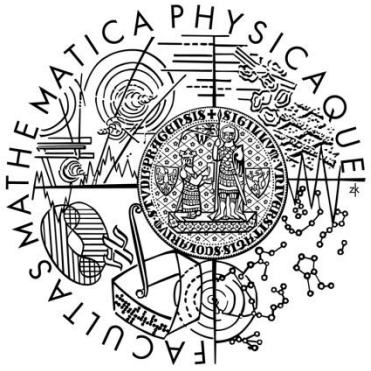
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Ondrej Kováč

Automobile mirror testing

Supervisor: RNDr. Jan Kofroň Ph.D.

I would like to thank to RNDr. Jan Kofroň Ph.D. for the time and advice dedicated to help me to accomplish this software work. I also appreciate the help of Peter Kurhajec and Tomáš Kováč who introduced me into the issue of automation technology and lent me the necessary hardware for the testing purpose.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague 23.7.2012

Název: Testování automobilových zrcadel

Autor: Ondrej Kováč

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Jan Kofroň, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Testování je součástí výrobního procesu v mnoha průmyslových odvětvích. Slouží jako prevence před vypuštěním chybných kusů na trh. Manuálně skládaná automobilová zrcadla procházejí testováním, při kterém se ověřují základní požadavky výrobce a vyřadí se špatně vyrobené kusy. Rozhodli jsme se tento proces zautomatizovat, což bude mít za následek zrychlení a zvýšení spolehlivosti testování. Pro potřeby testování slouží speciálně navrženy přístroj. Testovací přístroj je kompletně ovládán prostřednictvím aplikace spuštěné na běžném počítači. Paralelním spouštěním testů jsme dosáhli znatelného zrychlení. Vývoj projektu také přispěl ke vzniku knihovny určené k ovládání průmyslového hardware.

Klíčová slova: automatizace, RTU, vizualizace, EtherCAT, montážní linka

Title: Automobile Mirror Testing

Author: Ondrej Kováč

Department / Institute: Department of Distributed and Dependable Systems

Supervisor or the bachelor thesis: RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract: In many sectors of the industry, the testing is a part of the production process. It prevents defective goods to appear on the market. Manually assemble automobile mirrors are tested to meet the minimal requirements of the producer and to reject badly mounted pieces. We decided to automate this process in order speed up and ensure the reliability of the testing. A special hardware was designed for this purpose. It is completely controlled by our application running on a regular PC. One of the improvements we have accomplished is a noticeable acceleration of the testing achieved by the parallel test

execution. Development of this project contributes to the creation of a library for controlling industrial hardware.

Keywords: automation, RTU, visualization, EtherCAT, assembly line

Contents

1. Introduction

In many sectors of industry, man-power is still being used as a primary resource. Because of the imperfection of human beings, many errors and defects may appear on final product. Defective goods on the market usually have serious consequences such as high additional expenses and the loss of goodwill. Testing is therefore an inseparable part of the production process. In this work, we focus on assembly lines in an automobile factory producing mirrors. These are made manually and therefore defects arise. Each mirror passes through several assembly stations mounting its components. At the end, one testing operator checks every piece. Correct mirrors are packaged and prepared for dispatching, while defected are excluded from the production. There are many kinds of mirrors of different brands, for a passenger car or a van, electrical or manual, with a direction light or without, etc. Contrary to what one would expect, nowadays automobile mirrors are quite complex and contain several electrical circuits. These need to be tested very well to prove that the currents are in range and that they would not cause any harm on the mirror in the future. Also we need to make sure that there are no badly mounted, missing or damaged parts. Miscellaneous aspects can be tested on the mirror according to requirements of the customer.

To ensure the testing reliability and to increase the productivity we decided to make the testing process automated. A special hardware is designed for this purpose, which is manipulated by a human operator. The tester is controlled by a computer over the network. Testing is very often the bottleneck in the assembly line. Therefore, acceleration of this process will speed up the entire production. Considering the previous requirements we are going to create an application which maintains the tester hardware and automates the testing part of the production process. We would like to achieve as much scalability as possible to allow hardware changes with minimum effort and to support various kinds of tests to be executed depending on the hardware configuration. As a possibility of the acceleration we considered to perform some tests in parallel. Visualization can be added as an additional feature for better user experience. Frequently occurring failures on the mirror may indicate some deeper error caused by an assembly line defect, improperly designed production process or even unqualified operating personnel. Recording and

storing the entire testing process can reveal some of these insufficiencies. It is also useful for monitoring the efficiency and allowing the traceability of manufactured articles.

The remainder of the work is organized as follows. Chapter **2** analyzes the above described application and its requirements and outlines basic decisions made during the development. Chapter **3**, based on the previous chapter, describes implementation details and is intended to be used as the programmer manual. In chapter **4** the entire project is evaluated and it is illustrated how our objectives were fulfilled. Chapter **0** provides an overview of the developed software work.

2. Analysis

Operator in the factory inserts the mirror into the tester which is fully controlled by a service computer where **M**irror **T**esting **S**ystem (MTS) application is running. In order to support the functionality described in the previous section, the application will consist of the several parts with the core one providing testing logic of desired tests. The logic needs to be customizable by user defined parameters. We need a way of enabling the user to enter them. Also there are usually several mirror models produced in one factory having various components which require different tests[1] (different parameters) to be supported. The tester is controlled by the testing logic from the application communicating over (local) network. The communication layer is interfacing an exchangeable and configurable hardware. Provided tests are visualized through the graphical user interface. Tests results are kept in a database where they are accessible for statistics and other reference.

When there are too many parameters the user interface can be quite large and complicated and the testing setup may be considerably time-consuming. User-friendly interface which does not require hours of operator's attention will definitely prevent many failures caused by the loss of user's attention. Also, our ambition is to simplify the application control in such a way that no special training of testing operators will be necessary.

In the following chapters we are going to describe the main difficulties we have faced during the development and decisions we have made to build the application. We focus on the possible future extensibility, good programmer manners and the usability of our application.

Chapter **2.2** describes details about the testing hardware and its control from the application. Chapter **2.3** shows how application is incorporated to the assembly line and basic interaction with users. Chapter **2.4** gives reasons for the usage of graphical user interface. Chapter **2.5** focuses on configuration of tests. The requirement of data storage is covered in **2.6**. Finally, chapter **2.7** describes how test execution is performed.

[1] Some mirrors have a direction light while other not.

2.1. Programming Language

It is not the programming language that solves the given problem, but still it has serious impact on the complexity of our solution.

MTS is intended to be an application up to standards of modern software. In order to satisfy this demand, it is reasonable to use appropriate instruments. The professional developers advise to *take advantage of market maturity by taking advantage of existing platform and technology options. Build on higher level application frameworks where it makes sense, so that you can focus on what is uniquely valuable in your application rather than recreating something that already exists and can be reused* [1].

The modern programming languages we have taken into consideration are Java, C# and C++. All of them are widely used so there is an extensive support and many third party libraries. Comparing Java and C# to C++, memory management is not the programmer's responsibility anymore.

The *advantage of existing platform and technology options* we find important is the comfort of the programmer. It helps us to focus on what is important in our application rather than on the difficulties with the implementation. The familiarity of the C# language is the main reason for choosing it instead of Java, even if we lose some portability options. When talking about C#, rather than the language, we deliberate the .NET platform. It is not only the garbage collector that simplifies the development, but many advantages of this language (and underlying .NET platform) that help to create powerful and stable applications. Some of these are: safety references, system of exceptions handling and generics. We also appreciate the .NET support for UI programming.

Although the garbage collector simplifies many things, it is also the root of a complication. During the control of the tester we need to ensure some regularity of the communication. Let's say some kind of "real-time" operations. Nondeterministic run of the GC can negatively influence the communication latency. In spite of that, we have decided to use C#. As we shall demonstrate later, a small delay of a communication message is acceptable and can only lead to a negligible impreciseness of the measurement.

The efficiency of C++ comparing to C# is discussable. It was not the ground of our decision, but it could be considered as well. Some of the specialists say [2] that, in spite of the fact that C++ is a native compiled code, C# programs can sometimes be faster than C++ ones, since the C# code is compiled twice and can be adapted to the current architecture. Whether it is true or not, we have chosen C# for the reasons given above.

2.2. Hardware Control

The essential part of the implementation is communication layer connecting the application with the tester hardware. Tester consists of several components such as measurement instruments of electrical current and distance, sensors, actuators and hydraulic arms. These are controlled by the application using *master-slave model* [3]. The application as a master sends commands to slaves (tester components) and waits for an answer. The important point in this model is that slave never actuates itself, only replies to incoming requests.

2.2.1. Remote Terminal Unit[2]

Communication between the tester components and the application is realized using hardware modules called remote terminal units (RTU). *RTU is a microprocessor-controlled electronic device that interfaces objects in the physical world* [4]. It contains a piece of addressed memory accessible to both sides over network, which is usually divided into several memory cells referred to as *channels*. The application controls the tester by writing defined values to some channels, while tester responds by writing result values to other channels. As a consequence of using RTU we have a uniform interface for controlling various hardware components (see **Figure 2.2.1**).

[2] Different suppliers use their proprietary denomination i.e. slot, remote terminal block or simply terminal.

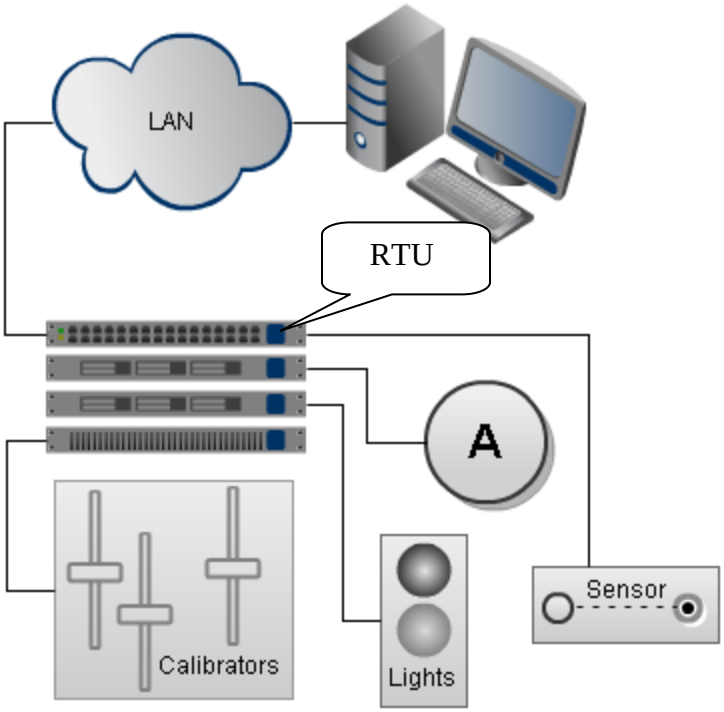


Figure 2.2.1 Application communicating with tester components over local network through terminals

From the application point of view channels are divided into two groups according to the direction of the communication flow:

- Inputs – all readable channels. These are used by the hardware components to communicate their current state to the application.
- Outputs – all writable channels. Application uses them to control hardware components.

As a simplification we consider all outputs to be a subset of input channels (**Figure 2.2.2**). When current state of *the physical world is read* all channels are accessed. Outputs need to be initialized before the first access. We also expect that an output channel may change its value in some extraordinary situation and we find it out.

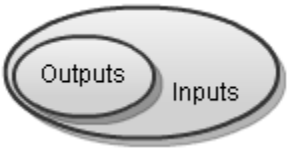


Figure 2.2.2 Sets of input/output channels

Channels are divided according to their size (and method of use) into two groups:

- Digital – logical value of one bit size. Used to switch on/off electrical circuits of actuators, hydraulic system, lights etc. (output channel) and for sensors indicating presence of some elements on the mirror or tester (input channel).
- Analog – numeric value of at least one byte size[3]. Used for current value of ammeters, voltmeters and calibrators[4].

This implies the existence of four channel types – digital input, digital output, analog input and analog output. In this application analog outputs are not used.

2.2.2. Communication Basics

When tester components are controlled by the application, remote channels must be accessed. We have considered two following concepts:

- On demand – read or write channel value immediately when it is necessary.
- Cyclic – cyclically collect all requests and then write or read them at once.

Disadvantage of on demand access is pretty large overhead, because with each request to read or write a channel all auxiliary data are sent (addressing, synchronization, etc.). Generally, this type of communication is not very useful when using *master-slave model* [3] as the slave only responds on received commands and we have to repeatedly check for its state. A command within the meaning "Do this job and let me know when you finish" is not possible. Cyclic access is on the other hand useful for its simplicity, as the IO[5] operations are centralized. Commonly used operation flow in automation control system is the following (see also **Figure 2.2.3**):

1) Read inputs
2) Execute application logic
3) Write outputs

[3] Size of analog channel may vary according its type and producer. For example: 8, 12, 16, 24 or 32 bits.

[4] Distance measurement device

[5] Input/output – any read or write operation.

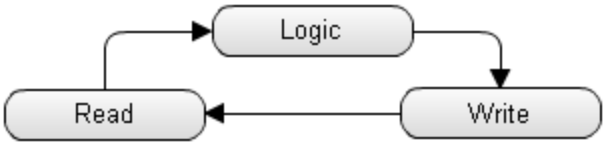


Figure 2.2.3 Communication loop: read input channels, execute logic and write outputs. Based on [5]

In other words, make an image of the entire terminals memory in the application memory, use it to execute application logic – operations will affect the memory image (inputs are locked and cannot be changed), synchronize application image with the terminals.

EtherCAT [6] communication protocol used in industrial technology is supporting the cyclic scenario naturally. As explained in [7] *EtherCAT Network is using communication principle master-slave. Data are not sent to slave devices as single Ethernet frames, but each frame is passing through all devices equipped with a special hardware having at least two Ethernet ports allowing to process frames while they are passing through with a minimal time delay (at nanoseconds level).*(see **Figure 2.2.4**)

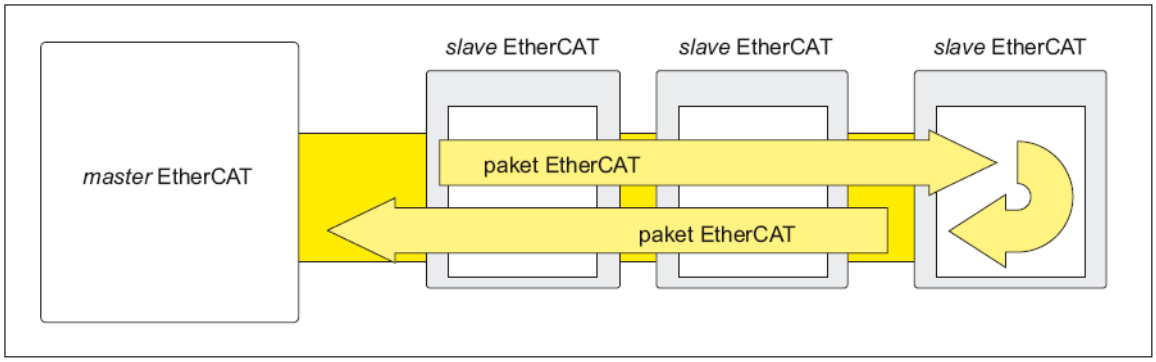


Figure 2.2.4 EtherCAT frame traveling across the network (Taken form [7])

Example 2.2.1 Measuring power-fold[6] current

As an example consider a scenario where the current of mirror power-fold is measured. During this test a mirror is folded and unfolded again, while electrical current must not exceed defined maximum value.

[6] Power-fold is an electronic system allowing automatic mirror folding and unfolding. It is installed inside the mirror.

	Read	Write	Description
1.	All	-	Read all channels to initialize application memory image. Go to 2.
2.	-	Fold = true	Start mirror folding. Do not measure current yet. Go to 3.
3.	PowerfoldCurrent, IsFolded	-	Check whether current is under the maximum. Repeat step 3 until IsFolded is true. Then go to 4
4.	-	Fold = false	Stop folding. Go to 5 immediately.
5.	PowerfoldCurrent, IsUnfolded	Unfold = true	Check whether current is under the maximum. Repeat step 5 until IsUnfolded is true. Then go to 6.
6.	-	Unfold = false	Stop unfolding.

For complete list of tests see the user manual.

2.2.3. Common Protocols

In industrial technology various communication protocols are used according to the purpose. The usage of a particular protocol requires dedicated hardware which is the only interface between tester hardware and tester application. Independence on communication protocol implies independence on selected hardware.

We have considered two commonly used protocols and another one just for the purpose of testing. Specifically: TCP modification of Modbus protocol and EtherCAT. For both of them communication library provided by hardware supplier is used. The main difference is ISO/OSI layer on which they are communicating. While Modbus puts its commands into standard TCP packets, EtherCAT overpasses all network layers and uses directly Ethernet frames[7].

According to the investigation of EtherCAT Technology Group *EtherCAT remains the fastest industrial Ethernet technology* [8] in the world. This fact makes EtherCAT protocol adequate for usage with our application. But even slower protocols can be used without significant loss of measurement preciseness. There are other advantages, mainly from the point of business strategy view that lead us to prefer EtherCAT. The industrial network based on EtherCAT protocol is not limited by the number of nodes. There is

[7] EtherCAT specification also allows communication on transport layer. Commands are packed into the UDP datagrams.

incomparably major support of various products running on EtherCAT and even networks with different protocols[8] can be interconnected together.

2.3. Shift Workflow

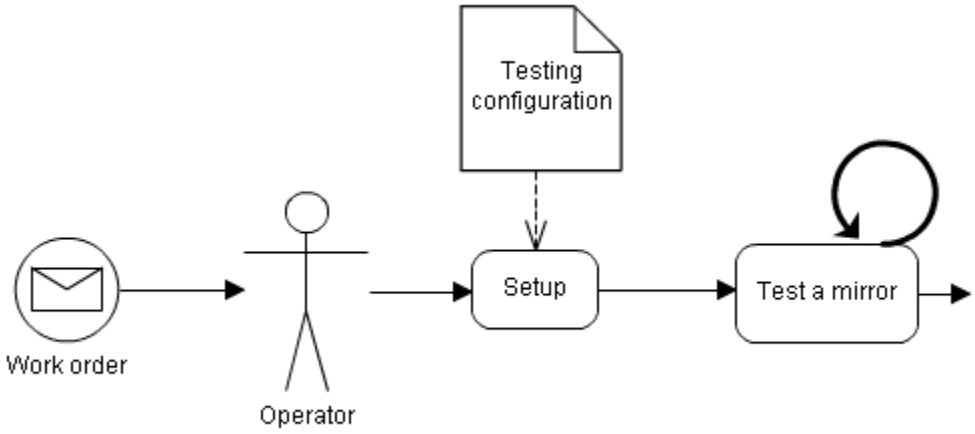


Figure 2.3.1 Workflow of the testing process

The manufacturing process starts with a work order (see **Figure 2.3.1**) telling us the type and number of mirrors to be produced. The order is processed by an operator who makes ready the testing machine, starts up the service computer and selects test configuration for given mirror type. Test configuration is meant to be a set of tests that should be executed on each mirror and their parameters. It is not intended to change any configuration setting during the testing.

When the machine is prepared and configuration selected correctly the testing process may start. Operator inserts a mirror and initiates the *testing sequence* – all the tests are executed. This action is repeated until all mirrors are tested.

[8] Profinet, Profibus, CANopen, …

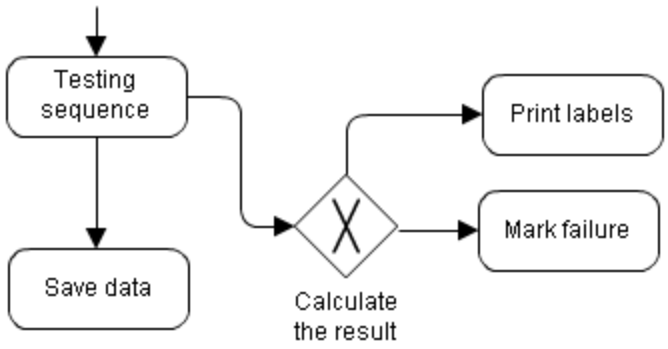


Figure 2.3.2 Process of one mirror test

During the testing of each mirror a given set of tests is executed (testing sequence on **Figure 2.3.2**). Almost any of them produce result data – measured values, and the status whether it was completed correctly or failed – which are saved for future use. Failed mirrors are marked as defective and excluded. The mirror is considered to be correct if all tests finished correctly. In this case it is marked with a label proving that it has been tested.

The entire factory process of handling the work order, setting up and executing testing sequences is referred to as a *shift*. Our application is intended to assist the operator during the shift lifetime.

2.3.1. Machine Control

Tester machine is controlled by a computer (1) connected through Ethernet. At the beginning of a shift a service computer is started. Launching the application enables the machine to be switched on (2). The computer also allows the operator to follow current testing state through application visualization (3). At the beginning of a test, the mirror is inserted to the machine (4). A special exchangeable fixture (5) allows only a mirror of a right type and orientation to be inserted. A mirror cable is connected to machine connector (6) which allows controlling its electrical circuits. Testing of the mirror is started with the START button on the main board (2). Before any test, the machine is closed (7) and the mirror is strongly fixed. During the testing of mirror glass (pull-off and travel tests) additional components are elevated from the bottom (8). After all tests have been finished the result is indicated with a colored light (9) and the machine is opened (7)

so that the mirror can be removed. For the next test a new mirror is inserted and START button is pressed. **Figure 2.3.3** shows a basic scheme of tester components.

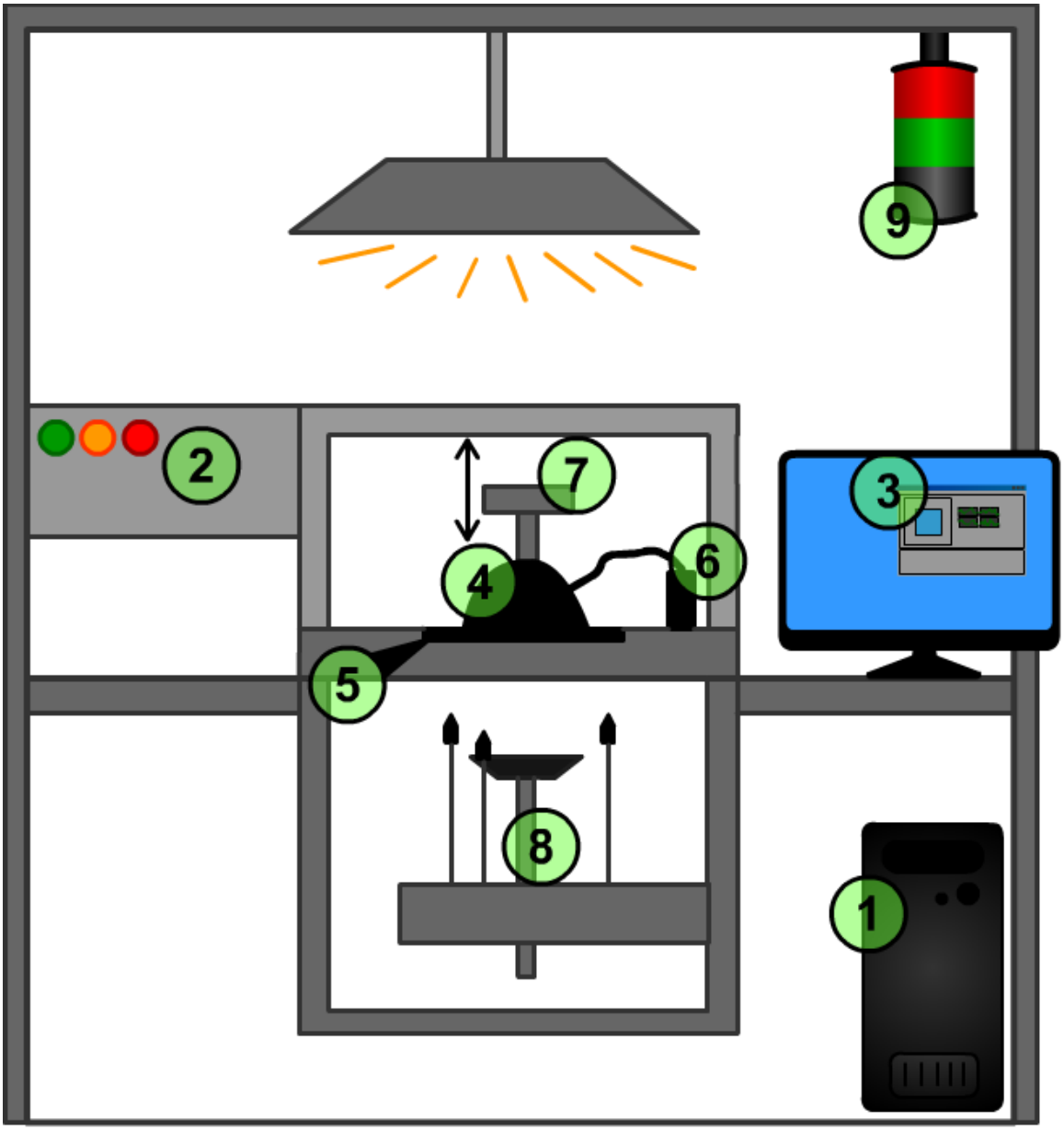


Figure 2.3.3 Tester machine

2.4. User Interface

Setting up test configuration is quite complicated process. Each test has several different parameters to be set up. This is the reason why we have decided to use a graphical user interface for our application. Configuration of tests through the console or a text file could be possible. But GUI simplifies many things and even an inexperienced

user is able to go through it and understand it quickly. It also incorporates many technical details such as unit of a parameter or maximal and minimal allowed value and helps to avoid mistakes.

As discussed in **2.1** the application is developed in C#. .NET provides two concepts for implementation of an application with user interface: WinForms and WPF [9]. Considering the requirements of our application we decided to use WPF for several reasons:

- It allows a programmer to create dynamic UI based on data templates [10]. Data templates will definitely help us to handle changes of tests and parameters with minimum effort.
- Code of user interface is separated in a single file which simplifies the maintenance and makes possible future localization easier [11].
- Generally WPF offers more sophisticated ways of creating custom controls. We use this advantage when creating special controls supporting unit values, which are necessary on many places in the application, or graph controls for test visualization.
- WPF supports creation of simple 3D objects (in a very programmer-friendly way) directly from XAML [12] and we have used it for displaying the current position of mirror glass.

WPF technology brings not only advantages. It is the performance[9] that makes WinForms still being used in many applications.

2.4.1. GUI Scheme

Application user interface can be divided into several independent parts. The essential one is the testing part containing visualization elements and controls for regulating the process. Some activities during testing are not visualized, i.e. that a particular test has been started/finished with correct/failed result etc. These and similar statuses or messages are displayed to user in an output console as text messages.

[9] Performance comparison: http://www.kynosarges.de/WpfPerformance.html

As we have already mentioned, configuration of testing parameters is a time-consuming process. Usually, there are the same parameters for one mirror type. So default ones may be created once and then reused or propagated also to other computers. For this purpose an editor is designed. It allows a user to create or edit existing files with testing parameters. Application requires many settings of hardware (channels, calibration, printer …) or just user preferences (operators). These need to be presented on the user interface as well as the data generated during the testing and reviewed by the operator later.

Considering these requirements we have decided to use a library for docking panels, which helps us to divide user interface into several independent parts. It is expected that users will spend hours working with this application so it is reasonable to let them adjust their workspace according to their preferences. Dockable panels perfectly satisfy this demand.

2.5. Testing File

Testing requires a configuration of used tests and their parameters. As a storage medium regular file is used – it allows the user to reuse configuration on other computers. In order to achieve extensibility with minimum effort, the editor should be independent on current file format. As we have specified earlier, newer version of testing hardware may require additional tests to be supported, or modifications of tests execution logic may need more or less parameters. In the best case the editor supports both newer and older formats.

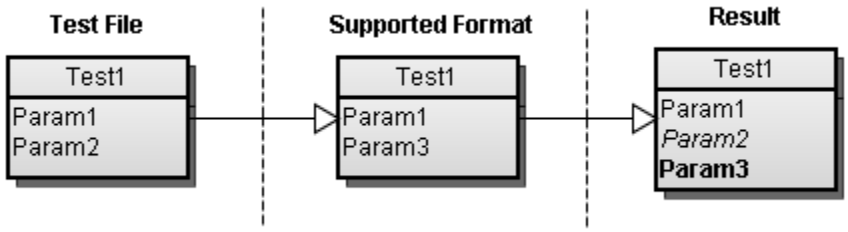


Figure 2.5.1 Conversion of testing file to different format

Figure 2.5.1 shows how the conversion to supported format works. Suppose that we have a file with test configuration and we open it in some version of our application.

It contains one test (Test1) with two parameters (Param1 and Param2). Param2 is not supported by the current version so it is not displayed to user. But we leave it in the file without any change, as it may be displayed in other version. When the file is saved unrecognized parameters are deleted. On the other hand Param3 supported by the current application is not included in the given file. It is displayed to the user with default values and added to the file when it is saved. So conversion to newer files can be done easily and even old formatted files can be viewed. The same principle is applied on the tests.

2.5.1. Expected Parameter Values

Test configuration contains a collection of tests that could be executed. Some of them could be disabled – will be ignored during testing. But we would like to save all of them to configuration as the operator may only temporarily disable a test and we do not want to lose its parameters and other settings. We remember for each test a value indicating whether it is enabled.

The test contains a collection of parameters that may vary. The parameters are of various types, but usually only primitive ones such as a number or a logical value. Numeric parameters are values of a certain unit. Time is also considered to be a numeric value of the milliseconds unit.

We expect at least these types of parameters: real, integer, text, logical value and enumeration value. However, as mentioned in chapter **2.2.1**, there are only two types of inputs channels – digital and analog. This implies that for test execution only two types of parameters are necessary. Logical value of a digital input (presence of a mirror component) and numeric value of an analog input (current in the electrical circuits, etc.). All other types are only informative or they may be used for label printing.

2.5.2. File Format

During the development we have considered several techniques of file storage. The very first idea was a .NET serialization. When using the editor or running the testing, an instance of object representing the testing configuration must exist in the memory. Serialization gives us a powerful instrument to do so and we do not care about the implementation of the file format.

.NET provides two basic types of serialization – XML and binary. Binary has *better performance* [13], but its format is highly dependent on .NET[10]. Serialization output contains object type and full assembly name where it is defined. On de-serialization specified assembly is loaded and instance of object is created.

XML serialization using `SoapFormater`[11] is obsolete. `XmlSerializer`[12] serializes only public fields and properties and does not support cyclic structures, which is not a difficulty in our case. Finally, we have decided to use our proprietary format based also on XML, which we considered to have several important advantages:

- Human readable format; can be changed even without any special editor
- Widely used format so the parsers are very effective
- Easy to validate using DTD [14]
- Easy to extend

The main reason why not to use already implemented `XmlSerializer` is that we want to have absolute control over the file format. We need to know when the format of any test changes. While a serialized object describes definition of a particular type, our file format holds data in a logical structure. Only parameter values and some test properties significant for execution behavior are serialized and the rest is initialized according to other application settings. Serialization will only complicate the entire process.

2.5.3. File Description

Definition of used tests and their parameters may change in different versions of the application. It would be convenient to have a specification of supported test types with parameters. The advantage we are trying to approach is an easy way to extend application functionality without extensive code changes. If the editor is a standalone application (or

---

[10] Format of binary serialization hasn't changed since .NET 1.0 and probably we can expect not to be changed in the future versions.

[11] `System.Runtime.Serialization.Formatters.Soap` namespace in `System.Runtime.Serialization.Formatters.Soap` assembly

[12] `System.Xml.Serialization` namespace in `System.Xml` assembly

an isolated module), which only produces the testing file as an output, the new type of test supported by the hardware will not require any change to the editor code.

The format specification is very simple – a collection of tests where each has a collection of parameters with a default value. A simple way we can handle it is to hardcode it. When a new file is created, the (hardcoded) collection of tests is initialized and saved to XML file as described in **2.5.2**.

Rather than defining it in the code, we have used a configuration file to define the format of a testing file. It is much more transparent as the structure of tests may contain many entries. The configuration file contains settings that could be changed by the customer, i.e. the test name or description. That way we provide a chance to do so in one place. We would like to delay the process of the customization of GUI up to the deployment. The customer decides how editor should look like.

The configuration file containing the description of the used tests and parameters is referred to as a *template*. The principle is similar to the MS Office Word template file. When a new file is created, the template is opened what results in a new file with default values. When an existing file is opened, new file is created from template and overwritten by the opening one. Notice that the template contains settings and localization data that are not saved to the file. Also the default values of parameters are placed here, so the customer may adjust them in such a way that new file will be created very quickly just by editing some of them.

The XML format for template file is adequate because the customer needs to modify it in a text editor manually. No template editor is shipped with this application. It is also possible to apply an XSLT transformation on the template and create a new file with default values.

2.6. Data Storage: Test Measurement Results

During the testing a quite large amount of data is produced. Some of them we need to conserve for future use, even though the most significant is the result telling us whether a mirror has passed or not. Frequently occurring failures on the mirror may indicate some deeper error caused by an assembly line defect, improperly designed

production process or even unqualified operating personnel. Recording and storing the entire testing process can reveal some of these insufficiencies. It is also useful for monitoring the efficiency and allowing the traceability of manufactured articles.

Running a testing sequence can be done only by identified personnel. Business strategy requires keeping track over operator's activities therefore every one must be authenticated before testing is started.

It is not expected that several instances of the application will be running in the same factory. Thus, storing of the data in an ordinary file is also possible. But the advantages of a database are unquestionable. Authentication of the users can be done easily using the database.

2.6.1. Database Application Layer: Entity Framework

We have chosen MS SQL Server as a database layer technology. It is easy to integrate and develop in a .NET application. The performance is not the main requirement. Even if SQL database is hosted on a remote computer common for several instances of our application, the load will never be considerably extensive. In the most cases SQL server is running on the same machine as our application.

Entity Framework [15] provides a very effective and comfortable way to access the database. Together with MS SQL Server it is a very powerful tool, which highly simplifies the process of database integration to our application. A part of EF is a code generation tool that creates strongly typed classes directly from database. Collection of instances of these classes are stored in the memory and synchronized with the database on our request. For such a simple database as our application is using, Entity Framework is the right choice.

Business logic of the application sometimes requires more complicated database interaction than only inserting or selecting some piece of data from the database. For example, when the shift is started several information are immediately saved, such as current time, logged in operator, tested mirror, used tests and parameters, etc. In these or similar situations we are using stored procedures which considerably simplify the code on the application side, hide implementation details about the database scheme and assure

data integrity. Entity Framework generates methods that allow us to call these procedures directly from the code.

2.7. Running Testing Sequence

The fundamental feature of the application is the ability of automobile mirror testing. This process can be divided into two independent parts: the execution logic and the visualization. The execution logic is the process of inserting mirror into the tester and running the code that **controls** the tester and **executes** the tests. The visualization part itself is not fundamental as it only displays the current state of execution logic to user.

As an example consider the case when the current of some electrical circuit is measured. The execution logic is the activation of the circuit and regular reading of the current value. Visualization can display the graph of current value over time.

While the execution logic is a time-critical operation, the visualization does not necessarily need to be so. A small delay in visualization is acceptable even if the user notices that.

2.7.1. Scheduling the Testing Sequence

Consider a simplification that tasks are executed sequentially (next one can start only after the previous has been finished). Then scheduling is straightforward by defining an order of tasks and updating them one by one until they are finished.

As described in chapter **2.2.2**, cyclic communication is used. During the execution remote terminals memory is regularly read and written. It is necessary to cut down the latency between each read-write cycle as much as possible to ensure preciseness of the measurement. As an example we mention a test where maximum allowed current is measured. During this test the value of the current is regularly read and compared to a given maximum. **Figure 2.7.1** shows an imprecise measurement caused by a too high latency where the current exceeds the maximum allowed value and returns back between two read operations.

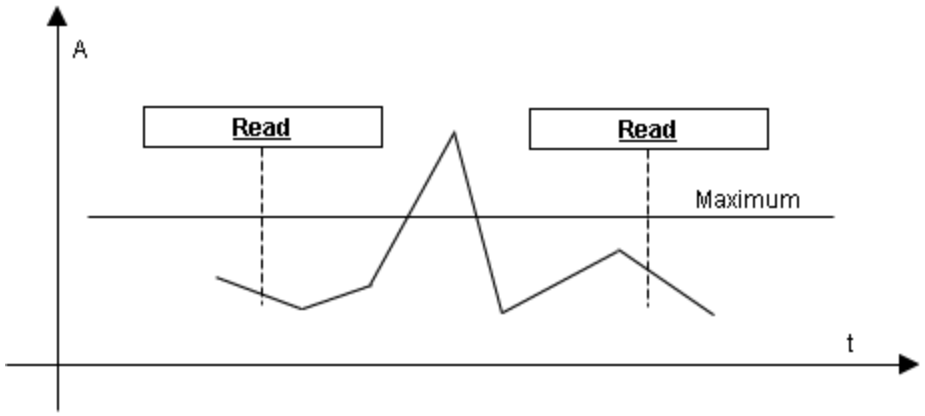


Figure 2.7.1 Imprecise measurement caused by high latency

To avoid such mistakes we would like to run only the test logic during the test sequence execution. Any initialization of tasks, even of those that will never run, can be done before. It will be also convenient to stop garbage collection for a while. At this point it seems that C# (or generally .NET) is not the right platform to use for such an application where time critical operations are performed. This issue is discussed in chapter **2.1**. The experiments show that the delay between two updates is 10-20 milliseconds, which is within the range of customer's acceptable preciseness.

Figure 2.7.2 describes the lifecycle of the test sequence execution. First of all a scheduling is performed – order of tasks is defined. As the scheduling can be a time consuming operation it will be done before any of the tests starts. When all tasks are prepared, they are executed in scheduled order, by regular update[13]. During the update the finished task is switched for the next one in the sequence. After the entire sequence has been executed generated data from tests are collected and saved. This way the CPU resources are reserved only to tasks during the update cycle.

[13] Read, write and logic operations are performed.

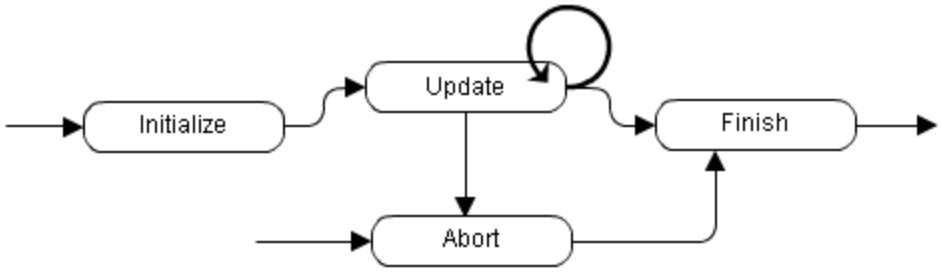


Figure 2.7.2 Lifecycle of test sequence execution

Notice that abortion of the execution can be enforced at any time, by the user or currently executed task if some fatal error occurs. An abort must be handled specially in such a way that no damage on the mirror or testing hardware will be caused. This usually requires additional updates which bring tester into a safe state. Consider a situation when the power-fold test has been aborted. At this moment the actuator moving the mirror must be stopped before test leaves the update loop. Otherwise the actuator may be damaged.

2.7.2. Test Execution Logic: Read/Write operations

The performance of a single test can be divided into two basic phases:

- Preparation phase – bring the tester into a state necessary for execution of a particular test, i.e. close the device with the mirror inserted or move the support with calibrators up to measure mirror glass movement.
- Test phase – execute given test, i.e. measure the spiral current

The preparation phase is not a part of the test, it is a requirement only. Several tests can have the same requirement and if they hold, all of them may start.

Let's denote a simple action, such as set value of a particular channel *a task*. Then the process of testing is a collection of tasks that have to be executed in some order. The entire control logic is abstracted to simple read/write operations. Write is always a control command for tester machine. Read is a check for status whether command was executed or a test of expected values. So there are only a few possible actions to be done during the preparation phase:

- Set value(s) of some channel(s)

- Wait for value(s) on some channel(s)[14]
- Decide which action will be executed next, depending on some channel(s) value(s) – condition
- Sometimes we need an additional operation which does not write or read any channel – wait for specified period of time

With a combination of these tasks we are able to describe all preparation phases. On the other hand the test phase is more complicated. Sometimes we need to check whether measured values are in a given range or whether they do not exceed a defined maximum. Even more complicated is a situation when the measured values must be calculated and depend on additional factors such as some application settings, i.e. mirror rotation angle is calculated from the value of three calibrators and depends on the distances between them.

2.7.3. Task Lifecycle

During the sequence lifecycle several tasks are executed. They are executed cyclically in a loop containing the following steps (see also **Figure 2.2.3**):

- Input channels are loaded to local memory image
- Task logic is updated observing the current channel values and writing outputs (local only)
- Output channels memory image is written to (remote) terminals

All tasks are intended to be created before the loop starts in order not to consume any CPU resources. However, at the time of execution the tasks are brought into the initial state that corresponds to the current sequence state – i.e. starting time is noticed. Afterwards, the task enters the update loop and stays there until it is finished by itself or aborted. At this time it leaves the update loop, but no additional operations are performed until all tasks are finished. The following list describes the task states (see also **Figure 2.7.3**):

- Initialization – proceeded only once just before entering the update loop. Prepares the task for execution and initializes its variables.

[14] We only wait for values on input channels as these do not depend on the application but on the tester.

- Update – proceeded many times in the update loop. Task logic is executed, abortion may be caused.
- Finish – proceeded once when leaving the update loop. Finalize tasks execution and hold the result data until all tasks are finished.
- Abort – the task may be aborted by some external force (i.e. the entire testing is aborted) or by itself (i.e. running too long). In this moment the task is performing aborting operations to bring tester into a safe state.

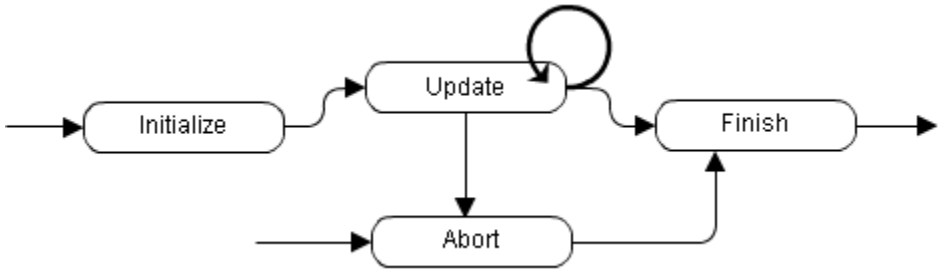


Figure 2.7.3 Task lifecycle

2.7.4. Parallel Task Execution

One update of the task logic is a very short operation. Many times it is simply a comparison of a channel value or a simple calculation. The major part of one update cycle is waiting for remote hardware response. The entire memory image is always transported over the network. Checking which channels have been modified is much more complicated and without any conducive results. Transport of one channel or of all of them takes almost the same period of time. The size of memory image never[15] exceeds one Ethernet frame. Because of that, executing several tasks at once would not have any impact on the communication latency and only a small delay during the execution logic could be noticed. The most important thing is that running time of the testing sequence can be decreased considerably.

Parallelism brings complications to the sequence scheduling. Some tasks are executed sequentially and a strict order is defined (i.e. we must wait until the testing device is closed and only after that we can start the first test) while others can be

[15] The maximum size of the Ethernet frame is 1500 bytes.

executed in parallel. We define a requirement relation[16] < between tasks, which says: if A, B tasks are in relation A < B then task A must be completed before B can be started. Now scheduling of tasks consists of:

1) Building a graph where tasks are the vertices and there is an edge from A to B vertices ⇔ A < B.
2) Topologically sorting the graph.

The advantage of this scheduling is that the order of tasks is defined before the execution and during the update loop no additional operations are performed.

Some tasks do not have to run in the predefined order, but they cannot run at the same time (i.e. mirror movement to two different sides). In this case we require the customer to define the strict order of these tasks, even if it can be considered to be a disadvantage. During topological sorting the order is defined and does not change during the execution. Hence, any other scheduling that dynamically chooses from two equivalent tasks is unnecessarily complicated.

Even if we allow several tasks to be running at a time we strongly refuse the idea of multithreading. Discrete simulation of tasks execution results in a parallelism impression. By synchronizing several threads during communication and execution logic, the application becomes quite complicated and not so stable. Safety aspects should be considered as well, deadlock could in many cases lead to hardware damage. Time of one update cycle is so short that it is questionable whether even parallel threads bring acceleration of the tasks execution.

[16] Anti-reflective and anti-symmetric

3. Implementation

Mirror testing system (MTS) was designed to automate the process of automobile mirrors testing which is a part of the assembly line. In particular, the process includes following steps: create parameters for testing, run specified tests using these parameters and save measured data. During the development we had to take into account several aspects:

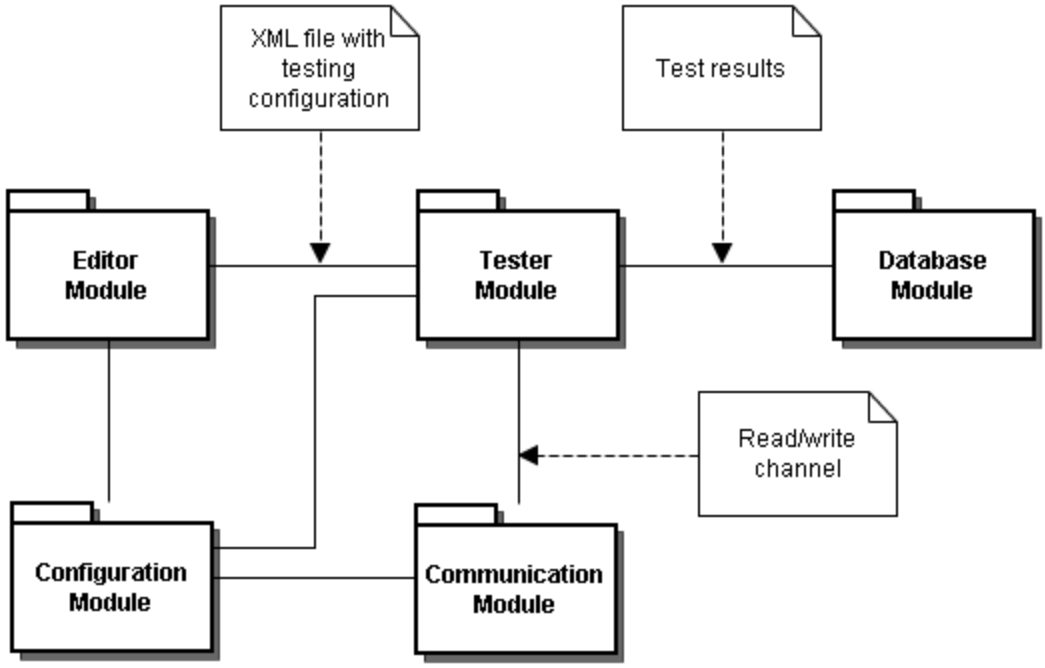
- Communication with the tester machine; this includes selection of a protocol.
- Scalability in defining parameters and saving data.
- Possible configuration of any application setting that may change in the future.

In the following chapters we describe implementation details of issues covered in the **Analysis**.

3.1. Application Structure

The application is divided into following logical modules:

- Communication module – provides safe communication and control of tester machine components.
- Editor of parameters – provides user friendly interface to create or edit a file with testing parameters. Also used when loading parameters from disk to memory.
- Database module – provides access to application data such as results of executed tests or validation of operator login.
- Configuration module – provides access to configuration files and allows user to setup application according current requirements. This leads to a configurable and extendable application which is independent on many settings.
- Tester – allows the user to schedule and execute a collection of tests using defined parameters.



Picture 3.1.1 Application module and dependencies between them

The editor is a general tool for creating testing parameters. All types of tests and parameters are supported. The application configuration defines the structure of the supported file format. The output of the editor is an XML file that is read by the tester. It contains necessary configuration of executing tests. The tester is responsible for creating the tests from given XML configuration, executing them and providing a visualization. Hardware is controlled by writing defined channel values. This ability assures the communication module. The communication module is a general interface to remote terminal units (see **2.2.1**) supporting various protocols. Application configuration defines used channels and their addresses inside the RTU as well as the communication protocol. The test results are passed to database layer based on Entity Framework using CLR type instances mapped to database tables or methods mapped to stored procedures.

3.2. Communication Module

The communication module constitutes the interface between the application and the tester machine. This is the crucial part of our application, because the performance and safety depends on it. Therefore we have dedicated a significant amount of time to design and implement this part.

Our intension is to make the memory of remote terminals accessible to application directly and without any time delay. The principle of operating is similar to computer games. Regularly in a loop the logical step[17] is executed and memory is synchronized with remote terminals (comparable to the process when the screen is redrawn in a game).

3.2.1. Communication Module

One of the possible ways to increase configurability is the independence on the used communication protocol. To implement this feature we decided to make a common interface for all protocols. The application will only access this interface while technical details stay hidden. As described in Chapter **2.2.2**, during the execution loop only three types of operations are needed: read all channel values from terminals, write all output channel values to terminals and access particular channels during the logical phase. These operations includes `IModule` interface (see **Figure 3.2.1**), which provides access to the underlying protocol implementation. During the execution logical phase the memory image is accessed by getting a reference to a particular channel identified by a name.

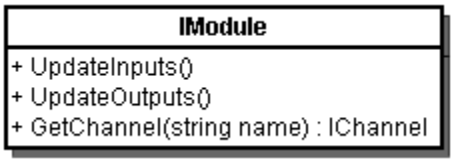


Figure 3.2.1 A common interface for all communication protocols

The list of known channels and their unique identifiers (names) is placed in an external file. It should be defined by the customer[18] and loaded at the time of module initialization. Using channel names is not necessary, but it is more user-friendly to use names instead of numeric addresses. Different protocols use different addressing, but in the application we only use unified identifiers. Also the name gives a meaning to the channel and the configuration becomes more transparent.

The channel value is accessed by various protocols and transferred through the network. There are two types of channels – digital and analog – which differs in the size.

---

[17] The logic of the hardware control consists of processing input channels and writing negotiated output channels

[18] This is a part of the deployment process

In the case of analog channels the size is varying (8, 12, 14, 16, 32 bits). Therefore the common channel interface should be elaborated properly.

The channel data type is not important when transferred over network. It is represented as a byte array which is accessed by a particular protocol implementation as shown in **Figure 3.2.2 -** `IChannel` interface. Clearly it is not very programmer friendly and also does not prevent from writing an input. We strongly distinguish between all types of channels using specialized interfaces. Because the maximum range of an analog channel is 32 bits we have implemented the analog interface using unsigned integer type. It is wastage, but the number of analog channels is very limited[19] and the simplification of our code is noticeable. The used protocol defines how many of bits in the integer are utilized. Output interfaces (**Figure 3.2.2**) only add the ability of writing channel values. This prevents from causing an error by writing to an input channel. An instance of `IChannel` is only a container of remote memory unit. We need to write to it when the value of a remote channel changes. For this purpose the `SetValue` method is defined which is used only internally by the communication module.

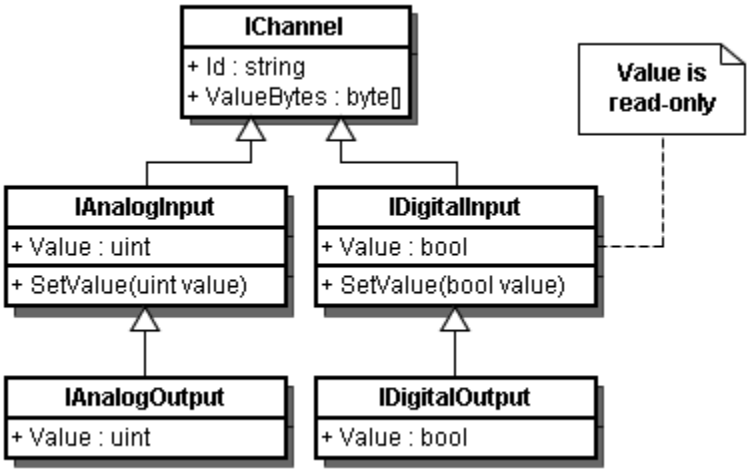


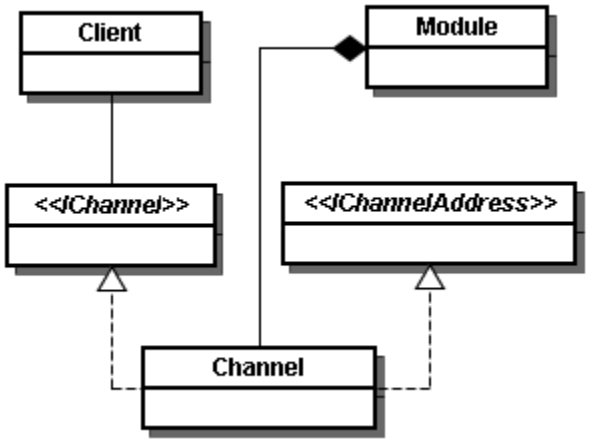
Figure 3.2.2 Channel interfaces

[19] The price of one 12-bit analog terminal is higher than 2GB of DDR2 memory (compared on 15.4.2012).

3.2.2. Addressing Channels

Each protocol uses a proprietary address format. It can be a number pointing to a specific memory cell or a pair of numbers specifying a group of memory cells and a cell inside this group. TwinCAT ADS Library[20] for the EtherCAT protocol supports several ways of channel access. One of them is a numeric group identifier and a channel **name**. We are using this addressing method for the same reason as the DNS is used in the networks.

The address does not need to be accessed from the application; it is only used by the protocol implementation. That is the reason why `IChannel` does not contains any address information. We define a special interface `IChannelAddress<T>` instead, where `T` is the type of the structure holding address information. Then implementation of a channel has two interfaces where `IChannel` is accessed from the application code while `IChannelAddress<T>` is accessed from `IModule` only. The implementation of `IModule` defines the addressing format – the type of `T`.



Picture 3.2.1 Hiding channel address in specific protocol implementation – it is accessed only from `IModule` implementation.

We only define one generic channel type for any protocol. The implementation of `IChannel` interface (and derived interfaces) is fairly large, so we do not need to repeat it.

[20] http://www.beckhoff.com/english.asp?twincat/twincat_ads_communication_library.htm

The implementation of `IModule` for a specific protocol instantiates the right type of this generic class, with specific addressing.

3.2.3. Module Specialization

The implementation of the `IModule` interface is general enough so any application may use it. On the other hand, a channel must be accessed by the name and its existence is checked at the runtime only. When a typing mistake in the name occurs, the intended channel will not be written, and a serious damage on the mirror may be caused. Also channels accessed through the `IModule` interface are not strongly typed and a cast is necessary which leads to more mistakes.

To prevent this threat, we decided to make an adaptation of the `IModule` implementations for requirements of our application – the `Channels` class. With the use of decorator pattern [16] application dependant channels are added as strongly typed properties[21] (see **Figure 3.2.3**). During the initialization phase, all properties are assigned using the reflection and if a channel is not present, an exception is thrown in underlying module, which prevents the communication to be started. This way, the existence of a channel is proved before communication is started and not when firstly accessed. There is also a small optimization as an instance of a channel is accessed directly and not looked up by a string identifier[22]. This is significant during the execution logic phase which needs to run as fast as possible to ensure low latency. The name of a property is always the same as the name of the channel defined in configuration file.

[21] Type of a property is specified according to the channel type: input/output and digital/analog. For each channel type is defined special interface.

[22] Even if `IModule` implemented using Dictionary

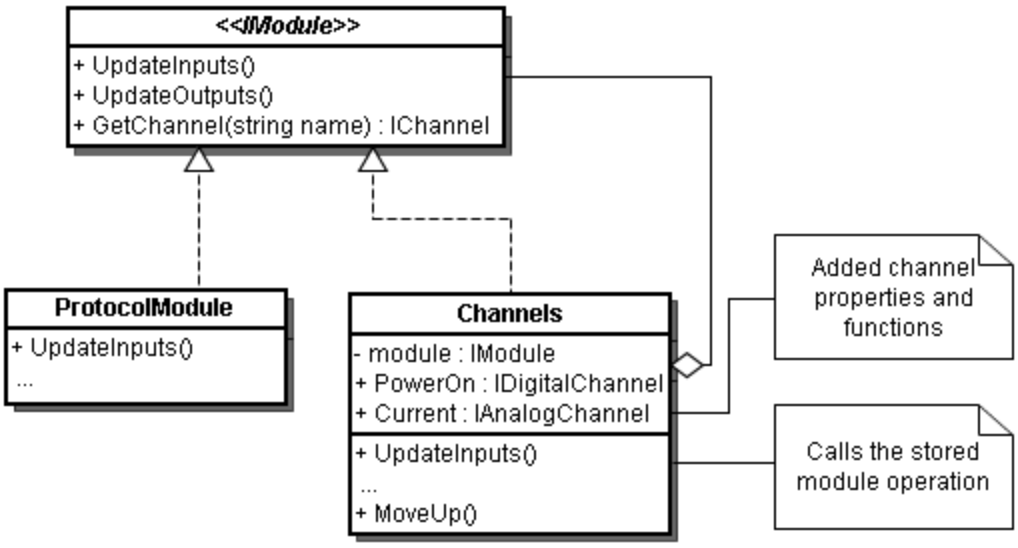


Figure 3.2.3 Extending `IModule` implementation with the decorator pattern (Based on [16] decorator UML diagram)

Operating the tester machine just by writing to channels can be sometimes a little complicated. Some actions require setting up several channels. In this case, an error can occur easily when one of these channels is omitted. Another improvement achieved by the `Channels` decorator is the possibility to add special methods for well-known actions, i.e. moving mirror glass up, which requires several channels to be written to.

Example 3.2.1 Extending module functionality by adding well-known actions

```
public void MoveMirrorUp()
{
   MoveMirrorVertical.Value = false;
   MoveMirrorReverse.Value = true;
   MoveMirrorHorizontal.Value = true;
}
```

3.2.4. Module Lifecycle

Figure 3.2.4 describes a typical module lifecycle. First of all, an instance of empty module is created. When a communication is needed to be established, channels configuration is loaded from an external file – the initialization step. At this time, in the case of the decorator class, the presence of all channels and their types is checked. If any of them is missing, an exception is thrown and the use of this module is blocked.

Otherwise the module is ready for the next step where the physical connection with the remote hardware is established. After that the module is prepared to enter the communication loop and update is called regularly. The connection is closed by the `Disconnect` method, but the module can be reused and connected again until it is disposed. The implementation of `IModule` for a specific protocol is usually done using an external library written as an unmanaged code which requires disposing modules.

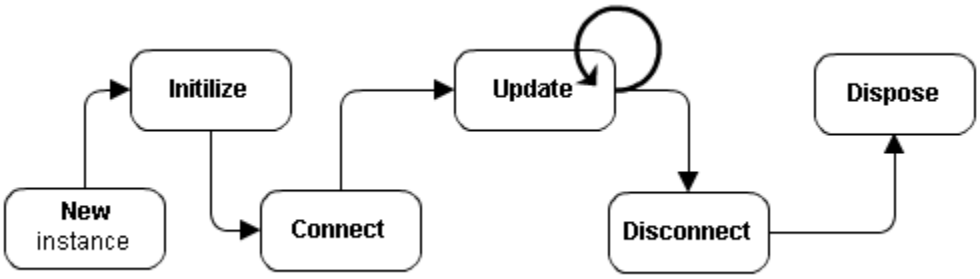


Figure 3.2.4 Module lifecycle

3.2.5. Protocol Types and Management

Three types of protocol are used for communication. However, it is important that any protocol can be integrated to our application by implementing the `IModule` interface. The following list contains supported protocols:

- EtherCAT – industrial protocol developed by Beckhoff[23]
- Modbus – TCP adaptation of Modbus protocol
- Dummy – for testing purpose only. Proprietary protocol type. Useful when debugging complicated configuration.

Each communication module needs some external configuration containing the list of all channels and its properties. In particular: the name, type (input/output), size and address. Loading of this configuration is intended to be done during initialization step explained in **3.2.4**. For the file format we have chosen ".csv" because it can be generated by many third-party tools. For more information see **Appendix B**.

For EtherCAT and Modbus external libraries are used. The principle is always very similar. We use this library to read or write to all channels and distribute their values to

[23] http://beckhoff.com/

`IModule` channels where they are accessible to the testing logic. For more information about particular protocol see attached documentation:

- EtherCAT (5):
- Modbus TCP (4)

3.3. Editor Module

The application is deployed for a specific tester machine supporting a well defined set of tests to be provided. The editor allows the user to configure executing tests. It is important to mention that the deployment process does not require any change to editor code and the set of tests is configurable. Notice that the tester hardware is frequently designed for a specific factory with specific requirements.

The editor is designed to be independent on the rest of the application modules. It also could be transformed into a standalone application easily. Communication with the tester module is based on a XML file with a well-defined structure. To achieve this goal we have defined the test and parameters structure configurable, which allows us to modify tests without any change to the code. Each test can have a different number of parameters of a certain type. The numeric types use various units.

Each test has an ID defined, which is used for internal requirements of the application as a reference when a certain test is needed (i.e. during testing). The same principle is applied to parameters except that their IDs must be unique only inside the test they belong to. Following picture (**Figure 3.3.1**) shows the hierarchy of classes defined for this purpose.

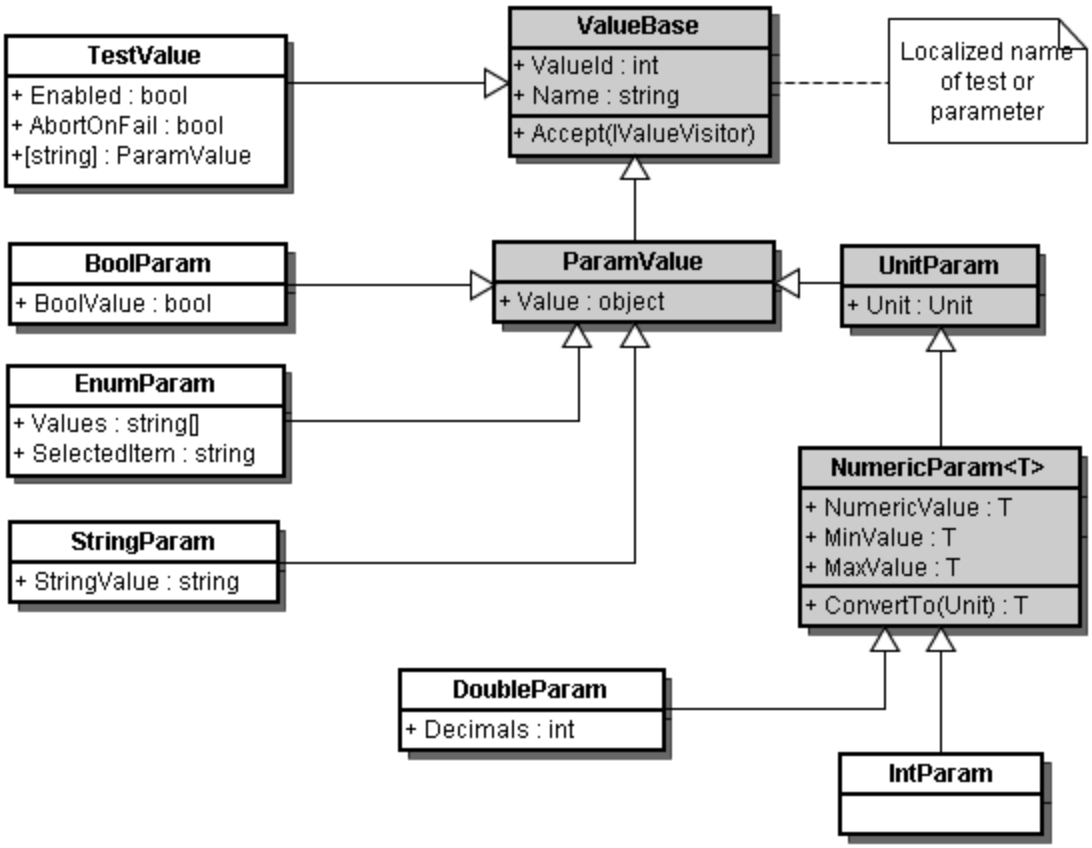


Figure 3.3.1 The class hierarchy of test and parameter implementations

The definition of the tests and their parameters is dragged out to an external configuration file – a so-called template. New files are created according to this configuration. The template is an XML file so we can modify it by using any text editor. The root element contains definition of all tests. A test element contains a collection of all its parameter definitions.

In the template file are defined default values of parameters and tests[24]. This leads to a more comfortable user interface. We can enable all usual and disable all unusual tests, so user must not explicitly set up all of them. When a new file is created default values of parameters are filled automatically in and the user only modifies the necessary ones.

[24] A test also has some specific value like the parameter – it can be enabled / disabled.

Example 3.3.1 Snippet of a template file

This example shows a snippet of a template file defining a test with unique `TravelEast` id and disabled by default. The ID is used only internally by the application while the name and description are localized values and can be displayed in the user interface. The definition of a parameter is very similar except that there are various types of parameter values which require additional settings. In this case, a double parameter of one decimal place with default value 5.0 degrees, minimum possible value 0 and maximum 15 is defined. See also attached sample file (3) for a complete example.

```
<test id="TravelEast" enbled="false">
   <name>Travel East</name>
   <group>Travel Tests</group>
   <description>…</description>
   <param id="MinAngle">
       <name>Min angle</name>
       <description>…</description>
       <type decimals="1">double</type>
       <value>5.0</value>
       <min>0</min>
       <max>15</max>
       <unit type="degrees"/>
   </param>
</test>
```

To improve user experience, similar tests are grouped together. This is achieved by the `<group>` setting in the template.

The classes displayed in **Figure 3.3.1** are used by the editor to hold the test configuration. To make the editor more independent, we use the visitor design pattern [16] in order to extend the functionality; otherwise we would have to go to each class and modify its code. Also, the classes are accessible through user interface and we do not want to open unrelated functionality. As an example, we consider the case when parameters are loaded from the template. They are saved in the string representation and we need to convert them to strongly typed instances. Rather than defining a method `FromString` on `ValueBase` we use `FromStringVisitor` which handles the conversion.

3.3.1. Graphical User Interface

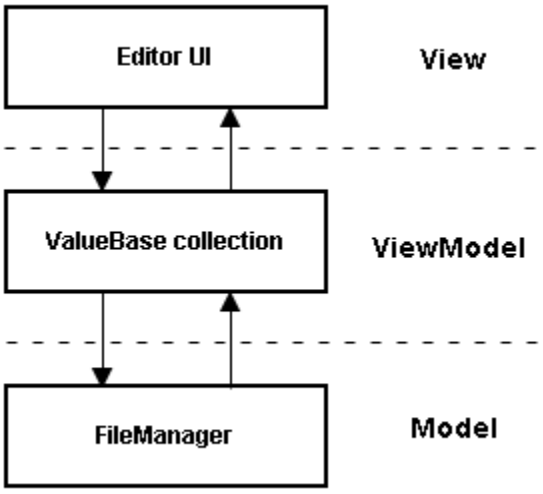
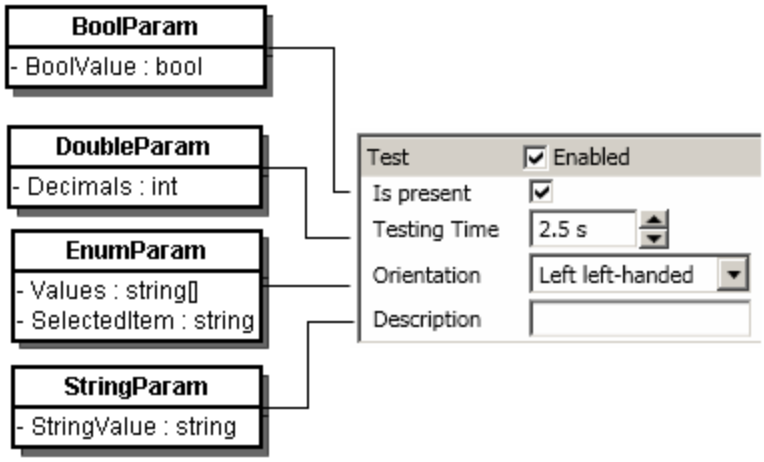


Figure 3.3.2 Model-View-ViewModel pattern used for editor UI

The editor user interface is build using Model-View-ViewModel [17] pattern. As described on **Figure 3.3.2**, the user interface (View) is created from given collection of tests having collection of parameters (ViewModel). These tests are loaded from or saved to a file using the `FileManager` class (Model). This class implements the file format as described in the previous chapter.

A collection of tests is displayed through graphical user interface. Each parameter derivation has strongly typed property for accessing its value. This property is bound to some control of the user interface. With the advantage of data templates [10] in WPF, dynamic GUI is achieved – its structure depends on current template configuration. Following picture shows how different types of parameters are mapped to UI controls according to their type.



Picture 3.3.1 Mapping parameters to user interface using WPF data templates

In the view, only the value of the parameter or test is accessible. Other functionality required by the model is implemented with the use of visitors.

3.3.2. Unit Conversion

Parameters derived from `UnitParam` have a unit defined – have a property of `Unit` type. This property besides describing the attributes of current unit allows conversion from and to the upper and lower unit, i.e. from millimeters to centimeters etc. Before the testing, the parameter value is converted to the base-unit (millimeters for longitude or milliseconds for time) and measurement is performed. After the testing, the result values in the base-unit are converted back to the unit of the parameter. By this way, the customer can choose the unit used in the editor without any impact on the testing logic.

3.4. Database Module

The database module is based on Entity Framework and collection of classes and methods generated from the database using T4 Template [18]. The entry point to the stored data is a `DbContext` derived class which makes accessible database tables as properties and stored procedures as methods.

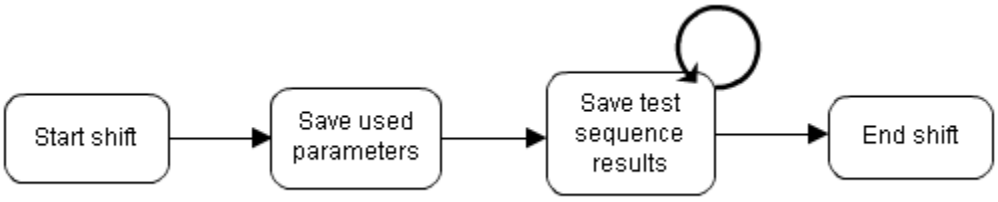
Similarly to the editor module (**3.3**), the database is designed to be independent of the rest of the application. The test types and its parameters are not strictly defined in the database.

Mirror testing requires an identified operator so to open testing window valid login and username must be provided. The authentication data are stored in the database. During testing sequence we want to save to database information about the logged in operator, tested mirror, used tests and used parameters, test results and results of test parameters. For more information see attached database schema.

3.4.1. Running Shift

The application keeps track of the shift execution. The time, mirror type and executing operator are saved to database so that the company may tack employee productivity.

From the point of database, a shift is started with the call of `StartShift` method on `DbContext`. This method returns object representing instance of newly created shift with unique **ID** generated. Used tests and parameters are added to the database (unless it is already created) and associated with this shift. This procedure is handled on the database side by `AddTest` and `AddParam` methods. **IDs** of used tests and parameters are stored on the application side during the entire shift lifetime and referenced when saving associated result at the end of each testing sequence – `AddParamOutput` and `AddTestOutput` methods. The shift is finished by the call of `FinishShift` method.



Picture 3.4.1 Shift lifecycle from the database point of view

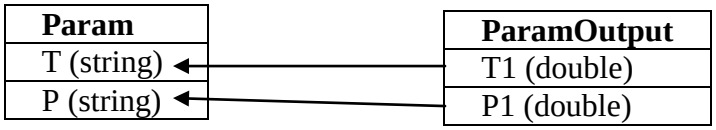
3.4.2. Database Scheme of a Test

The most important database tables are `Test` and `Param` storing used parameters and `TestOutput` and `ParamOutput` storing measured values during the testing. Usually one set of testing parameters is used thousand times repeatedly to tests several mirrors. Therefore `Test` and `Param` tables have unique values where a parameter is added only if it has never been used before. On the other hand, after each test result values are added to `TestOutput` and `ParamOutput` tables with a reference to `Test` and `Param`.

Parameter values in `Param` table are in string representation, because we want to support more data types. All results are numeric values and therefore `ParamOutput` table uses real type for parameter value.

Example 3.4.1 Measuring mirror glass inclination angle

As an example we consider measurement of glass inclination angle. The given parameter P tells how many degrees the mirror should be inclined to satisfy the test. This angle must be reached in time T. At the time T1 > T current angle was P1 < P, so the test is finished. Following values will be saved to the database:



See attached database diagram (2) for detailed information.

3.5. Tester Module

The implementation of the tester module is divided into two main parts: visualization and execution logic. The execution logic is running in a separate thread with real-time priority. Communication module is a shared intermediate layer.

3.5.1. Task Definition

Any simple interaction with tester hardware is denoted as *a task*. Implementation of a specific task must derive from the abstract `Task` class and override its `Update` method. This method is called regularly in a loop. Usual implementation of a task is holding its current state as an enumeration value and `Update` method contains a switch for all possible states. All tasks define at least these states: initializing, finalizing and aborting.

Example 3.5.1 Wait task

Following code snippet shows implementation of `Wait` task that blocks communication with tester for specified period of time (`waitingTime`).

```
public override void Update(DateTime time)
{
   switch (exState)
   {
       case ExState.Initializing:
           StartWatch(time);   // start measuring time
           goTo(ExState.Measuring);
       break;
       case ExState.Measuring:
           if (TimeElapsed(time) > waitingTime)
               goTo(ExState.Finalizing);
       break;
       case ExState.Finalizing:
       case ExState.Aborting:
           Finish(time);       // leave the update loop
       break;
   }
}
```

Finish method defined in the `Task` class leaves the update loop.

We have considered four types of basic tasks that can be performed to control the tester – set value of some channel, wait for some channel to have specific value, wait for specified period of time and make a decision based on some channel value (a condition). Using combination of these tasks we can fully control the hardware. The next step is the implementation of tests.

A mirror test is a specific task, derived from the abstract `TestTask` class. The main difference is that `TestTask` requires an instance of `TestValue` with parameters (**Chyba! Nenalezen zdroj odkazů.**) at the initialization time. Also, it is usually overridden `getResult` method which returns a `TaskResult` instance holding information about task execution and possible results of parameters (for tests). `TestTask` results are saved to the database.

3.5.2. Task Scheduler

The responsibility for executing the given set of tasks is wrapped into the `TaskScheduler` class. The `TaskScheduler` lifecycle is very similar to the task one, as its logic is performed in an update loop. Given set of tasks is contained in four queues – I (input), P (prepared), R (running) and F (finished). At the beginning the I queue holds all

of them. When a task is prepared and there is no restriction for its execution, it is moved from I to P. At the next update, all tasks from P are initialized (call of the `Initialize` method) and moved to R. After that all R tasks are updated. When a task is finished, it is moved to F. This logic is performed in the `Update` method.

The definitions of input tasks are contained in an external XML file. For the four basic tasks and for the test special elements are defined. You can optionally specify a unique id for each element. Also each task element can have a `behavior` child element with `required` attribute which contains list of task ids that must be finished before the current one can start. Well defined requirement relations compose a direct acyclic graph where tasks are the vertices and requirements are the edges. For the details description of input tasks file configuration see the advanced part of attached user manual (1).

Tasks in I are ordered topologically. When preparing new tasks to be run, from I are taken those that do not have any predecessor in P or R.

3.5.3. Visualization

In the testing window are displayed current values of all analog channels over time as a graph. For this purpose `FlowControl` is designed which displays given set of channel values. You can add new value by `AddValue` method which automatically removes the old ones. The proportions of graph are adjusted according to the maximum value in the set.

`FlowControl` is connected to some analog channel through delegate that returns current channel value. The only demand is to call Update method which takes current channel value and adds it to the graph. This is handled by a handler registered in a timer which regularly updates the user interface. The frequency shouldn't be as high as for the communication module. Current timer interval is 400 milliseconds.

Another part updated by the GUI timer is 3D model of the mirror glass. It is encapsulated in the `MirrorView` user control with `RotationAxis` and `RotationAngle` properties. These values are directly passed to WPF, so no graphics calculations are done in our code.

3.6. Simulator

With the MTS application, a software simulator of the tester is also shipped. It can be used for application testing or even by the customer, for debugging complicated configurations before running on the real hardware.

The simulator is listening on given port and waiting for read command[25]. The response is a list of all channels and their values in the following format {channel name}:{value}. MTS application is using the Dummy protocol for communication with the simulator. This protocol is intended to be used for debugging and testing only and is not optimized for speed. The simulator code is also considered to be quickly written testing code.

[25] "read" string

4. Evaluation

The requirement of the automation in the industry led to the creation of the MTS application. This requirement was clearly satisfied by the MTS core providing control logic of the tester machine. The mirror testing is automated completely – only one press of a button is necessary. The performance was improved by the parallelism. Very powerful feature is the possibility of defining the control logic of the machine in an external file.

The design was noticeably influenced by given development conditions – we were not certain about all the tests that will be possibly necessary in the future. Therefore, some of the development decisions were made with future changes in mind. As a positive consequence we consider easy to extend this application with small just a small effort.

Also other aspects necessary for effective usage of the MTS application were implemented: the editor, printing, users, settings etc. From this point of view, we admit that some features may be improved or changed to satisfy the customer demands. For example, the mirror type to be tested can be read from a barcode, the user roles and authorization are very simple and can be improved, the summaries from the database can be exported to excel or other format, etc.

One of the achievements that we strongly appreciate is the secondary product of this software project. It is a communication library for remote terminal units, which can be used standalone. Very nice feature in this library is that it hides hardware details and the communication protocol from the programmer, who can develop the execution logic without knowing what kind of RTU's will be installed. This advantage is clearly visible on the usage of the simulator, which is from the application point of view unrecognizable. The applicability of this library is also confirmed by the fact that it is used by developers in the company MTS s.r.o. for other projects[26].

Although we know about the existence of a related work the implementation of this project is unknown to us. It is expected to be difficult to maintain it and to implement

[26] At the time of writing this thesis for projects under the development

such features as we have, using mentioned technologies. Beckhoff announced the release of TwinCAT 3 which will allow the control of RTU's and PLC computers directly from .NET languages and presumable will replace our communication library. Its goal is for non real-time applications.

5. Conclusion

The goal of this project was to make a very specific application for mirror testing. It requires a special hardware dedicated for this purpose. From this point of view, it seems that our application is intended to be for a single use in some factory adapted for its requirement only. But the development, with the future extensibility in mind, leads us to build a customizable application that can be used for testing almost any kind of mirror for which the testing hardware is adapted.

The usefulness of this application is also proved by the fact that it will be used for mirror testing in the automobile factory. A video recording from the testing can be found on http://code.google.com/p/mirror-testing-system/

Appendix A. CD Content

doc/ – documentation of MTS application, tools documentation

    html/ – generated documentation from source files

install/ – installation package

samples/ – sample files created by MTS application

src/ – source files including database scripts

    config/ – application configuration files

    lib/ – external libraries referenced by MTS

    Setup/ – setup project for the installation package

index.html – generated documentation from source files

readme.txt – read this file

Appendix B. Configuration Files

EtherCAT *.csv configuration file used by our application is generated by TwinCAT System Manager. For more information how to configure a task in TwinCAT System Manger and export it to *.csv file readable by MTS see our video tutorial or Beckhoff documentation.

The format of this file is the following: Name;Type;Size;>Address;In/Out

Name: Channel name given by the customer. Unique identifier used to reference this channel in the application.

Type: Data type of the channel value (BOOL, UINT, INT, SHORT …), indicates the channel size.

>Address: Channel address in the format X.Y, where X is the order of byte where current channel data begins and Y is the order of bit inside X byte where current channel data begins.

In/Out: A constant values: Input or Output indicating whether current channel is an input or an output.

Modbus *.csv configuration file is very similar (it is not generated by any third party tool): [Channel Name];[Slot Number];[Channel Number];[I/O type];[I/O Data Length (bits)];[Comment]

[Channel Name]: Channel name given by the customer. Unique identifier used to reference this channel in the application.

[Slot Number]: Address of a group of channels. This association is given by the hardware slot which contains several channels.

[Channel Number]: Address of the channel inside the slot.

[I/O type]: constant values: Input or Output indicating whether current channel is an input or an output.

[Comment]: Optional user description of current channel.

List of Abbreviations

RTU – Remote Terminal Unit: an electronic device for controlling hardware components. Contains a piece of addressed memory accessible from the network.

XAML – Extensible Application Markup Language: Declarative markup language used for creating .NET Framework UI applications.

XML – Extensible Markup Language: general markup language for encoding documents in a human readable format – stored as a text.

GUI – Graphical User Interface: Application layer interacting directly with the user through control elements. Sometimes referred to as user interface (UI).

WPF – Windows Presentation Foundation: system for creating Windows application with rich graphical user interface.

Attachments

1. User Manual.pdf
2. database.gif – ER model of the database
3. simulation.tc – sample file with testing configuration
4. MXIO.Net_API_v3.pdf – Documentation of a library for TCP Modbus protocol
5. TcAdsNetRef.chm – Documentation of a library for EtherCAT protocol

Bibliography

1. *Microsoft Application Achitecture Guide.* s.l. : Microsoft Corporation, 2009. 9780735627109.
2. C++ performance vs. Java/C#. *Stack Overflow.* [Online] December 9, 2009. [Cited: July 12, 2012.] http://stackoverflow.com/questions/145110/c-performance-vs-java-c.
3. Master/slave (technology). *Wikipedia.* [Online] February 24, 2012. [Cited: March 19, 2012.] http://en.wikipedia.org/wiki/Master/slave_%28technology%29.
4. Remote Terminal Unit. *Wikipedia.* [Online] February 18, 2012. [Cited: March 14, 2012.] http://en.wikipedia.org/wiki/Remote_Terminal_Unit.
5. Programable Logic Controller Operations. *Plctutor.* [Online] [Cited: February 22, 2012.] http://www.plctutor.com/plc-operations.html.
6. EtherCAT - the Ethernet Fieldbus. *EtherCAT.* [Online] [Cited: March 14, 2012.] http://ethercat.org/en/technology.html.
7. **Zezulka, František and Hynčica, Ondřej.** Průmyslový Ethernet IX: EtherNet/IP, EtherCAT. *odbornecasopisy.cz.* [Online] October 2008. [Cited: March 16, 2012.] http://www.odbornecasopisy.cz/index.php?id_document=37910.
8. EtherCAT Performance Analysis. *EtherCAT.* [Online] [Cited: April 23, 2012.] http://www.ethercat.org/en/performance.html.
9. **Nathan, Adam and Lehenbauer, Daniel.** *Windows Presentation Foundation Unleashed.* Indianapolis : Sams Publishing, 2007. ISBN 0-672-32891-7.
10. Customize Data Display with Data Binding and WPF. *MSDN.* [Online] [Cited: February 26, 2012.] http://msdn.microsoft.com/en-us/magazine/cc700358.aspx.
11. WPF Globalization and Localization Overview. *MSDN.* [Online] [Cited: April 1, 2012.] http://msdn.microsoft.com/en-us/library/ms788718.aspx.
12. **Adam, Nathan and Lehenbauer, Daniel.** *Windows Presentation Foundation Unleashed.* Indianapolis : Sams Publishing, 2007. 0-672-32891-7.
13. **Chartier, Robert.** Serialization in the .NET Framework. *Codeguru.* [Online] January 10, 2008. [Cited: April 7, 2012.] http://www.codeguru.com/csharp/.net/net_framework/article.php/c19541/Serialization-in-the-NET-Framework.htm.
14. Document Type Definition. *Wikipedia.* [Online] June 30, 2012. [Cited: July 22, 2012.] http://en.wikipedia.org/wiki/Document_Type_Definition.
15. ADO.NET Entity Framework. *MSDN.* [Online] Microsoft. [Cited: April 23, 2012.] http://msdn.microsoft.com/en-us/data/ef.
16. **Bishop, Judith.** *C# 3.0 Design Patterns.* s.l. : O'Reilly Media, Inc., 2007. 0-596-52773-X.
17. **Smith, Josh.** WPF Apps With The Model-View-ViewModel Design Pattern. *MSDN Magazine.* [Online] February 2009. [Cited: May 15, 2012.] http://msdn.microsoft.com/en-us/magazine/dd419663.aspx.
18. Code Generation and T4 Text Templates. *MSDN.* [Online] March 2011. [Cited: July 22, 2012.] http://msdn.microsoft.com/en-us/library/bb126445.aspx.