input parameters are: *dap* – number of DAP simulated according to the DevKit GSD file (string or integer); *app* – number of the application example tested (string or integer). The method returns the NameOfStation in string format used in the configuration in the project for the IO controller. The form of the name is: “*dap<num>-ap<num>-250us”.*

**Table 6: DevKit.py module content**

| **Class** | **Method** | **Description** |
|---|---|---|
| NameGenerator | NameOfStation | Generates default NameOfStation in format: *dap<num>-ap<num>-250us* |
| | NameOfStationForIo | Generates NameOfStation with the value of IOCycle at the end: *dap<num>-ap<num>-<num><"us" or "ms">* |
| | EcosFileName | Generates file name for ecos binary containing the firmware for update: *ecos_dap<num>_ap<num>_<flashbitnum>_<flashBuild>* |
| | IpAddress | Generates IP address according to DAP, Application example and IO cycle setting |
| Terminal | OpenPort | Checks the presence of the portName attribute and according to it either calls opening of the port or search for available one. |
| | GetPortName | Searches for available DK-ERTEC ports and returns its name |
| | Enter | Enters commands to the open console of serial communication with DK-ERTEC. If the command is 'f', stands for firmware update, checks if the device is prepared for the update by means of CatchOutput method and sets readyForFwUpdate attribute |
| | ReadOutput | Reading the output from the console and writes it down to return attribute |
| | CatchOutput | Checks the console output for a string required and returns Boolean value |
| | EnablePrint | Enters 'P' command to enable console output print and calls CatchOutput for 'enable serial console output' |
| | FlushComPort | Flushes the buffers of the terminal |
| | Close | Closes the serial port |
| FirwareUpdate | Run | Runs the firmware update for a specified firmware image |
| | Reset | Resets the attributes that handles the update process and terminates the TcpFwLoader application |

The second method handles the changing of the name according to the IO cycle being tested, and the method is *NameOfStationForIo*. The input parameters to the method are: *deviceName* – the default NameOfStation generated by NameOfStation method (string); *ioCycle* – the IO cycle in the form of *"<number><us or ms>"* (string).

The third method handles the IP address generation, and the method is called *IpAddress*. The input parameters of the method are: *dap* – number of DAP simulated according to the DevKit GSD file (string or integer); *app* – number of the application example tested (string or integer); *io_cycle* – the IO cycle in the form of number itself (integer). The method returns IP address in the following form: "*192.168.<dap><app>.<io_cycle>*". As the standard IO cycle values could be 250µs, 500µs and 1, 2, 4, 8, 16, 32 and so on ms, there is no collision of the last numbers in the generated IP address. The 500µs address number is solved by dividing the number by 10, and thus we get 50, which still doesn't collide with the ms values.

As was mentioned before, application example 1 and 2 have no differences from the PROFINET communication point of view, therefore these two options are united in one DevKit project configuration with the name and IP address according to application example 1 value.

The same names as generated by these three methods shall be set in the project for IO controller for the functional test application.

The last method included into the *NameGenerator* class is *EcosFileName*. As the test application expects a great amount of firmware types and their sequential uploading to the DevKit, this method handles the generation of new names for the firmware images that shall be prepared in advance. The input parameters to the method are: *dap* – number of DAP simulated according to the DevKit GSD file (string or integer); *app* – number of the application example tested (string or integer); *flashBit* – the number bits of the target flash according to the compilation configuration (string or integer); *flashBuild* – the type of the firmware build (see chapter 5.1). The method returns the string with the name in the following form: "*ecos_dap<num>_ap<num>_<flashbitnum>_<flashBuild>*".

### 6.1.2 Terminal Class

The Terminal class manages the serial communication between the PC and the DevKit. While initialization of the instance an optional input parameter *portName* can be entered. The attribute shall have string format and contain the name of the port connected to the DevKit. If the parameter is missing the application searches for active serial ports with the device name ‘DK-ERTEC’.

The opening of the port for communication is handled by *OpenPort* method. The method has no input and returns parameters. When the method is called, the *portName* attribute is checked firstly. If the attribute is not defined, it takes over the return value of *GetPortName* method. The *GetPortName* method itself by means of *serial* Python module handles the searching of the connected serial ports and returns the name of the port connected to the DevKit (in case if no port is connected returns *None*). After defining the *portName*, the private method *_openPort* is called. This method assigns to *ComPort* attribute the instance of the connected serial port with the *portName*. The connection and instance creation is done by means of *serial* Python module, *Serial* class. If the connection succeeded, enabling print of the user interface is done by special method *EnablePrint*. The method does nothing more than enters ‘*P*’ command to the open terminal console and checks the appropriate response from the DevKit application.

The result of calling the *OpenPort* method is expected to be the definition of *ComPort* attribute as the instance of the open terminal console.

The entering of the commands is handled by the *Enter* method. The input parameter is the *command* in string format. The command entering is done by means of *ComPort*, which should already be an instance of the terminal console. In case if the command is ‘*f*’, which stands for firmware update via TCP, the checking of the appropriate output on the console is automatically run. And if the output corresponds to the expected one for the successful preparation of DevKit for the update, the attribute *readyForFwUpdate* is set to *True* value. Thus we can check later the state of the DevKit by other applications. There is no return parameter from the *Enter* method.

Reading the console output is handled by *ReadingOutput* method. The optional input parameter for the method is *myTime* of float type with default value of 60.0 . This parameter stands for the timeout interval for reading process.  After the reading process is finished the method writes the currently read output *ComPort.out* attribute and returns the value.

The *ReadingOutput* method is used by *CatchOutput* method, which handles the searching of a required output string. So the input parameters of the *CatchOutput* method re: *output* – the searched string (string); *myTime* – the timeout value (float, default value is 0.5). The method cyclically calls the *ReadingOutput* method until the required output is found or until the timeout interval doesn't expire.

The next method is *FlushComPort*. The method has no input and return parameters and its function is to flush the input and output buffers of the *ComPort* instance.

The last method of the *Terminal* class is the *Close* method.  The method has no input and return parameters and the method handles the buffers flushing (by calling the previous method), termination of the serial communication and resetting the *ComPort* attribute to *None* value.

### 6.1.3 FirmwareUpdate Class

The last class of the *DevKit.py* module is *FirmwareUpdate* class. The class is responsible for managing the complete process of the automatic firmware update according to the settings. As the update process is based on serial communication, the initialization of an *FirmawareUpdate* instance requires an input parameter in the form of terminal console instance.

The class has only two public methods: *Run* and *Reset*; and several private auxiliary methods.

The *Run* method handles the complete update process. The method requires four input parameters for that: *ecosFileName* – the name of the firmware image file (string), *ip* – the IP address of the DevKit (string); *flashType* – the flash type for uploading the firmware (string); *flashBit* – the bits of the flash (string).

After calling, the method starts from checking the correctness of the input parameters. Then the method continues with verification of the files (*TcpFwLoader.exe* and the firmware image file) presence in the directory defined in *LocalSettings.py*. If no error occurs, the application enters the '*f*' command for starting the update process itself. Then the method repeats the update sequence from the chapter 5.1. After getting the appropriate output, the command for executing the *TcpFwLoader.exe* is generated. The execution by Windows command prompt is done by means of *subprocess* Python module and *Popen* class. When the DevKit application asks for the flash type for firmware booting, the method enters the value according to the *flashType* attribute specified. After getting the appropriate response from the DevKit application notifying about the end of the firmware update, the attribute *fwUpdated* is set to *True*, allowing other applications to check the state of the update process. The method running is finished by terminating the *TcpFwLoader* application and resetting the auxiliary attributes to their default values. The *Reset* method handles the function.

## 6.2 pywireshark.py Module

Capturing the communication on the network is one of the crucial functionality required for the realization of the test automation. Therefore the *pywireshark.py* module is the second module added to the PyTeMat core.

The automation of this functionality required great effort, as the complete application involves investigation of the TShark application and preparation of scripts in Lua programming language.

### 6.2.1 TShark Application

According to source [13], "TShark is a network protocol analyzer. It lets you capture packet data from a live network, or read packets from a previously saved capture file, either printing a decoded form of those packets to the standard output or writing the packets to a file." The analyzer is installed together with the open source software tool Wireshark and Winpcap library.

The TShark can be executed by the command prompt and, if no additional options are specified, the application will display a summary line after each received frame on the configured network interface to the output of the prompt.

The TShark analyzer supports a great number of options, but only few of them are useful for my application. The first one is '*-i <network interface>*', which sets the network interface for capturing the communication by its name. For checking the presence of the interface, I used '*-D*' option, which prints the list of accessible network interfaces. The next option is '*-a duration:<time>*' , which sets the duration of capture process in seconds. The following option is '*w <output file>*', which saves the captured frames to an output file specified. The file has ***.pcap** format, which can be later read by Wireshark program for manual investigation. For capturing specified type of frames only, I used '*-f <frame type>*' option, which filters the network communication and prints the frames of specified types only. As for my application I need concrete information in the frame and the default information displayed in the summary line is not enough, I preprocess the frame by extension option '*-X lua_script:<name of the script>*' and print the information in the format I require. The processing is done script programmed in Lua programming language. The detailed description of the scripts' design is in the chapter 6.2.2. The TShark application allows using several Lua scripts at once by repeating the argument with new specified script in the command. As I need only printing the frames specified by the script, I can switch off printing of the summary lines by '*-Q*' option. [13]

### 6.2.2 Lua Scripts for TShark

The extension option with Lua script does nothing more than adds the specified script to TShark's other scripts.

By defining a listener in the scripts I get the frames from the network for later decoding and processing the information from it. In the definition of the listener I add the filter for my specified type of the frames.

The filters for the frames are used in the form that is used in the Wireshark tool, where each object in the decoded frame has an appropriate name, so called field name. This name is used while building the filter rules for displaying the appropriate frames. For example, for displaying the ApplicationReady request only I use the following filter: *(dcerpc.opnum == 4) and (dcerpc.pkt_type == 0) and (pn_io.control_block_properties.appl_ready0 == 0x0).*

As it was mentioned in the chapter 2, the ApplicationReady request is a part of PNIO-CM protocol that is based on RPC protocol. Therefore the first two fields of the filter are related to RPC (*dcerpc*). The protocol defines the operations being processed by operation number (*opnum*) and the packet or frame types (*pkt_type*). As the *ApllicationReady* request relates to Control requests from the RPC protocol the operation number equals 4, which stands for Control operation, and the packet type is 0, which stands for request type. As both fields with the specified values shall be present for specifying RPC Control request, the operand between them is *and*. From this moment I have only Control requests filtered, but there are various types of these requests. Therefore I specify one field more, which relates to the PROFINET IO itself (*pn_io*). The field *ControlBlockProperties* defined by the

PROFINET standard can be used for specifying the ApplicationReady request only and the field value is *control_block_properties.appl_ready0 == 0x0*.

In this way I unambiguously specified the ApplicationReady request frames by the filter condition. Thus the listener definition looks as follows:

```
local filter = "(dcerpc.opnum == 4) and (dcerpc.pkt_type == 0) and
(pn_io.control_block_properties.appl_ready0 == 0x0)"
local tap = Listener.new(nil, filter)
```

At this moment the tap variable contains the frames that meet the filter condition. Then I need to specify the fields from the frame that will be printed. And the *items* list contains the fields:

```
items = {
        Field.new("frame.time_relative"),
        Field.new("ip.dst"),
        Field.new("ip.src"),
        Field.new("pn_io.block_type")
        }
```

The field definition works similar to the filter fields definition, excluding their values. In my case, I defined the following fields:

- Time stamp with the time relative to the first frame captured ("*frame.time_relative*")
- IP destination address of the frame ("*ip.dst*")
- IP source address ("*ip.src*")
- Block type of the PROFINET IO part of the frame ("*pn_io.block_type*")

The printing itself is handled in the framework of the following function:

```
function tap.packet(pinfo,tvb)
        frame = get_frame(items)
        str = string.format("%s | %s -> %s | APPLRDY_REQ%s",
        frame.time, frame.ip_src, frame.ip_dst, frame.mod_diff)
        print(str)
end
```

This function is executed with every coming frame that passes the filter. In the first step the fields specified in the *items* list are withdrawn from the received frame, processed to an array, sorted by their offsets and then parsed to a required format. These operations are done in the framework of *get_frame* function.

The parsing operation is responsible for the final string that is printed. Therefore I describe it in more details.

```
function parseFrame(fields)
        nextField = list_iter(fields)
        local frame = {}
        frame.mod_diff = ""
        while true do
                field = nextField()
                if field == nil then
                        return frame
                elseif field.name == "frame.time_relative" then
                        frame.time = field
                elseif field.name == "ip.dst" then
                        frame.ip_dst = field
                elseif field.name == "ip.src" then
                        frame.ip_src = field
                elseif field.name == "pn_io.block_type" then
                        block_type = "0x" .. num2hex(field.value)
                        if block_type == "0x8104" then
                                frame.mod_diff = " | ModuleDiffBlock"
                        end
                else
                        do end
                end
        end
end
```

The operation is handled by *parseFrame* function with input parameter fields. The *fields* parameter is the list of the withdrawn and processed fields. While iterating the fields I compare the field name with the names from the *items* list and if the names are equal I add to the *frame* list the value of the field. In case with the block type field, I do a further investigation of the value in order to search the ModuleDiffBlock with block type equal 0x8104.

After all the operation the result line should be printed according to the defined format. For example, in case of this exact format, the line could look as follows:

```
3.394011 | 192.168.0.50 -> 192.168.0.1 | APPLRDY_REQ | ModuleDiffBlock
```

, where the first number stands for the time relative to the first frame captured, then the delimiter '|' follows, then the source IP address '->' destination IP address, then another delimiter, then '*APPLRDY_REQ*' token, then delimiter and the '*ModuleDiffBlock*' token notifying about the presence of the block in the frame received. In case if the ModuleDiffBlock is absent, the line will be finished by '*APPLRDY_REQ*' token only.

In the similar way each type of the PROFINET standard frame can be processed and printed by the means of the Lua scripts.

### 6.2.3 Wireshark Class

The *pywireshark.py* module has a single class named *Wireshark*. This class handles the capture process on the specified network interface and uses for that the TShark application with its options.

The class contains four public methods and a number of private auxiliary methods.

The first public method is *StartCapture* method. This method is responsible for generating the command for the command prompt for running TShark application with the specified options. The command generation starts from defining of first two arguments *"tshark"* for executing the TShark analyzer and "-Q" option for switching of printing of the summary frame description. The method itself expects extension of the TShark options with some Lua scripts, therefore the printing of ordinary frame description is turned off by default. The following adding of the arguments to the command is done according to the input parameters specified by the user. These input parameters are:

- *inputPcap* – an optional string parameter with the name of the input ***.pcap** file for processing by TShark application. If this parameter is defined the application adds the reading TShark option with the directory and file specified.
- *interface* – an optional string parameter with the name of the network interface. Before adding the appropriate arguments to the command the method checks the presence of the network interface by *InterfaceExists* method. If the parameter is defined, adding of the reading option of the TShark is skipped.
- *filter* – an optional string parameter with the filter according to the filter rules of the Wireshark. If the parameter is defined the application adds filtering TShark option with the filter itself.
- *duration* – an optional integer parameter with the value in seconds of the capture duration. If the parameter is defined the application adds the argument with the capture duration option.
- *savePath* – an optional string parameter with the directory to the output file, which should be saved after the capture process is completed.
- *saveName* – an optional string parameter with the name of the output ***.pcap** file for saving the output from the capturing process. If this parameter is defined the application adds the writing TShark option with the directory (*savePath*) and file specified.

- *sniffers* – a list of the Lua scripts names (string) added as extension option to the TShark analyzer. If the list is not empty the application iterates every name and adds it to the command according to the correct definition of this option.

After the command generation is done, the application defines the *Capture* instance of *subprocess.Popen* class running the generated command, in other words the *Capture* attribute is an instance of capture operation.

The following method from the *Wireshark* class is the *GetFrames* method. This method handles the reading the output from the capture operation. The output is the printed lines coming from the Lua scripts involved in the operation. After getting a frame as a string the method processes it separating each field inside and from a single string makes a list of the fields. From this moment the frame is represented by a list of fields, where the first item is the relative time, the second – the source address(either MAC or IP), the third – the destination address (either MAC or IP), and the forth – frame type.

The input to the *GetFrames* method is presented as a variable length argument list (**args*), which expects to be another filter for the frames in the PyTeMat application itself. At the moment only source and destination addresses (either MAC or IP) are implemented as it meets the requirement of my test automation. If the argument list is defined the first argument stands for source address and the second argument stands for destination address (either MAC or IP). So the method compares the addresses from the frame and from the input and if they are equal, the processed frame is put in the *frames* list. After finishing the capture process the reading is finished too and the method returns the gotten and filtered frames in the form of *frames* list of lists.

Another public method included in the *Wireshark* class is *InterfaceExists* method. This method is an auxiliary method for verifying the presence of the network interface, which name shall be the input parameter to the method. Within the method the TShark analyzer with option '*-D*' is run by the *Popen* class. The output of the command prompt executed by means of the Popen class is the list of the currently accessible network interfacing. After iterating the list and comparing the input interface name with the names from the list, the method returns either *True* value, in case if the name is present in the list, or *False*, if the name is absent.

The last public method is *StopCapture* method that handles terminates the capturing operation, if it is still running, and resets the *Capture* attribute to *None*.

# 7 Test Implementation

The last step in the automation of the regression test is the implementation itself. All the information that was presented in the previous chapters will be brought together for creating the real application.

## 7.1 Station Configuration

I start the implementation of the test design with the assembling of the real station that meets all the requirements. The final station configuration is shown in Figure 10.

The list of the devices composing the station and their brief description can be found in Table 7.

**Table 7: List of the devices composing the station**

| # | Device | Order no. | Description |
|---|---|---|---|
| 1 | PC | - | A computer with PyTeMat application, OPC server, network interface, 1xEthernet RJ45 port, 2xUSB ports. |
| 2 | S7-1500 CPU 1511-1 PN | 6ES7 511-1AK01-0AB0 | PROFINET IO controller with the appropriate project uploaded to it |
| 3 | S7-1500 Digital Output module | 6ES7 522-1BF00-0AB0 | A digital output module with 8xDC 24V/2A channels plugged to the CPU 1511-1 PN |
| 4 | Ethernet Breaker | - | Homebrood device from Siemens s.r.o. CT in Prague. A device with 8 ethernet switches managed by 24V signals. |
| 5 | SCALANCE X204IRT | 6GK5 204-0BA00-2BA3 | An Ethernet switch that can be configured as PROFINET IO device and supports IRT communication |
| 6 | Profishark 100M | C1AP-100 | A TAP device for capturing the Ethernet communication |
| 7 | Development Kit DK-ERTEC 200 PN IO | 6ES7 195-3BE00-0YA0 | The development kit under the test |
| 8 | Partner station SIMATIC ET200SP | 6ES7 155-6AU00-0CN0 | The partner station for the DevKit – PROFINET IO device consisting of IM155-6PN HF, Digital Output module, Digital Input module and Server module |
| 9 | Relay switch | - | The relay switch |

The complete test process is managed by the PyTeMat application. The OPC server sets the output channels on S7-1500 Digital Output module and some Memory bits that trigger Write and Read requests in the CPU 1511-1.

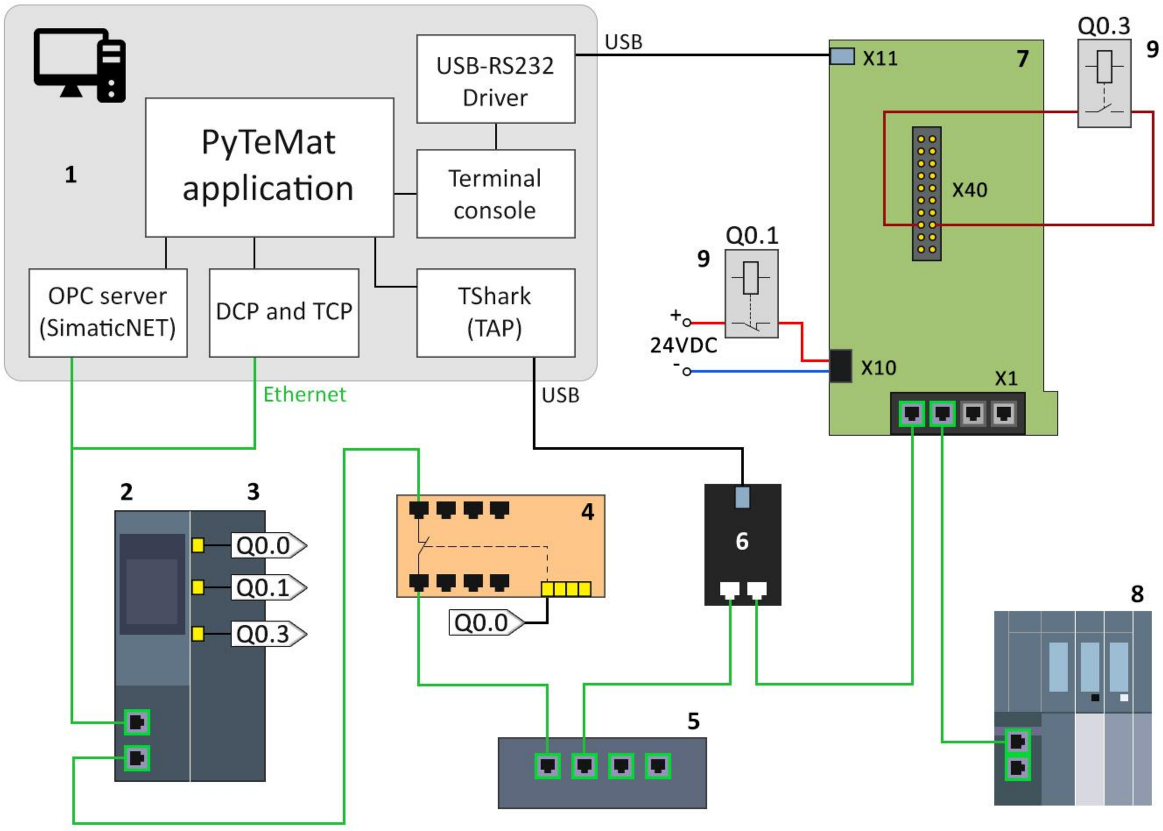


*Figure 10: Test station*[3]

The list of output channels used for controlling the test process is provided in Table 8.

**Table 8: Digital output channels used for test control**

| Channel | | Function |
|---|---|---|
| **No.** | **Tag** | |
| Q0.0 | Q0_EthBreak | The signal breaks the connection between the CPU and the SCALANCE switch by means of the Ethernet Breaker |
| Q0.1 | Q1_PowerSupply | The signal disconnects the DevKit from the power supply by means of a relay switch |
| Q0.3 | Q3_FlashSPI | The signal connects the pins on DevKit connector X40 for SPI flash booting by means of a relay switch |

The SCALANCE switch (device no. 5) located between the Ethernet Breaker and Profishark has the role of IO device, which is not included in the configuration, but

[3] The number on the devices correspond to the numbers in Table 7.

is able to forward all the communication. In case, when the connection is broken by the Ethernet Breaker, the ports on both sides are deactivated. But the SCALANCE switch keeps the port connected to Profishark always active. Therefore the Profishark is able to capture even when the connection with the IO controller is broken. If the Profishark was connected directly to the Ethernet Breaker, it would stop capturing after deactivation of any port. This functionality is significant for DHT test step, when the capture process should continue after the ports disconnection.

## 7.2 TIA Portal Project

For configuring the PROFINET IO station and programming the CPU 1511-1 I used TIA Portal V14. This is new generation multifunctional software for configuring, programming, testing and diagnosis of the controllers. This software was designed by Siemens AG and principally is similar to Step 7.

Firstly so called hardware configuration is required, where are defined all the stations involved in the network communication including OPC server. Each station is represented by IOC module or IOD module with input/output modules or without them.  During the configuration the NameOfStation, IP suite, parameterization are set on each device. The configuration used for the test implementation is shown in Figure 11.

In the hardware configuration I defined CPU 1511-1 with “ioc” NameOfStation. Its IP address is “192.168.0.1” and Subnet mask is “255.255.0.0”. Every device connected to the CPU shall have the same Subnet mask, therefore the Subnet mask of the CPU is the Subnet mask specified for the whole network.

The ET200SP station has “partner” NameOfStation and “192.168.0.2” IP address.

The PC station with the OPC server has “opc_pc” NameOfStation and “192.168.0.3” IP address.

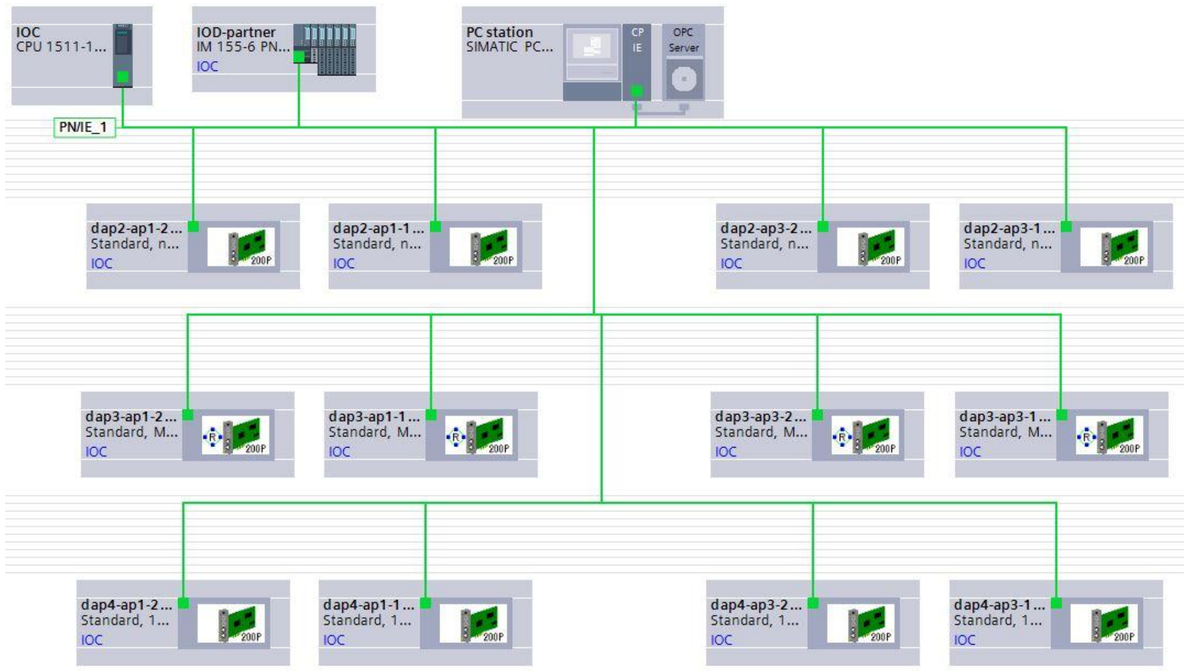


*Figure 11: The hardware configuration (or Devices and networks) in TIA Portal*

Configuring of the DevKit is a little bit more complicated. As it was already mentioned, there is a great amount of various simulated devices that are involved in the test. Therefore I decided to configure the network including each tested option. As the result there are 12 DevKit stations configured covering 27 options chosen from the chapter 5.1 and two different IO cycle SendClock parameter. The tested option is specified by the unique NameOfStation and IP address of each station, for example: NameOfStation is “dap**3**-ap**1**-**1**ms” and the IP address is “192.168.**31**.**1**”. The example configuration stands for DAP**3** device simulation with Application example **1** and the IO cycle set to **1**ms. This feature is closely related to the *NameGenerator* class from the *DevKit.py* module.

From the IO controller point of view all the DevKit stations should be present on the network and, if some of them are missing, it regularly sends DCP Identification request with the appropriate NameOfStation. Nevertheless this condition doesn’t inhibit establishment of an AR with any of the DevKits separately. This feature is used for using a single project for testing various options. As I have only one real DevKit, I can switch over the test configurations by changing its NameOfStation only.

According to the test requirements from chapter 5 we need to be able to send Write and Read requests for specified Record Indexes and the OPC server should be able to manage the sending process. This can be handled by the CPU program and by setting appropriate memory bits. The list of memory bits used for triggering the appropriate Read and Write requests is provided in Table 9.

**Table 9: Digital output channels control**

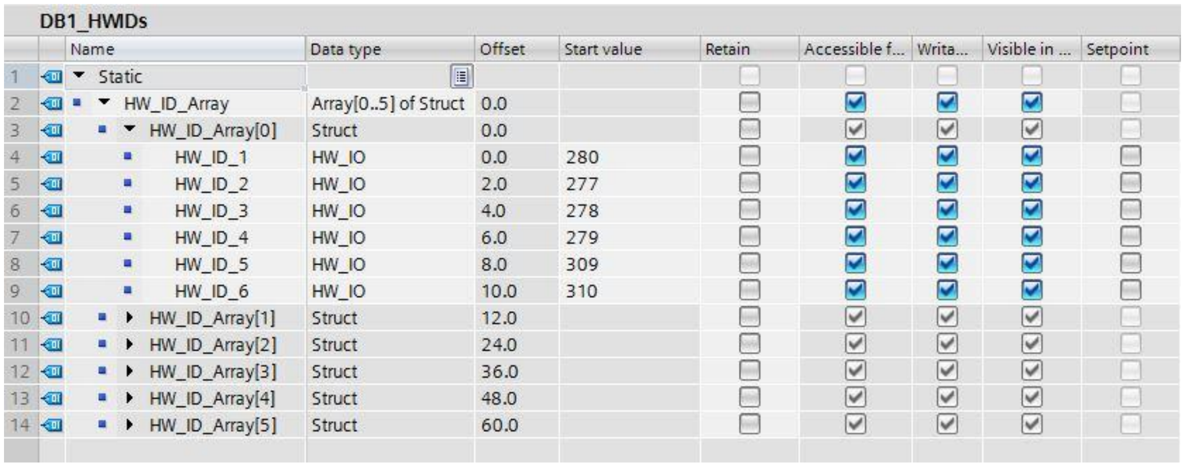
| Memory bit | | | Function |
|---|---|---|---|
| **Addr.** | **Tag** | **Data type** | |
| M0.0 | M0_ReadInterfaceMrpReal | Boolean | Triggers PDInterfaceMrpDataReal Read request |
| M0.1 | M1_WriteIMData | Boolean | Triggers the sequence of I&M0-I&M4 datasets Write requests |
| M0.2 | M2_ReadIMData | Boolean | Triggers the sequence of I&M0-I&M4 datasets Read requests |
| MW1 | MW_HWIDArray | Integer | Set the number of the bundle of station's hardware identified |

I prepared a program for the CPU which handles sending the Read request for PDInterfaceMrpDataReal by triggering the *M0_ReadInterfaceMrpReal* memory bit.

The Write requests and Read requests for I&M0-I&M4 procedure is represented as sequences that are divided into several iterations. The first iteration relates to the submodule address and the second one relates to dataset index. So the application takes the first dataset index and sends the requests to each submodule in the station iterating the submodules' addresses. After finishing processing requests of one index, the application takes the following index and repeats sending requests to all submodule addresses once again, but this time with new index. This procedure is done for all 5 indexes related to I&M0-I&M4 datasets.

The addressing to a specified submodule is done by means of the hardware identifier number that is assigned to each submodule after connecting it to the CPU. The hardware identifiers of the tested stations are written down to a bundle within an array in the database block of the CPU. So the OPC server has an access to the database and is able to set the appropriate identifier bundle according to the station being tested at the moment. This setting is done by writing the integer value of memory word "*MW_HWIDArray*" that later refers to the bundle from the database by means of the CPU program. The array with the identifiers should be prepared before the test run. The array with the first bundle expanded used for the test is shown in Figure 12.

The complete project with the hardware configuration and the CPU program can be found in the TIA Portal project provided on the CD attached to the thesis.



DB1_HWIDs

| | Name | Data type | Offset | Start value | Retain | Accessible f... | Writa... | Visible in ... | Setpoint |
|---|---|---|---|---|---|---|---|---|---|
| 1 | ▼ Static | | | | ☐ | ☐ | ☐ | ☐ | ☐ |
| 2 | ▼ HW_ID_Array | Array[0..5] of Struct | 0.0 | | ☐ | ✓ | ✓ | ✓ | ☐ |
| 3 | ▼ HW_ID_Array[0] | Struct | 0.0 | | ☐ | ✓ | ✓ | ✓ | ☐ |
| 4 | HW_ID_1 | HW_IO | 0.0 | 280 | ☐ | ✓ | ✓ | ✓ | ☐ |
| 5 | HW_ID_2 | HW_IO | 2.0 | 277 | ☐ | ✓ | ✓ | ✓ | ☐ |
| 6 | HW_ID_3 | HW_IO | 4.0 | 278 | ☐ | ✓ | ✓ | ✓ | ☐ |
| 7 | HW_ID_4 | HW_IO | 6.0 | 279 | ☐ | ✓ | ✓ | ✓ | ☐ |
| 8 | HW_ID_5 | HW_IO | 8.0 | 309 | ☐ | ✓ | ✓ | ✓ | ☐ |
| 9 | HW_ID_6 | HW_IO | 10.0 | 310 | ☐ | ✓ | ✓ | ✓ | ☐ |
| 10 | ▶ HW_ID_Array[1] | Struct | 12.0 | | ☐ | ✓ | ✓ | ✓ | ☐ |
| 11 | ▶ HW_ID_Array[2] | Struct | 24.0 | | ☐ | ✓ | ✓ | ✓ | ☐ |
| 12 | ▶ HW_ID_Array[3] | Struct | 36.0 | | ☐ | ✓ | ✓ | ✓ | ☐ |
| 13 | ▶ HW_ID_Array[4] | Struct | 48.0 | | ☐ | ✓ | ✓ | ✓ | ☐ |
| 14 | ▶ HW_ID_Array[5] | Struct | 60.0 | | ☐ | ✓ | ✓ | ✓ | ☐ |

*Figure 12: Database of the hardware idetifiers*

## 7.3 Lua Scripts Preparation

For implementing the designed test I needed to prepare some Lau script sniffers for getting the required frames to the PyTeMat application. Therefore I programmed several sniffers (according to the method described in chapter 6.2.2) and each of them handles a specific type of frames providing the required information from them. The list of the sniffer scripts is provided in Table 10.

**Table 10: TShark sniffer scripts used in the PyTeMat application**

| **Script name** | **Frame type** | **The output line format** |
|---|---|---|
| appl_req_sniff.lua | ApplicationReady - Control request | <time relative> \| <source IP> -> <destination IP> \| APPLRDY_REQ \| ModuleDiffBlock* |
| conn_res_sniff.lua | Connect request | <time relative> \| <source IP> -> <destination IP> \| CONN_RES \| Status:<OK or ERROR> |
| interfacemrpreal_sniff.lua | PDInterfaceMrpDataReal – Read request | <time relative> \| <source IP> -> <destination IP> \| READ_RES_PDINTERFACEMRPDATAREAL \| Status:<OK or ERROR> |
| imdata_sniff.lua | I&M data – Write and Read requests | <time relative> \| <source IP> -> <destination IP> \| <READ or WRITE>_RES \| Status:<OK or ERROR> \| SlotNumber:<hex number of the slot> \| SubslotNumber:< hex number of the subslot> \| Index:<hex index number of the reacord data> |
| rtc1_sniff.lua | IO cyclic data – RTC1 | <time relative> \| <source MAC> -> <destination MAC> \| RTC1 \| FrameID: <hex number of the frame ID> \| DataStatus: <OK or ERROR> \| TransferStatus: <OK or ERROR> |
| err_dht_sniff.lua | Error - RTA | <time relative> \| <source MAC> -> <destination MAC> \| ERR-RTA \| Status: Error: "RTA error", "PNIO", "RTA_ERR_CLS_PROTOCOL", "AR consumer DHT/WDT expired (RTA_ERR_ABORT)" |
| * The "*ModuleDiffBlock*" element is present only under certain conditions specified within the Lua script. | | |

Using this Lua scripts as the extension option of TShark application I am able to provide enough information from the received frames to PyTeMat application in order to cover all steps of the test designed.

The Lua scripts can be found on the CD attached to the thesis in the directory *\PyTeMat\tshark_sniff\sniffers\*.

## 7.4 Test Application in PyTeMat

The regression test application is implemented in the PyTeMat test tool. I followed the rules of creating the test application in PyTeMat described in chapter 4.2.

### 7.4.1 PyTeMat Hardware Configuration

In the *DevKit_Automat_rack.py* file I defined the hardware configuration used by PyTeMat. The configuration is composed by project definition and specification of three device instances. The instances are represented by the real devices in the real test station and therefore the instances have defined MAC address, IP suite and NameOfStation arguments taken from their representatives.

The first instance is *Controller* instance of *PN_Devices.Simatic* class and is represented by the CPU 1511-1. The CPU 1511-1 is connected to the OPC server and the parameters of the connection are specified for the instance too. Thus the definition of the *Controller* instance looks as follows:

```
Controller = PN_Devices.Simatic()
Controller.Name                     = "SIMATIC_CPU"
Controller.DeviceNumber             = 1
Controller.IPSuite.MAC              = "28:63:36:aa:0e:5f"
Controller.IPSuite.IpAddress        = "192.168.0.1"
Controller.IPSuite.SubNetMask       = "255.255.0.0"
Controller.IPSuite.DeviceName       = "ioc"
Controller.OPC                      = OPCs.SimaticOPC()
Controller.OPC.OPCServerName        = "OPC.SimaticNET"
Controller.OPC.OPCTopicName         = "S71500ET200MP station_1.IOC"
```

The first instance is *Partner* instance of *PN_Devices.ET200SP* class and is represented by the ET200SP station. The definition of the instance looks as follows:

```
Partner = PN_Devices.ET200SP()
Partner.Name                        = "partner"
Partner.DeviceNumber                = 2
Partner.IPSuite.MAC                 = "28:63:36:3b:20:90"
Partner.IPSuite.IpAddress           = "192.168.0.2"
Partner.IPSuite.SubNetMask          = "255.255.0.0"
Partner.IPSuite.GateWay             = "192.168.0.2"
Partner.IPSuite.DeviceName          = "partner"
```

The last instance on the hardware configuration file is *DeviceDK* instance of *PN_Devices.DK_ERTEC* class and is represented by the DevKit. The *DeviceDK* instance includes instances of the classes from the *DevKit.py* module for the regression test performance. Thus the definition of the instance looks as follows:

```
DeviceDK = PN_Devices.DK_ERTEC()
DeviceDK.GenerateName               = DevKit.NameGenerator()
DeviceDK.Console                    = DevKit.Terminal()
DeviceDK.FWUpdate                   = DevKit.FirwareUpdate(DeviceDK.Console)
DeviceDK.Name                       = "dk-ertec"
DeviceDK.IPAddrForFWUpload          = LocalSettings.IPAddForFWUpload
DeviceDK.DeviceNumber               = 3
DeviceDK.IPSuite.MAC                = "08:00:06:02:01:10"
DeviceDK.IPSuite.DeviceName         = "dk-ertec"
DeviceDK.IPSuite.IpAddress          = LocalSettings.IPAddForFWUpload
DeviceDK.IPSuite.SubNetMask         = "255.255.0.0"
DeviceDK.IPSuite.GateWay            = "0.0.0.0"
```

The file can be found on the CD attached to the thesis in the directory *\PyTeMat\Tests\DP_DEVKIT_TESTS\HWconfigs\*.

### 7.4.2 PyTeMat Source File

The regression test sequence designed in chapter 5.6 is implemented in the *TC_DevKit_Automat.ptm* source file according to the TBL syntax.

The *INITIALIZATION* section starts from the definition of *tests* list of dictionaries. The list contains all firmware option combinations that should be tested and each dictionary represents one option specified. The examples of three options follow:

```
tests = [
        {"enum": 0, "dap": 2, "app": 1, "flashType": "NOR", "flashBit": 32,  "flashBuild":
"posix", "hw_id": 0, "2Ports": True, "MRP": False, "DHT": True, "DataSets": True},
       {"enum": 10, "dap": 3, "app": 1, "flashType": "NOR", "flashBit": 32, "flashBuild":
"native", "hw_id": 1, "2Ports": True, "MRP": True, "DHT": True, "DataSets": True},
       {"enum": 20, "dap": 4, "app": 1, "flashType": "SPI", "flashBit": 32, "flashBuild":
"posix", "hw_id": 2, "2Ports": False, "MRP": False, "DHT": True, "DataSets": True}
       ]
```

The keys of the dictionary represent parameters of the firmware option and some additional parameters for test performance on the specified firmware option. The keys are described in Table 11.

**Table 11: The dictionary of test option**

| **Key** | **Values** | **Description** |
|---|---|---|
| “enum” | integer | Enumerator of the test option |
| “dap” | integer | DAP value of the simulated device according to the DevKit GSD file |
| “app” | integer | Application example value |
| “flashType” | “NOR” of “SPI” | Type of the flash, which the firmware will be uploaded to |
| “flashBit” | integer (16 or 32) | The bit value of the NOR flash (SPI has value 32 by default) |
| “flashBuild” | “posix” or “native” | The operating system build of the firmware |
| “hw_id” | integer (0..5) | Hardware indetifier bundle number according to the TIA Portal project |
| “2Ports” | Boolean | True if the simulated device has 2 ports; False if the device has 1 port |
| “MRP” | Boolean | The value of the MRP functionality support according to the DAP |
| “DHT” | Boolean | The test setting of performance of the IO cycle SendClock and DHT timer test steps |
| “DataSets” | Boolean | The test setting of performance the I&M datasets test steps |

In the similar way the *dataSets* list of responses to I&M datasets request is defined for each submodule being tested. Logically the content of the list repeats the content of the Table 5, where *X* is replaced by *False* value and *OK* is replaced by *True* value.

The IO cycles tested in the framework of the test case are represented by *ioCycles* dictionary. The key of the dictionary is the string of the IO cycle value with the unit of time and the value of each key is the float value in milliseconds. In case of my test case the *ioCycles* dictionary includes two IO cycle values for the tests, which looks as follows:

```
ioCycles = {"250us": 0.25, "1ms": 1.0}
```

In the *INITIALIZATION* section I create an instance of pywireshark.Wireshark class, which is called *Wireshark* and run the connection to the OPC server by calling *Controller.OPC.ProjectOnline* method. Some auxiliary methods are defined as well. For example, *DevicePowerOFF* method writes *True* value to the *Q1_PowerSupply* output CPU address via OPC server, causing the power supply turning off on the DevKit.

The *TESTCASE* sections starts from iterating the test option taken from the *tests* list. So the whole test case performance is repeated as many times as the length value of the *tests* list initialized and with the parameters got from the current test option.

The *TESTCASE* section is divided into *STEP* sections with the appropriate *CRITERION* sections. The test case sequence implemented in the PyTeMat is fully consistent to the designed test sequence described in chapter 5. All the operations involved in the test process are implemented by means of the original modules from the PyTeMat (*DCP.py*, *OPC.py*), by means of the modules designed and programmed by me (*DevKit.py*, *pywireshark.py*). The complete code from final version of the *TC_DevKit_Automat.ptm* source file can be found in the **Annex B** or on the CD in *\PyTeMat\Tests\DP_DEVKIT_TESTS\TC\src\* directory.

### 7.4.3 PyTeMat Run File

The *run* file designed for execution of the test case by the PyTeMat tool (*DevKit_Automat.ptc*) corresponds to the structure described in chapter 4.2 and its content looks as follows:

```
HWconfig Tests\DP_DEVKIT_TESTS\HWconfigs\DevKit_Automat_rack.py

PrefixTCS Tests\DP_DEVKIT_TESTS\TC\src\
PrefixPRM Tests\DP_DEVKIT_TESTS\TC\prm\

Testcase TC_DevKit_Automat.ptm PRM PRM_DevKit_Automat.ptm
```

The *DevKit_Automat.ptc* file itself can be found on the CD in *\PyTeMat\Tests\DP_DEVKIT_TESTS\TC\run\* directory.

As the presence of *parameter* file is a mandatory condition for the execution it is created (*PRM_DevKit_Automat.ptm*) in the appropriate folder and is mentioned in the *run* file. But actually the content of the *parameter* file includes the definition of idle method without any function.

After implementing the test sequence by the PyTeMat tool the test scenario is ready for the performance.

## 7.5 Test Run

Before starting the test performance by executing the PyTeMat *run* file, I needed to perform several actions related to the test preparation, besides the assembling of the real test station, creating TIA Portal project with the functional program and implementing the test sequence by the PyTeMat tool.

The first action is the preparation of the firmware images used in the test sequence. I compiled 18 firmware images for the test options selected for performance in the framework of the thesis. The image files where named according to the rules defined by *NameGenerator* class from the *DevKit.py* module (for example, *ecos_dap2_ap1_32_native.bin*, *ecos_dap4_ap3_32_posix.bin* and so on).

The second action is the specification of the settings for the PyTeMat test tool. The settings are defined in the *LocalSetting.py* file. In case of the implemented test the file has the following settings:

```
import LocalSettings

# version of PyTeMat, that will be run
LocalSettings.PyTeMatPath            = "c:\\PyTeMat\\"
LocalSettings.IntermediateFilesPath = "c:\\PyTeMat\\Test_System\\"

# network card selection - will be used for DCP services
LocalSettings.NetworkConnectionName = "Siemens"
LocalSettings.EthInterfaceIP        = "192.168.0.3"

# settings for wireshark.py module
LocalSettings.WiresharkInterface    = "Profishark"
LocalSettings.LuaPath               = "c:\\PyTeMat\\tshark_sniff\\sniffers\\"
LocalSettings.PcapFilePath          = "c:\\PyTeMat\\Test_System\\Log_Files\\Captures"

# other optional local settings
LocalSettings.TcpLoaderPath         = "c:\\PyTeMat\\BootleBinaryTest\\"
LocalSettings.IPAddForFWUpload      = "192.168.0.123"

# default loggers settings
LocalSettings.LogFilePath           = "c:\\PyTeMat\\Test_System\\Log_Files"
LocalSettings.SumFilePath           = "c:\\PyTeMat\\Test_System\\Test_Results"
```

With these settings the test sequence was executed by opening the command prompt with the directory of the *run* file and entering the following command:

```
pytemat DevKit_Automat.ptc -f
```

From this moment the test performance starts in the Force PyTeMat mode and the summary information about the process is printed to the command prompt output.

The first test run was interrupted while performance of test option no. 18 (iterated test from *tests* list with key *"enum": 18*, for details see Annex A). After I got the DevKit under the test to its default state, I run the second test performance starting from the test option no. 18. At this time the test was performed to its end without interruption.

As the result the test run, I received two log files in ***.txt** format containing the data related to the test performance procedure and two folders with the trace log files with the communication of the network while the test performance. This data should be enough for investigation of the test step failures and other problems occurred while the test performance.

# 8 Result Analysis

Each test procedure ends with the analysis of its results.

The summary at the of output log ***.txt** file received from the interrupted test run has the following content:

```
19.12.2016 20:14:39 <Exception ; Preprocessor>: ===================== END TRACEBACK
=====================
19.12.2016 20:14:39 line 588 <Finally ; Preprocessor>: FINALLY - Clean up the test environment
19.12.2016 20:14:56 <Testcase_evaluation ; PyTeMat>: =============================
19.12.2016 20:14:56 <Testcase_evaluation ; PyTeMat>: Testcase has failed !
19.12.2016 20:14:56 <Testcase_evaluation ; PyTeMat>: with 5 faults
19.12.2016 20:14:56 <Testcase_evaluation ; PyTeMat>: === Fault  recapitulation ===
19.12.2016 20:14:39 <Testcase_evaluation ; Preprocessor>: CRITERION has failed
19.12.2016 20:14:39 <Testcase_evaluation ; PyTeMat>: ========= Exception OCCURED - PyTeMat
stopped =========
19.12.2016 20:14:39 <Testcase_evaluation ; Preprocessor>: ======================= TRACEBACK
=======================
19.12.2016 20:14:39 <Testcase_evaluation ; Preprocessor>: Traceback (most recent call last):
  File "C:/PyTeMat/Test_System/1-executable_code-TC_DevKit_Automat.py", line 628, in <module>
    flashType=test["flashType"], flashBit=test["flashBit"])
  File "c:\PyTeMat\\Core\PyModules\DevKit.py", line 283, in Run
    if self._parent.ComPort.is_open:
AttributeError: 'NoneType' object has no attribute 'is_open'

19.12.2016 20:14:39 <Testcase_evaluation ; Preprocessor>: ===================== END TRACEBACK
=====================
19.12.2016 20:14:56 <Testcase_evaluation ; PyTeMat>: =============================
```

This test run has one criteria failure that causes the test run interruption.

After investigation of the error and the DevKit serial communication the defect was found. After uploading firmware image *ecos_dap4_ap3_32_native.bin* (test case no. 17) to the NOR 32 bit flash the DevKit stops communicating via serial port. Therefore the uploading of the following firmware is not possible via TCP as the terminal console application doesn’t work.

The second summary file of the second run has the following content:

```
19.12.2016 21:33:28 <Testcase_evaluation ; PyTeMat>: =============================
19.12.2016 21:33:28 <Testcase_evaluation ; PyTeMat>: Testcase has failed !
19.12.2016 21:33:28 <Testcase_evaluation ; PyTeMat>: with 7 faults
19.12.2016 21:33:28 <Testcase_evaluation ; PyTeMat>: === Fault  recapitulation ===
19.12.2016 20:51:08 <Testcase_evaluation ; Preprocessor>: CRITERION has failed
19.12.2016 20:53:38 <Testcase_evaluation ; DevKit>: The TcpFWLoader is not executed or the
update is not possible.
19.12.2016 20:56:38 <Testcase_evaluation ; DevKit>: The firmware update process FAILED or
exceeded 180 second period.
19.12.2016 20:56:38 <Testcase_evaluation ; Preprocessor>: CRITERION has failed
19.12.2016 21:06:15 <Testcase_evaluation ; Preprocessor>: CRITERION has failed
19.12.2016 21:10:03 <Testcase_evaluation ; Preprocessor>: CRITERION has failed
19.12.2016 21:33:00 <Testcase_evaluation ; Preprocessor>: CRITERION has failed
19.12.2016 21:33:28 <Testcase_evaluation ; PyTeMat>: =============================
```

The second test run has four criteria failures and according to the time of their occurrence I am able to investigate the cause of the failure.

The first failure occurred at 20:56:38 was caused by DevKit's falling into the defect state, when the device is not able to communicate at all. This happened when *ecos_dap2_ap1_32_posix.bin* firmware image was uploaded to the SPI flash of the DevKit.

The second and the third failures occurred at 21:06:15 and 21:10:03 respectively relate to the AR establishment. In this case the DevKit was not able to establish the AR with the IO controller after uploading the firmware images *ecos_dap3_ap2_32_posix.bin* and *ecos_dap3_ap2_32_posix.bin*.

The last failure occurred at 21:33:00 was caused by wrong IO cycle time interval, which values were out of the acceptable range.

The received results approve the functionality of the test design and implementation according to the requirements of regression test. The defects found during the test performance were reported to the software developers.

The results of the tests are added to the CD and can be found in *\PyTeMat\Test_System\Log_Files\* directory.

# 9 Conclusions

In concordance with the thesis assignment, all the tasks were successfully accomplished.

I investigated the basis of PROFINET communication standard and briefly described main functionalities supported by the standard. Then I familiarized with the Siemens PROFINET IO Development Kit device and its functionality based on the PROFINET standard and serial communication operations. The main operation related to the test automation was the firmware update via TCP. After I investigated the PyTeMat test tool core modules, the structure of the test design by the tool and the test execution procedure. Based on the knowledge of the PROFINET standard, the DevKit functionality and PyTeMat application, I designed a test case for automation of regression tests. The test case covers the main part of the regression tests, which should be performed on new releases of the DevKit. For the implementation of the designed test case, I had to expand the PyTeMat core with two modules. One of them managed the serial communication with the DevKit and automatic firmware update; another one realized the automatic capturing of the communication on the network. The automation of capturing was based on the Lua scripts, which had to be created in advance – one for each type of the analyzed frame. Then I created the TIA Portal project including the hardware configuration and the program, which manages sending appropriate requests to the DevKit while the test performance. After I assembled the test station and implemented the test sequence in the PyTeMat, I successfully performed the test run. I selected 27 test options with 18 different firmware options for the test and the results of the test performance were generated by PyTeMat tool in text format and a number of files with the frames captured on the network.

The result analysis showed that the tested version of the DevKit release has potential defects. Therefore I reported about them to the software developers and they fixed the defects in the final version of the DevKit before its release to the market.

The automation of the tests was a highly reasonable task, due to several facts. The first one is that the entire test performance lasted for about 2 hours and 20 minutes, when the manual testing of the same amount of the options could last several working days. The second fact is that this test design and implementation is being currently used in the real application while development of the future releases of the DevKit software. For the real application, it is planned to expand the scope of the test cases. According to plan, the future version of the test case will cover testing of the full set of the firmware options and some new test steps will be added (for example testing of the user interface), and the current solution for the developed automatic test application is designed in a way that these new features can be easily integrated to the application.

# References

[1] *Programmable logic controller* [online]. 2016 [vid. 2016-12-20]. Available: https://en.wikipedia.org/w/index.php?title=Programmable_logic_controller&oldid=755575480

[2] *Fieldbus* [online]. 2016 [vid. 2016-12-20]. Available: https://en.wikipedia.org/w/index.php?title=Fieldbus&oldid=750986345

[3] *PROFINET* [online]. 2016 [vid. 2016-12-20]. Available: https://en.wikipedia.org/w/index.php?title=PROFINET&oldid=754378057

[4] *PROFINET System Description, Technology and Application*. B.m.: PROFIBUS Nutzerorganisation e. V. (PNO), PROFIBUS & PROFINET International (PI). October 2014

[5] Profinet, Industrial Ethernet for advanced manufacturing. *PI North America (PTO)* [online]. [vid. 2016-12-20]. Available: http://us.profinet.com/technology/profinet/

[6] *Application Layer services for decentralized periphery, Technical Specification for PROFINET IO*. B.m.: PROFIBUS Nutzerorganisation e.V. March 2016

[7] *GSDML, Technical Specification for PROFINET*. B.m.: PROFIBUS Nutzerorganisation e.V. April 2016

[8] *Application Layer protocol for decentralized periphery, Technical Specification for PROFINET IO*. B.m.: PROFIBUS Nutzerorganisation e.V. March 2016

[9] *PROFINET Test case specification, Basic: V2.32, Title: DataHoldTimer*. B.m.: PROFIBUS Nutzerorganisation e.V. May 2015

[10] *Guidelines for Evaluation Kit, ERTEC 200P-2 V4.4.0*. November 2016

[11] *Evaluation Board ERTEC 200P-2, Manual*. B.m.: Siemens AG

[12] *Media Redundancy Protocol* [online]. 2016 [vid. 2016-12-25]. Available: https://en.wikipedia.org/w/index.php?title=Media_Redundancy_Protocol&oldid=748796764

[13] *tshark\ -\ The\ Wireshark\ Network\ Analyzer\ 2.0.0* [online]. [vid. 2017-01-01]. Available: https://www.wireshark.org/docs/man-pages/tshark.html

# Annex A

## Startup sequence of the IOD

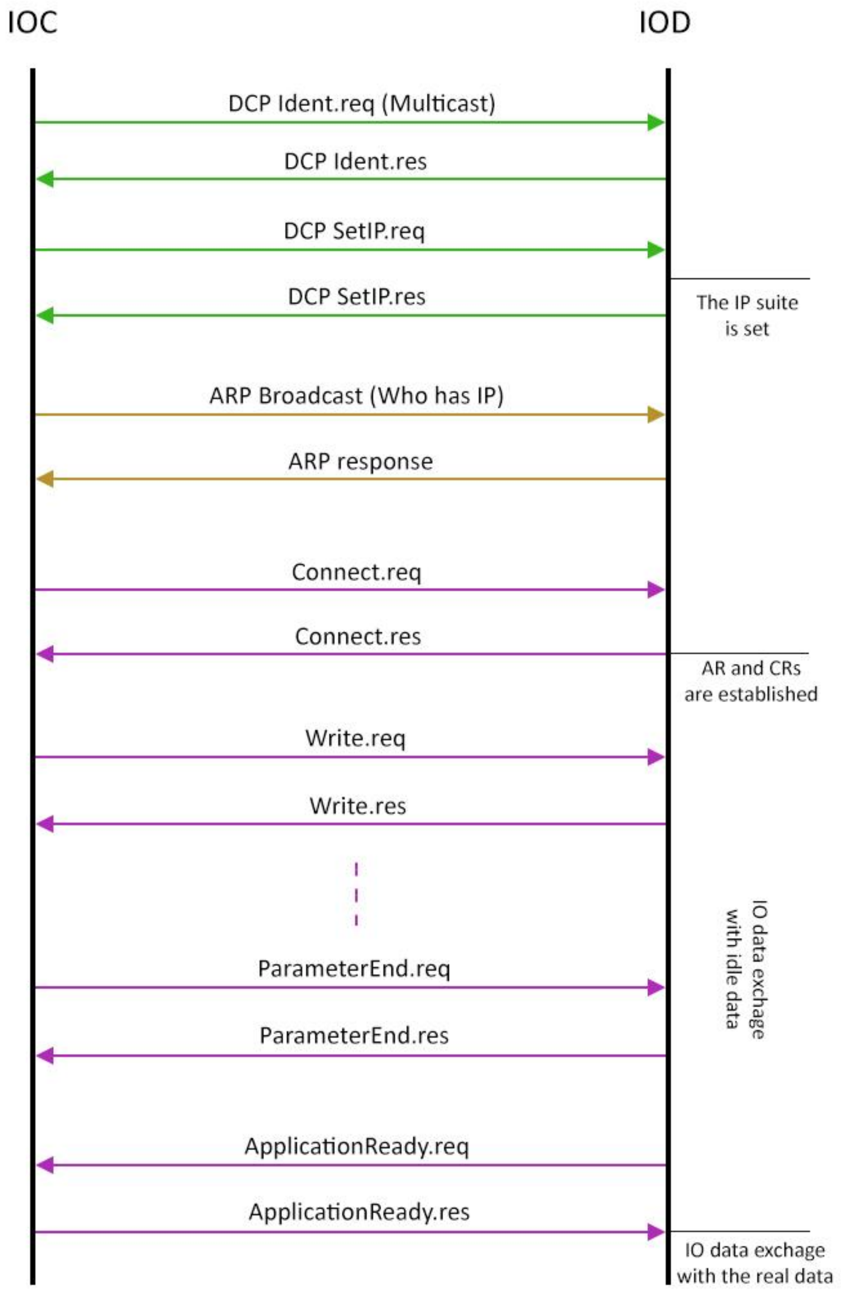


*Figure A.1: The schematic representation of the startup sequence*

# Annex B

## The Content of TC_DevKit_Automat.ptm Source File

```
INITIALIZATION:
    tests = [
             {"enum": 0, "dap": 2, "app": 1, "flashType": "NOR", "flashBit": 32,
"flashBuild": "posix", "hw_id": 0,
              "2Ports": True, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 1, "dap": 3, "app": 1, "flashType": "NOR", "flashBit": 32,
"flashBuild": "posix", "hw_id": 1,
              "2Ports": True, "MRP": True, "DHT": True, "DataSets": True},
             {"enum": 2, "dap": 4, "app": 1, "flashType": "NOR", "flashBit": 32,
"flashBuild": "posix", "hw_id": 2,
              "2Ports": False, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 3, "dap": 2, "app": 2, "flashType": "NOR", "flashBit": 32,
"flashBuild": "posix", "hw_id": 0,
              "2Ports": True, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 4, "dap": 3, "app": 2, "flashType": "NOR", "flashBit": 32,
"flashBuild": "posix", "hw_id": 1,
              "2Ports": True, "MRP": True, "DHT": True, "DataSets": True},
             {"enum": 5, "dap": 4, "app": 2, "flashType": "NOR", "flashBit": 32,
"flashBuild": "posix", "hw_id": 2,
              "2Ports": False, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 6, "dap": 2, "app": 3, "flashType": "NOR", "flashBit": 32,
"flashBuild": "posix", "hw_id": 3,
              "2Ports": True, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 7, "dap": 3, "app": 3, "flashType": "NOR", "flashBit": 32,
"flashBuild": "posix", "hw_id": 4,
              "2Ports": True, "MRP": True, "DHT": True, "DataSets": True},
             {"enum": 8, "dap": 4, "app": 3, "flashType": "NOR", "flashBit": 32,
"flashBuild": "posix", "hw_id": 5,
              "2Ports": False, "MRP": False, "DHT": True, "DataSets": True},


             {"enum": 9, "dap": 2, "app": 1, "flashType": "NOR", "flashBit": 32,
"flashBuild": "native", "hw_id": 0,
              "2Ports": True, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 10, "dap": 3, "app": 1, "flashType": "NOR", "flashBit": 32,
"flashBuild": "native", "hw_id": 1,
              "2Ports": True, "MRP": True, "DHT": True, "DataSets": True},
             {"enum": 11, "dap": 4, "app": 1, "flashType": "NOR", "flashBit": 32,
"flashBuild": "native", "hw_id": 2,
              "2Ports": False, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 12, "dap": 2, "app": 2, "flashType": "NOR", "flashBit": 32,
"flashBuild": "native", "hw_id": 0,
              "2Ports": True, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 13, "dap": 3, "app": 2, "flashType": "NOR", "flashBit": 32,
"flashBuild": "native", "hw_id": 1,
              "2Ports": True, "MRP": True, "DHT": True, "DataSets": True},
             {"enum": 14, "dap": 4, "app": 2, "flashType": "NOR", "flashBit": 32,
"flashBuild": "native", "hw_id": 2,
              "2Ports": False, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 15, "dap": 2, "app": 3, "flashType": "NOR", "flashBit": 32,
"flashBuild": "native", "hw_id": 3,
              "2Ports": True, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 16, "dap": 3, "app": 3, "flashType": "NOR", "flashBit": 32,
"flashBuild": "native", "hw_id": 4,
              "2Ports": True, "MRP": True, "DHT": True, "DataSets": True},
             {"enum": 17, "dap": 4, "app": 3, "flashType": "NOR", "flashBit": 32,
"flashBuild": "native", "hw_id": 5,
              "2Ports": False, "MRP": False, "DHT": True, "DataSets": True},


             {"enum": 18, "dap": 2, "app": 1, "flashType": "SPI", "flashBit": 32,
"flashBuild": "posix", "hw_id": 0,
              "2Ports": True, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 19, "dap": 3, "app": 1, "flashType": "SPI", "flashBit": 32,
"flashBuild": "posix", "hw_id": 1,
              "2Ports": True, "MRP": True, "DHT": True, "DataSets": True},
             {"enum": 20, "dap": 4, "app": 1, "flashType": "SPI", "flashBit": 32,
"flashBuild": "posix", "hw_id": 2,
              "2Ports": False, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 21, "dap": 2, "app": 2, "flashType": "SPI", "flashBit": 32,
"flashBuild": "posix", "hw_id": 0,
              "2Ports": True, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 22, "dap": 3, "app": 2, "flashType": "SPI", "flashBit": 32,
"flashBuild": "posix", "hw_id": 1,
              "2Ports": True, "MRP": True, "DHT": True, "DataSets": True},
             {"enum": 23, "dap": 4, "app": 2, "flashType": "SPI", "flashBit": 32,
"flashBuild": "posix", "hw_id": 2,
              "2Ports": False, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 24, "dap": 2, "app": 3, "flashType": "SPI", "flashBit": 32,
"flashBuild": "posix", "hw_id": 3,
              "2Ports": True, "MRP": False, "DHT": True, "DataSets": True},
             {"enum": 25, "dap": 3, "app": 3, "flashType": "SPI", "flashBit": 32,
"flashBuild": "posix", "hw_id": 4,
              "2Ports": True, "MRP": True, "DHT": True, "DataSets": True},
             {"enum": 26, "dap": 4, "app": 3, "flashType": "SPI", "flashBit": 32,
"flashBuild": "posix", "hw_id": 5,
              "2Ports": False, "MRP": False, "DHT": True, "DataSets": True}
             ]

                # DAP       #Interface #Port1      #Port2      #Input      #Output
    hw_sub = [[ True,       True,       True,       True,       True,       True],
              [ True,       True,       True,       True,       True,       True],
              [ True,       True,       True,       False,      True,       True],
              [ True,       True,       True,       True,       True,       True],
              [ True,       True,       True,       True,       True,       True],
              [ True,       True,       True,       False,      True,       True]
              ]

    ioCycles = {"250us": 0.25, "1ms": 1.0}

    dataSets = [[{"slot": "SlotNumber:0x0", "subslot": "SubslotNumber:0x1"}, {"name":
"I&M0", "wr": False, "rd": True},
                 {"name": "I&M1", "wr": True, "rd": True}, {"name": "I&M2", "wr": True,
"rd": True},
                 {"name": "I&M3", "wr": True, "rd": True}, {"name": "I&M4", "wr": True,
"rd": True}],  # DAP
                [{"slot": "SlotNumber:0x0", "subslot": "SubslotNumber:0x8000"}, {"name":
"I&M0", "wr": False, "rd": True},
                 {"name": "I&M1", "wr": False, "rd": True}, {"name": "I&M2", "wr": False,
"rd": True},
                 {"name": "I&M3", "wr": False, "rd": True}, {"name": "I&M4", "wr": False,
"rd": True}],  # Interface
                [{"slot": "SlotNumber:0x0", "subslot": "SubslotNumber:0x8001"}, {"name":
"I&M0", "wr": False, "rd": True},
                 {"name": "I&M1", "wr": False, "rd": True}, {"name": "I&M2", "wr": False,
"rd": True},
                 {"name": "I&M3", "wr": False, "rd": True}, {"name": "I&M4", "wr": False,
"rd": True}],  # Port1
                [{"slot": "SlotNumber:0x0", "subslot": "SubslotNumber:0x8002"}, {"name":
"I&M0", "wr": False, "rd": True},
                 {"name": "I&M1", "wr": False, "rd": True}, {"name": "I&M2", "wr": False,
"rd": True},
                 {"name": "I&M3", "wr": False, "rd": True}, {"name": "I&M4", "wr": False,
"rd": True}],  # Port2
                [{"slot": "SlotNumber:0x1", "subslot": "SubslotNumber:0x1"}, {"name":
"I&M0", "wr": False, "rd": True},
                 {"name": "I&M1", "wr": False, "rd": False}, {"name": "I&M2", "wr": False,
"rd": False},
                 {"name": "I&M3", "wr": False, "rd": False}, {"name": "I&M4", "wr": False,
"rd": False}],  # Input
                [{"slot": "SlotNumber:0x2", "subslot": "SubslotNumber:0x1"}, {"name":
"I&M0", "wr": False, "rd": True},
                 {"name": "I&M1", "wr": False, "rd": False}, {"name": "I&M2", "wr": False,
"rd": False},
                 {"name": "I&M3", "wr": False, "rd": False}, {"name": "I&M4", "wr": False,
"rd": False}]  # Output
                ]
    Wireshark = pywireshark.Wireshark()

    Controller.OPC.ProjectOnline()

    flashSpi = False

    def DevicePowerOFF():
        Controller.OPC.OPCWrite("Q1_PowerSupply", True)
        time.sleep(2)

    def DevicePowerON():
        Controller.OPC.OPCWrite("Q1_PowerSupply", False)
        time.sleep(15)

    def DeviceInit():
        DevicePowerOFF()
        Controller.OPC.OPCWrite("Q0_EthBreak", False)
        Controller.OPC.OPCWrite("Q3_FlashSPI", False)
        if DeviceDK.Console.ComPort and DeviceDK.Console.ComPort.is_open:
            DeviceDK.Console.Close()
        DevicePowerON()

    def GenerateMacDcp(mac):
        if len(mac) == 17:
            new_mac = ""
            mac_list = mac.split(":")
            for byte in mac_list:
                new_mac += byte
            return new_mac
        elif len(mac) == 12:
            return mac
        else:
            raise AttributeError("MAC Address has wrong format! Check input parameter.")

    def GenerateMacAppl(mac):
        if len(mac) == 12:
            new_mac = ""
            i = 0
            for char in mac:
                if i == 2:
                    new_mac += ":"
                    i = 0
                new_mac += char
                i += 1
            return new_mac
        elif len(mac) == 17:
            return mac
        else:
            raise AttributeError("MAC Address has wrong format!\nCheck input parameter.")

    testcaseCaptureFolder = "TC_DK_" + time.strftime("%Y%m%d%H%M%S", time.localtime())


TESTCASE("Testing firmware"):
    for test in tests:
        testcaseName = "dap%s_ap%s_%s%s_%s" % (
        str(test["dap"]), str(test["app"]), test["flashType"], str(test["flashBit"]),
str(test["flashBuild"]))
        print
"\n====================================================\n=====================================
==============="
        print "Testcase: %s %s" % (str(test["enum"]), testcaseName)
        print "Firmware update testing: ecos_dap%s_ap%s_%s_%s.bin" % (test["dap"],
test["app"], test["flashBit"], test["flashBuild"])
        time.sleep(1)

        STEP ("[OPC] Initialize the station"):
            """
            Sets the station to initial state by means of OPC, gets MAC addresses of the
devices using DCP.Browse
            """
            DeviceInit()
            Controller.OPC.OPCWrite("MW_HWIDArray", test["hw_id"])
            Controller.OPC.EthBreak = Controller.OPC.OPCRead("Q0_EthBreak")
            Controller.OPC.PowSupply = Controller.OPC.OPCRead("Q1_PowerSupply")
            # Controller.OPC.NOR16        = Controller.OPC.OPCRead("Q2_FlashNOR16")
            Controller.OPC.SPI = Controller.OPC.OPCRead("Q3_FlashSPI")
            Controller.OPC.HWIDArray = Controller.OPC.OPCRead("MW_HWIDArray")
            initSet = False
        CRITERION(initSet):
            initSet = (Controller.OPC.EthBreak == False and
                       Controller.OPC.PowSupply == False and
                       # Controller.OPC.NOR16 == flashOpcSettings[1] and
                       Controller.OPC.SPI == flashSpi and
                       Controller.OPC.HWIDArray == test["hw_id"])
        UNTIL(reached_in = 1 , remains_for = 0)
        PASS:
            time.sleep(1)

        SECTION("Firmware Update"):
            time.sleep(1)

            STEP ("[DCP] Set NameOfStation and IP address for uploading FW under the test"):
                DeviceList = Project.PGPC.DCP.Browse()
                mac = GenerateMacDcp(DeviceDK.IPSuite.MAC)
                Project.PGPC.DCP.SetDeviceName(mac, DeviceDK.Name, permanent=True)
                Project.PGPC.DCP.SetIP(mac, ipAddr=DeviceDK.IPAddrForFWUpload,
mask=DeviceDK.IPSuite.SubNetMask)
                time.sleep(60)
                DeviceDK.IPSuite.IpAddress = DeviceDK.IPAddrForFWUpload
                devicesPing = False
            CRITERION(devicesPing):
                devicesNotPinged = Project.Devices.Ping()
                if Partner.Name in devicesNotPinged:
                    devicesPing = True
                elif len(devicesNotPinged) == 0:
                    devicesPing = True
                else:
                    devicesPing = False
                time.sleep(1)
            UNTIL(reached_in = 3 , remains_for = 2)
            PASS:
                time.sleep(1)

            STEP ("[CON] Open COM port on DK-ERTEC enables serial input/output"):
                DeviceDK.Console.OpenPort()
                portIsOpen = False
            CRITERION(portIsOpen):
                if DeviceDK.Console.ComPort:
                    portIsOpen = DeviceDK.Console.ComPort.is_open and True
                else:
                    portIsOpen = False
            UNTIL(reached_in = 2, remains_for = 2)
            PASS:
                time.sleep(1)

            STEP ("[CON] Generate EcosFileName. Run the FW update"):
                DeviceDK.FWUpdate.ecosFileName =
DeviceDK.GenerateName.EcosFileName(test["dap"], test["app"],

test["flashBit"], test["flashBuild"])
                DeviceDK.FWUpdate.Run(ecosFileName=DeviceDK.FWUpdate.ecosFileName,
                                      ip=DeviceDK.IPAddrForFWUpload,
                                      flashType=test["flashType"],
flashBit=test["flashBit"])
                fwUpdated = False
            CRITERION(fwUpdated):
                fwUpdated = DeviceDK.FWUpdate.fwUpdated
            UNTIL(reached_in = 0 , remains_for = 0)
            FAULT:
                continue
            PASS:
                DevicePowerOFF()
                if test["flashType"] == "SPI":
                    Controller.OPC.OPCWrite("Q3_FlashSPI", True)
                    print "SPI flash is set."
                else:
                    Controller.OPC.OPCWrite("Q3_FlashSPI", False)
                    print "NOR flash is set."
                DevicePowerON()

        SECTION ("AR and CR establishment"):
            time.sleep(1)

            STEP ("[DCP] Generate and set NameOfStation"):
                deviceNameDefault = DeviceDK.GenerateName.NameOfStation(test["dap"],
test["app"])
                mac = GenerateMacDcp(DeviceDK.IPSuite.MAC)
                Project.PGPC.DCP.SetDeviceName(mac, deviceNameDefault, permanent=True)
                time.sleep(5)
                DeviceDK.IPSuite.IpAddress = DeviceDK.GenerateName.IpAddress(test["dap"],
test["app"],
                                                                              250)
                devicesPing = False
            CRITERION(devicesPing):
                devicesNotPinged = Project.Devices.Ping()
                if Partner.Name in devicesNotPinged:
                    devicesPing = True
                elif len(devicesNotPinged) == 0:
                    devicesPing = True
                else:
                    devicesPing = False
                time.sleep(1)
            UNTIL(reached_in = 3 , remains_for = 0)
            FAULT:
                continue
            PASS:
                time.sleep(1)

            STEP ("[OPC]+[TSH] EthBreak-Disconnect. Start capture. EthBreak-Connect. Get
frames. Check AR (Connect.res and ApplRdy.req)"):
                Controller.OPC.OPCWrite("Q0_EthBreak", True)
                time.sleep(6)
                saveNameAr = "%02d_AR_DK_%s" % (test["enum"], testcaseName)
                print "Checking AR establishment. (Capture %s.pcap)" % saveNameAr
                Wireshark.StartCapture(duration=7,
                                       filter="udp",
                                       savePath=LocalSettings.PcapFilePath
+"\\"+testcaseCaptureFolder+"\\",
                                       saveName=saveNameAr,
                                       sniffers=["conn_res_sniff.lua",
"appl_req_sniff.lua"])
                Controller.OPC.OPCWrite("Q0_EthBreak", False)
                framesAR = Wireshark.GetFrames()
                devicedkConnect = False
                devicedkAppl    = False
                partnerConnect  = False
                partnerAppl     = False
                arEstablished   = False
                if framesAR:
                    for frame in framesAR:
                        if frame[1] == DeviceDK.IPSuite.IpAddress and \
                                        "CONN_RES" in frame and \
                                        "Status:OK" in frame:
                            devicedkConnect = True
                        elif devicedkConnect and frame[1] == DeviceDK.IPSuite.IpAddress and
\
                                        "APPLRDY_REQ" in frame and \
                                not "ModuleDiffBlock" in frame:
                            devicedkAppl = True
                        elif frame[1] == Partner.IPSuite.IpAddress and \
                                        "CONN_RES" in frame and \
                                        "Status:OK" in frame:
                            partnerConnect = True
                        elif partnerConnect and frame[1] == Partner.IPSuite.IpAddress and \
                                        "APPLRDY_REQ" in frame and \
                                not "ModuleDiffBlock" in frame:
                            partnerAppl = True
            CRITERION(arEstablished):
                if test["2Ports"]:
                    arEstablished = devicedkConnect and devicedkAppl and partnerConnect and
partnerAppl
                else:
                    arEstablished = devicedkConnect and devicedkAppl and not partnerConnect
and not partnerAppl
                time.sleep(1)
            UNTIL(reached_in = 10 , remains_for = 0)
            FAULT:
                if not devicedkConnect or not devicedkAppl:
                    continue
            PASS:
                time.sleep(1)

            STEP ("[TSH] Check CR (RTC1 frames)"):
                devicedkRtc     = False
                partnerRtc      = False
                crEstablished   = False
                saveNameCr = "%02d_CR_DK_%s" % (test["enum"], testcaseName)
                print "Reading RTC1 frames. (Capture %s.pcap)" % saveNameCr
                Wireshark.StartCapture(duration=1,
                                       savePath=LocalSettings.PcapFilePath
+"\\"+testcaseCaptureFolder+"\\",
                                       saveName=saveNameCr,
                                       sniffers=["rtc1_sniff.lua"])
                framesRtc = Wireshark.GetFrames([DeviceDK.IPSuite.MAC, None],
[Partner.IPSuite.MAC, None])
                if framesRtc:
                    for frame in framesRtc:
                        if frame[1] == DeviceDK.IPSuite.MAC and (not "DataStatus:OK" in
frame or not "TransferStatus:OK" in frame):
                            devicedkRtc = False
                            break
                        elif frame[1] == DeviceDK.IPSuite.MAC and "DataStatus:OK" in frame
or "TransferStatus:OK" in frame:
                            devicedkRtc = True
                    if test["2Ports"]:
                        for frame in framesRtc:
                            if frame[1] == Partner.IPSuite.MAC and (not "DataStatus:OK" in
frame or not "TransferStatus:OK" in frame):
                                partnerRtc = False
                                break
                            elif frame[1] == Partner.IPSuite.MAC and "DataStatus:OK" in
frame or "TransferStatus:OK" in frame:
                                partnerRtc = True
            CRITERION(crEstablished):
                if test["2Ports"]:
                    crEstablished = devicedkRtc and partnerRtc
                else:
                    crEstablished = devicedkRtc
                time.sleep(1)
            UNTIL(reached_in = 10 , remains_for = 0)
            FAULT:
                if not devicedkRtc:
                    continue
            PASS:
                time.sleep(1)

        SECTION ("Check MRP configuration on the devices Interface submodule"):
            time.sleep(1)

            STEP ("[TSH]+[OPC] Read PDInterfaceMrpDataReal"):
                deviceInterfaceMrp = False
                readPdRealData = Controller.OPC.OPCRead("M0_ReadInterfaceMrpReal")
                if readPdRealData:
                    Controller.OPC.OPCWrite("M0_ReadInterfaceMrpReal", False)
                    time.sleep(1)
                saveNamePdInterface = "%02d_InterfaceMrpReal_%s" % (test["enum"],
testcaseName)
                print "Reading PDInterfaceMrpDataReal. (Capture %s.pcap)" %
saveNamePdInterface
                Wireshark.StartCapture(duration=10,
                                       filter="udp",
                                       savePath=LocalSettings.PcapFilePath
+"\\"+testcaseCaptureFolder+"\\",
                                       saveName=saveNamePdInterface,
                                       sniffers=["interfacemrpreal_sniff.lua"])
                time.sleep(3)
                Controller.OPC.OPCWrite("M0_ReadInterfaceMrpReal", True)
                frameInterfaceMrp = Wireshark.GetFrames([DeviceDK.IPSuite.IpAddress, None])
#
            CRITERION(deviceInterfaceMrp):
                if frameInterfaceMrp and len(frameInterfaceMrp) == 1:
                    if test["MRP"] and test["2Ports"]:
                        for item in frameInterfaceMrp:
                            if "Status:OK" in item:
                                deviceInterfaceMrp = True
                    elif not test["MRP"] or not test["2Ports"] and "Status:ERROR" in
frameInterfaceMrp:
                        deviceInterfaceMrp = True
                    else:
                        deviceInterfaceMrp = False
                time.sleep(1)
            UNTIL(reached_in = 0 , remains_for = 0)
            PASS:
                time.sleep(1)


        SECTION ("Checks DataSets"):
            print "DataSets test: %s" % str(test["DataSets"])
            if test["DataSets"]:
                time.sleep(1)

                STEP ("[TSH]+[OPC] Check I&M Datasets Write responses"):
                    saveNameDatasetsWrite = "%02d_DataSets_Write_%s" % (test["enum"],
testcaseName)
                    print "Datasets for Write.res. (Capture %s.pcap)" %
saveNameDatasetsWrite
                    Wireshark.StartCapture(duration=7,
                                           filter="udp",
                                           savePath=LocalSettings.PcapFilePath + "\\" +
testcaseCaptureFolder + "\\",
                                           saveName=saveNameDatasetsWrite,
                                           sniffers=["imdata_sniff.lua"])
                    time.sleep(3)
                    Controller.OPC.OPCWrite("M1_WriteIMData", True)
                    framesDataSets = Wireshark.GetFrames([DeviceDK.IPSuite.IpAddress, None])
# DeviceDK.IPSuite.IpAddress
                    writeResList = []
                    for frame in framesDataSets:
                        if "WRITE_RES" in frame:
                            if "Status:OK" in frame:
                                writeResList.append(True)
                            elif "Status:ERROR" in frame:
                                writeResList.append(False)
                    deviceWriteResList = []
                    for i in range(1, 6):
                        for j in range(6):
                            if hw_sub[test["hw_id"]][j]:
                                deviceWriteResList.append(dataSets[j][i]["wr"])
                CRITERION(datasetsWriteOk):
                    datasetsWriteOk = writeResList == deviceWriteResList
                    time.sleep(1)
                UNTIL(reached_in = 0 , remains_for = 0)
                PASS:
                    time.sleep(1)

                STEP ("[TSH]+[OPC] Check I&M Datasets Read responses"):
                    saveNameDatasetsRead = "%02d_DataSets_Read_%s" % (test["enum"],
testcaseName)
                    print "Datasets for Read.res. (Capture %s.pcap)" % saveNameDatasetsRead
                    Wireshark.StartCapture(duration=7,
                                           filter="udp",
                                           savePath=LocalSettings.PcapFilePath + "\\" +
testcaseCaptureFolder + "\\",
                                           saveName=saveNameDatasetsRead,
                                           sniffers=["imdata_sniff.lua"])
                    time.sleep(3)
                    Controller.OPC.OPCWrite("M2_ReadIMData", True)
                    framesDataSets = Wireshark.GetFrames([DeviceDK.IPSuite.IpAddress, None])
# DeviceDK.IPSuite.IpAddress
                    readResList = []
                    for frame in framesDataSets:
                        if "READ_RES" in frame:
                            if "Status:OK" in frame:
                                readResList.append(True)
                            elif "Status:ERROR" in frame:
                                readResList.append(False)
                    deviceReadResList = []
                    for i in range(1, 6):
                        for j in range(6):
                            if hw_sub[test["hw_id"]][j]:
                                deviceReadResList.append(dataSets[j][i]["rd"])
                CRITERION(datasetsWriteOk):
                    datasetsReadOk = readResList == deviceReadResList
                    time.sleep(1)
                UNTIL(reached_in = 0 , remains_for = 0)
                PASS:
                    time.sleep(1)


        SECTION ("Checks IOCycleTime and DHT timer"):
            print "DHT test: %s" % str(test["DHT"])
            time.sleep(1)
            if test["DHT"]:
                for io_cycle in ioCycles:

                    STEP ("[DCP] Set name for appropriate IOCycle"):
                        print "%s IOCycle for %s ms SendClock" % (io_cycle,
ioCycles[io_cycle])
                        deviceNameIo =
DeviceDK.GenerateName.NameOfStationForIo(deviceNameDefault, io_cycle)
                        mac = GenerateMacDcp(DeviceDK.IPSuite.MAC)
                        Project.PGPC.DCP.SetDeviceName(mac, deviceNameIo, permanent=True)
                        time.sleep(20)
                        DeviceDK.IPSuite.IpAddress =
DeviceDK.GenerateName.IpAddress(test["dap"], test["app"],

ioCycles[io_cycle])
                        devicesPing = False
                    CRITERION(devicesPing):
                        devicesNotPinged = Project.Devices.Ping()
                        if Partner.Name in devicesNotPinged:
                            devicesPing = True
                        elif len(devicesNotPinged) == 0:
                            devicesPing = True
                        else:
                            devicesPing = False
                        time.sleep(1)
                    UNTIL(reached_in = 0 , remains_for = 0)
                    FAULT:
                        continue
                    PASS:
                        time.sleep(1)

                    STEP ("[TSH] Check time interval with appropriate IOCycleTime for 5
random frames"):
                        saveNameIoCycle = "%02d_IOCycle_%s_%s" % (test["enum"], io_cycle,
testcaseName)
                        print "Time intervals for %s IOCycle. (Capture %s.pcap)" %
(io_cycle, saveNameIoCycle)
                        Wireshark.StartCapture(duration=1,
                                               savePath=LocalSettings.PcapFilePath
+"\\"+testcaseCaptureFolder+"\\",
                                               saveName=saveNameIoCycle,
                                               sniffers=["rtc1_sniff.lua"])
                        framesRtcIoc = Wireshark.GetFrames([DeviceDK.IPSuite.MAC, None])
                        limhigh = (ioCycles[io_cycle] + ioCycles[io_cycle] * 0.15) * 0.001
                        limlow  = (ioCycles[io_cycle] - ioCycles[io_cycle] * 0.15) * 0.001
                        ioCycleTimeGood = False
                    CRITERION(ioCycleTimeGood):
                        if framesRtcIoc:
                            for i in range(5):
                                j = randint(1, len(framesRtcIoc)-1)
                                frame1 = framesRtcIoc[j]
                                frame0 = framesRtcIoc[j-1]
                                difference = float(frame1[0])-float(frame0[0])
                                if limlow<= difference <= limhigh:
                                    ioCycleTimeGood = True
                                    print "Time interval for frame sent in %s is OK" %
str(frame0[0])
                                else:
                                    ioCycleTimeGood = False
                                    print "Time interval for frame sent in %s is WRONG.
(Capture %s.pcap)" % (str(frame0[0]), saveNameIoCycle)
                                    break
                        time.sleep(1)
                    UNTIL(reached_in = 0 , remains_for = 0)
                    PASS:
                        time.sleep(1)

                    STEP ("[TSH]+[OPC] Check DHT timer"):
                        saveNameDht = "%02d_DHT_%s_%s" % (test["enum"], io_cycle,
testcaseName)
                        print "Check DHT timer for %s IOCycleTime. (Capture %s.pcap)" %
(io_cycle, saveNameDht)
                        Wireshark.StartCapture(duration=,
                                               savePath=LocalSettings.PcapFilePath
+"\\"+testcaseCaptureFolder+"\\",
                                               saveName=saveNameDht,
                                               sniffers=["rtc1_sniff.lua",
"err_dht_sniff.lua"])
                        time.sleep(3)
                        Controller.OPC.OPCWrite("Q0_EthBreak", True)
                        framesRtcDht = Wireshark.GetFrames([DeviceDK.IPSuite.MAC, None],
[Controller.IPSuite.MAC, DeviceDK.IPSuite.MAC])
                        Controller.OPC.OPCWrite("Q0_EthBreak", False)
                        lastIocFrameNum     = 0
                        firstDevRtcFrame    = False
                        rtcFramesNum        = 0
                        dhtTimerPass        = False
                        if framesRtcDht:
                            for i in range(len(framesRtcDht)):
                                frame = framesRtcDht[i]
                                if frame[1] == Controller.IPSuite.MAC and frame[2] ==
DeviceDK.IPSuite.MAC and \
                                                "RTC1" in frame:
                                    lastIocFrameNum = i
                            for i in range(lastIocFrameNum + 1, len(framesRtcDht)):
                                frame = framesRtcDht[i]
                                if not firstDevRtcFrame and frame[1] == DeviceDK.IPSuite.MAC
and "RTC1" in frame:
                                    firstDevRtcFrame = True
                                    rtcFramesNum += 1
                                elif firstDevRtcFrame and frame[1] == DeviceDK.IPSuite.MAC
and "RTC1" in frame:
                                    rtcFramesNum += 1
                                elif firstDevRtcFrame and frame[1] == DeviceDK.IPSuite.MAC
and "ERR-RTA-PDU" in frame:
                                    break
                    CRITERION(dhtTimerPass):
                        dhtTimerPass = firstDevRtcFrame and 3 <= rtcFramesNum <= 5
                        time.sleep(1)
                    UNTIL(reached_in = 0 , remains_for = 0)
                    PASS:
                        time.sleep(1)

FINALLY:
    DeviceInit()
```