**NATIONAL AVIATION UNIVERSITY.**

**INSTITUTE OF INFORMATION DIAGNOSTIC SYSTEMS**

**DEPARTMENT OF AVIATION COMPUTER-INTEGRATED COMPLEXES**



**Term project**

By discipline

Methods of mathematical modelling

Done by:

students of IIDS 533

Illya Tretyakov, Michael Panarin

Kyiv 2017

1. **Introduction**

After creation of wind generator, it’s necessary to measure generator`s parameters, such as rotation speed and the force which is produced by the rotor.

For these aims measuring devices set and software were created.

This paper contains short devices description, software manual and user’s guide.

1. **Devices description**

Measuring devices set contains following elements:

* 1. Tachometer **Uni-t UT372**



**Specification:**

|  |  |
| --- | --- |
| Measurement: | 10 to 99.999 RPM |
| Range selection | 10 to 99 RPM, 100 to 999 RPM, 1000 to 9999 RPM, 10000 to 99999 RPM |
| Accuracy | 0.04% ±2dgt |
| Display | 99999 |
| Target distance | 50 mm to 200 mm |
| Start/Hold: | Start and Stop measurement, Data Hold |
| Max./min./ave/zero: | Maximum value, minimum value, average value, zeroing |
| Sleep Mode | Around 15 minutes |
| Low battery display | 4.8 V |
| USB interface | Yes |

**General characteristics:**

|  |  |
| --- | --- |
| Power: | 1.5V batteries (AA) x 4 |
| LCD size | 54 x 41 mm |
| Product size | 182 x 56 x 34 mm |
| Product weight | 100 g |

* 1. Force transducer **HBM C9C**



**Description:**

The compact and cost-effective C9C type series force sensor reliably measures compressive force where space is a constraint. The force transducer allows for both dynamic and static measurement tasks to be solved. Due to the high fundamental frequency, C9C is also suitable for very fast measurements. Its welded design from stainless steel makes the force sensor extremely robust.

* 1. Indicator panel for pressure sensor **HBM WE2107**

WE2107 weighing indicators are an amplifier for connection to commercially available strain gage load cells or scales. The load cell signal is amplified and digitally converted; all further processing steps then follow digitally in a microprocessor.

The WE2107 has available:

* load cell connection: up to six load cells of 350 or minimum 58 loading
* one COM port for serial communication with a PC / PLC (RS232 or RS485)
* one COM port for a printer or an external large-scale display (RS232)
* 2 digital control inputs
* 4 digital outputs (limit value switches or filling / dosing control)
* analog output (4...20 mA)
* two function keys (user-defined)

The electronics are set and parameterized via keyboard on PC.

**Additional features:**

* accuracy in legal-for-trade applications up to 6000e (0.8 µV/e)
* usable as a one, two or three-range scale
* menu functions can be disabled / enabled
* filter selection
* maximum capacity adjustment, partial capacity adjustment, mV/V calibration
* zero on startup
* automatic zero tracking
* weighing range linearization
* 4 limit value switches with hysteresis
* alibi memory for adjustment parameters and weighing results
* different print functions with summation memory
* numerous monitoring and error detection functions

The block scheme which shows the interconnections between functional blocks of the following system is shown on fig.1.

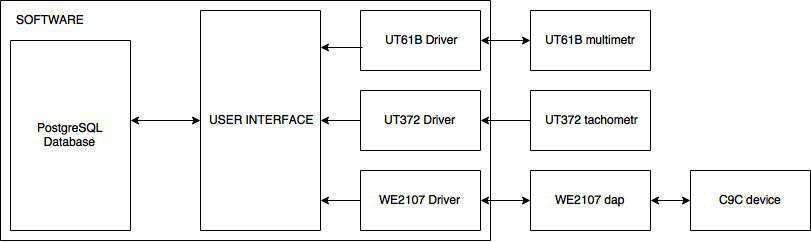


Fig.1. Block scheme of devices set.

1. **SOFTWARE IMPLEMENTATION**
   1. **USED SOFTWARE**

This software was created in Python 3.5.2 language with additional libraries: pywinusb, pyodbc, pyserial.

* + 1. **PYTHON 3.5.2**

**Python** is a widely used high-level programming language used for general-purpose programming, created by Guido van Rossum and first released in 1991. An interpreted language, Python has a design philosophy which emphasizes code readability (notably using whitespace indentation to delimit code blocks rather than curly braces or keywords), and a syntax which allows programmers to express concepts in fewer lines of code than possible in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale.

Python features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

Python is a multi-paradigm programming language: object-oriented programming and structured programming are fully supported, and many language features support functional programming and aspect-oriented programming (including by metaprogramming and metaobjects (magic methods)). Many other paradigms are supported via extensions, including design by contract and logic programming.

Python uses dynamic typing and a mix of reference counting and a cycle-detecting garbage collector for memory management. An important feature of Python is dynamic name resolution (late binding), which binds method and variable names during program execution.

**Why python?  
Readability**

Python very closely resembles the English language, using words like ‘not’ and ‘in’. This is also helped by Python’s very strict punctuation rules which means you don’t have curly braces ({ }) all over your code.

Also, Python has a set of rules, known as PEP 8, that tell every Python developer how to format their code. This means you always know where to put new lines and, more importantly, that pretty much every other Python script you pick up, whether it was written by a novice or a seasoned professional, will look very similar and be just as easy to read.

**Libraries**

Python has been around for over 20 years, so a lot of code written in Python has built up over the decades and, being an open source language, a lot of this has been released for others to use. You can install this software on your system to be used by your own projects. There are libraries for pretty much any use case you can come up with, from image manipulation, to scientific calculations, to server automation.

* + 1. **pywinusb**

**Introduction**

This library is a simple USB/HID user application space (hence no drivers needed) 100% python package.

**Advantages**

* All python code, using ctypes
* Top level handling of HID events (usage events calling hook handlers)

**Limitations**

Windows only (so far...)

* + 1. **pyodbc**

**pyodbc** is an open source Python module that makes accessing ODBC databases simple. It implements the DB API 2.0 specification but is packed with even more Pythonic convenience.

* + 1. **pyserial**

**Overview**

This module encapsulates the access for the serial port. It provides backends for Python running on Windows, OSX, Linux, BSD (possibly any POSIX compliant system) and IronPython. The module named “serial” automatically selects the appropriate backend.

It is released under a free software license.

**Features**

* Same class based interface on all supported platforms.
* Access to the port settings through Python properties.
* Support for different byte sizes, stop bits, parity and flow control with RTS/CTS and/or Xon/Xoff.
* Working with or without receive timeout.
* File like API with “read” and “write” (“readline” etc. also supported).
* The files in this package are 100% pure Python.
* The port is set up for binary transmission. No NULL byte stripping, CR-LF translation etc. (which are many times enabled for POSIX.) This makes this module universally useful.
* Compatible with I/O library

**4.2 MODULES**

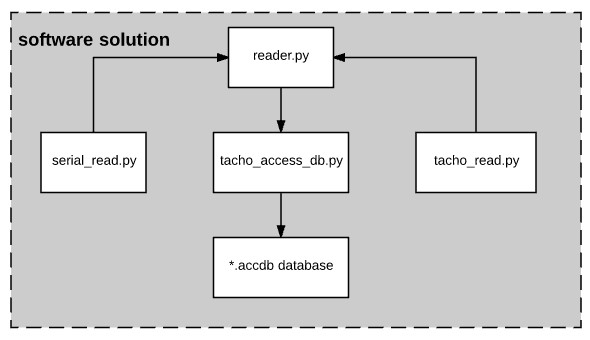


Fig.2. Software block scheme

All modules can be used as standalone or completely in another program

There is a main script “reader.py” that calls other modules.

**4.2.1 reader.py**

Main executable script. Calls to other modules to fish for functions. Has only the \_\_main\_\_ body that is executed in infinite loop. All the device availability and connection checks are built in the specific modules of that device.

**4.2.2 serial\_read.py**

Module created for connection to “we2107” device and read data from it. Requires pySerial library in order to work

**class we2107** – small class for device objects

**we2107.\_\_init\_\_** – constructor with required parameters

**we2107.read\_data** – function, that returns serial data from specific COM port as string

**we2107.\_\_del\_\_** – destructor of class to make sure that the connection is properly closed

**4.2.3 tacho\_acess\_db.py**

Module for inserting data into Microsoft Access database. Built around pyodbc library and requires it to work

**class AccessConnect** – main class for data manipulations in DB

**AccessConnect.\_\_init\_\_** – constructor with required parameters

**AccessConnect.read\_record(sdate, devices)** – function that read a records for a specific date and time for a specific devices

*sdate –* datetime object. Date of the required data

*devices –* iteratable object of strings(list, set, dict, tuple, etc.). List of the devices

*returns* list of tuples with data from DB

**AccessConnect.add\_record(sdate, device, value, data\_type)** – function that adds a record for a specific date and time for a specific device

*sdate* – datetime object. Date for the record

*device* – string object. Device name

*value* – float object. Data from device

*data\_type* – string object. Dimension of the value

*return* None

**4.2.4 tacho\_read**

**class DeviceIsNotConnected(Exception)**- exception derived class. Meant to be raised when device is disconnected. Has no functions

**class ut372device** – class for tacho device objects

**ut372device.\_\_init\_\_** – constructor with required parameters

**ut372device. sample\_handler\_count (data)** – legit data collector. Discards garbage and collects a full package into a global list

*data –* data object. Data from device.

*return* None

**ut372device. \_positioning\_count** – private functionthat that parses data and forms output

**ut372device.connect –** Function that connects to a device

**ut372device.** **\_send\_init\_packet –** private function that send initialization package to a device

**ut372device.** **receive\_package –** Function that waits for the package received event and handles device disconnection

**5. USER GUIDE**

**5.1 Setting up devices**

First of all, it’s necessary to set up the devices.

For UNI-T UT372:

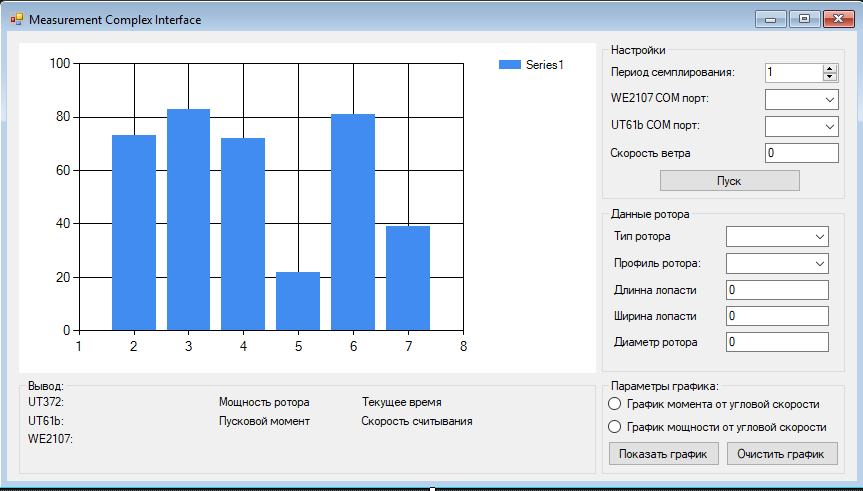
1. Turn on the device.
2. Long press **R/C** button, until setup menu will appear.
3. Turn USB mode on by pressing **M/M/A** button. The result should look like “USB 1” in the following menu.
4. Turn back to the measurement mode by pressing **ON/OFF** button.

For HBM WE2107:

1. Ensure that load cell (C9C) is connected to the device.
2. Connect WE2107 via COM port (or use COM-to-USB converter cable).
3. Check up the device settings:
   1. Press ENTER and UP buttons simultaneously to call menu.
   2. Press NEXT button 3 times for **UART1** to appear on the screen, press ENTER.
   3. Addr is not necessary to set up, press NEXT and ENTER when bAudr appears. For correct working of the software it’s necessary to set baud rate to 2400 by clicking NEXT button.
   4. After setting the baud rate press CE to set PAIRITY which should be EVEN. Go back using CE button.
   5. Press CE button 2 times to exit settings menu.

**5.2 Using executable file**

**MeasureComplexInterface.exe –** executable file which launches the interface software in project folder.



Execution process:

1. Open the program, set up the parameters in Settings block.
2. Set up parameters in Rotor Data block.
3. Optionally you can select between radio buttons which allows you to see the chart for chosen values.
4. Outputs are shown inside Output block.

Data is writing then in the database.

**5.3 Using Sources**

**5.3.1 Python installation**

Before you start, you will need Python on your computer, but you may not need to download it.

First of all check that you don't already have Python installed by entering **python** in a command line window. If you see a response from a Python interpreter it will include a version number in its initial display. Generally any recent version will do, as Python makes every attempt to maintain backwards compatibility.

**For Windows**: the most stable Windows downloads are available from the Python for Windows page.

**5.3.2 pywinusb installation**

The most convenient way of installing is using easy\_install or pip

Run “pip install pywinusb” command in command line

**5.3.3 pySerial installation**

pySerial can be installed from PyPI using:

pip install pyserial

or:

easy\_install -U pyserial

in command line

**5.3.4 pyodbc installation**

To install, use pip or setuptools, run

pip install pyodbc

in command line

**5.3.5 Getting software sources**

Cloning the github repository is a valid option: <https://github.com/PanarinM/com_slash_usb> to get sources for further usage.

**5.3.6 Starting the program**

* Navigate to the sources folder via “CD” command in command line
* Connect devices
* Run “python reader.py” command to start the program
* Follow instructions from 5.1.2

**6. Code listing**

**6.1 reader.py**

*from serial\_read import we2107*

*from pyodbc import Error*

*from tacho\_read import ut372device, DeviceIsNotConnected*

*from tacho\_acess\_db import AccessConnect*

*import serial.tools.list\_ports*

*from serial import SerialException*

*from os import listdir, path*

*from sys import exit*

*def \_\_db\_choose():*

*"""*

*Lists \*.accdb Access database files in the current folder. Then asks user to choose one*

*:return: String with path to the DB*

*"""*

*counter = 0*

*dbfiles = []*

*for file in listdir('.'):*

*if file.endswith('.accdb'):*

*counter += 1*

*print('{}. {}'.format(counter, file))*

*dbfiles.append(file)*

*if counter == 0:*

*print('No Accsess DB files found')*

*input()*

*exit()*

*while True:*

*userinput = int(input('Choose the DB file: '))*

*if userinput > counter or userinput < 0:*

*print('Wrong input!')*

*continue*

*else:*

*return path.realpath('.\\{}'.format(dbfiles[userinput-1]))*

*def \_\_comchoose():*

*"""*

*Lists availible COM ports. Then asks user to choose one*

*:return: String with com port*

*"""*

*comlist = serial.tools.list\_ports.comports()*

*for com in comlist:*

*print('{}. {} {}'.format(comlist.index(com) + 1, com.device, com.description))*

*if len(comlist) == 0:*

*print('No COM devices found')*

*input()*

*exit()*

*while True:*

*userinp = int(input('Input COM number: '))*

*if userinp > len(comlist) or userinp < 0:*

*print('Wrong input')*

*exit()*

*else:*

*return comlist[userinp-1].device*

*if \_\_name\_\_ == "\_\_main\_\_":*

*comnumb = str(\_\_comchoose())*

*pathtodb = \_\_db\_choose()*

*try:*

*db = AccessConnect(pathtodb, 'table')*

*except Error:*

*print('Error occured during the database connection. Check if Microsoft Access Driver is installed and availible')*

*input()*

*exit()*

*try:*

*pressure = we2107(comnumb)*

*except SerialException:*

*print('Wrong COM port for pressure device')*

*input()*

*exit()*

*try:*

*tacho = ut372device()*

*print('Tachometer device was found by product ID = {} and vendor ID = {}'.format(tacho.product\_id,*

*tacho.vendor\_id))*

*except DeviceIsNotConnected:*

*print('Tachometer device was not found by specified product ID and vendor ID')*

*input()*

*exit()*

*print('------------------------------------------')*

*while True:*

*tacho\_data = tacho.receive\_package()*

*if tacho\_data[2] is None or tacho\_data[2] == 0:*

*continue*

*db.add\_record(tacho\_data[2], 'ut372', tacho\_data[1], tacho\_data[0])*

*c9c\_data = pressure.read\_data() if pressure.read\_data() < 3500 else print('corrupt packet')*

*if c9c\_data is not None:*

*db.add\_record(tacho\_data[2], 'c9c', c9c\_data, 'gram')*

*elif c9c\_data is None:*

*c9c\_data = 0*

*db.add\_record(tacho\_data[2], 'c9c', c9c\_data, 'gram')*

*output = """\rut372: {} {} | c9c: {} gram | time: {} """.format(tacho\_data[1], tacho\_data[0], c9c\_data, tacho\_data[2])*

*print(output, end='')*

**6.2 serial\_read.py**

*from serial import Serial, PARITY\_EVEN, SerialException*

*class we2107(object):*

*"""*

*class for pySerial c9c device, connected through we2107*

*"""*

*def \_\_init\_\_(self, comnumb):*

*self.ser = Serial()*

*self.ser.baudrate = 2400*

*self.ser.port = comnumb*

*self.ser.parity = PARITY\_EVEN*

*self.ser.stopbits = 1*

*self.ser.open()*

*def read\_data(self):*

*"""*

*Sends serial command to device and then reads the output data*

*:return: String with output data*

*"""*

*self.ser.write(b';S15;')*

*self.ser.write(b'MSV?;')*

*data = self.ser.readline()*

*return data[0]+data[1]\*255+data[2]*

*def \_\_del\_\_(self):*

*"""*

*convenient close of serial port*

*:return: None*

*"""*

*self.ser.close()*

*if \_\_name\_\_ == '\_\_main\_\_':*

*comnumb = input('Input COM port number: ')*

*try:*

*device = we2107(comnumb)*

*except SerialException:*

*print('Wrong COM port')*

*exit()*

*while True:*

*print(device.read\_data())*

**6.3 tacho\_access\_db.py**

*import pyodbc*

*from pyodbc import Error*

*class AccessConnect:*

*"""*

*main class for data manipulations in DB*

*"""*

*def \_\_init\_\_(self, database=r'C:\Users\Public\Documents\database.accdb', table=r'table'):*

*self.database = database*

*self.table = table*

*self.connstr = r'DRIVER={Microsoft Access Driver (\*.mdb, \*.accdb)};DBQ=' + self.database*

*self.cnxn = pyodbc.connect(self.connstr, autocommit=True)*

*self.cur = self.cnxn.cursor()*

*def \_\_del\_\_(self):*

*try:*

*self.cnxn.close()*

*except Error:*

*pass*

*def read\_record(self, sdate, devices):*

*"""*

*function that read a records for a specific date and time for a specific devices*

*:param sdate: datetime object. Date of the required data*

*:param devices: iteratable object of strings(list, set, dict, tuple, etc.). List of the devices*

*:return: list of tuples with data from DB*

*"""*

*strsql = "SELECT dev\_name,dev\_state,dev\_date FROM [{table}] where dev\_date > #{date}# " \*

*"and dev\_name IN ({device})".format(table=self.table, date=sdate, device=','.join(["'"+j+"'" for j in devices]))*

*self.cur.execute(strsql)*

*t = list(self.cur)*

*return t*

*def add\_record(self, sdate, device, value, data\_type):*

*"""*

*function that adds a record for a specific date and time for a specific device*

*:param sdate: datetime object. Date for the record*

*:param device: string object. Device name*

*:param value: float object. Data from devic*

*:param data\_type: string object. Dimension of the value*

*:return: None*

*"""*

*strsql = "insert into [{table}] (dev\_name, dev\_state, dev\_date, dev\_data\_type) values('{name}', {state}, " \*

*"#{sdate}#, '{data\_type}')".format(table=self.table, name=device, state=value, sdate=sdate, data\_type=data\_type)*

*self.cnxn.execute(strsql)*

*self.cnxn.commit()*

**6.4 tacho\_read.py**

*from time import sleep*

*import pywinusb.hid as hid*

*from datetime import datetime*

*class DeviceIsNotConnected(Exception):*

*"""*

*exception derived class. Meant to be raised when device is disconnected.*

*"""*

*pass*

*class ut372device(object):*

*"""*

*class for tacho ut372 device objects*

*"""*

*def \_\_init\_\_(self, vendor\_id=0x1a86, product\_id=0xe008):*

*self.vendor\_id = vendor\_id*

*self.product\_id = product\_id*

*self.lst = []*

*self.data = 0, 0, 0*

*self.device = 'will further be overriden with device object'*

*self.marker = False*

*self.descriptor = {*

*'7;': '0',*

*'60': '1',*

*'5>': '2',*

*'7<': '3',*

*'65': '4',*

*'3=': '5',*

*'3?': '6',*

*'70': '7',*

*'7?': '8',*

*'7=': '9',*

*'?;': '0.',*

*'>0': '1.',*

*'=>': '2.',*

*'?<': '3.',*

*'>5': '4.',*

*';=': '5.',*

*';?': '6.',*

*'?0': '7.',*

*'??': '8.',*

*'?=': '9.',*

*'00': '00',*

*'78': 'conf',*

*'86': 'bit',*

*}*

*self.connect()*

*def sample\_handler\_count(self, data):*

*"""*

*Device data handler. Discards garbage and collects a full package into a global list*

*:param data: Data object. Data from device.*

*:return: None*

*"""*

*if not chr(data[2]) == '\x00':*

*if self.marker:*

*self.lst.append(chr(data[2]))*

*if len(self.lst) == 27:*

*self.data = self.\_positioning\_count()*

*if chr(data[2]) == '\n':*

*self.marker = True*

*def \_positioning\_count(self):*

*"""*

*Private function that parses data and forms output*

*:return: String of output data*

*"""*

*deciphered\_raw = []*

*for i in range(1, len(self.lst) - 6, 2):*

*try:*

*deciphered\_raw.append(self.descriptor[self.lst[i] + self.lst[i + 1]])*

*except KeyError:*

*self.lst.clear()*

*self.marker = False*

*return*

*count = deciphered\_raw[:5]*

*try:*

*count = int(''.join(list(reversed(count))))*

*mes\_unit = 'count'*

*except ValueError:*

*count = float(''.join(list(reversed(count))))*

*mes\_unit = 'rpm'*

*time = datetime.now()*

*time = time.strftime('%d-%m-%Y %H:%M:%S')*

*output = mes\_unit, count, time*

*self.lst.clear()*

*return output*

*def connect(self):*

*"""*

*Function that connects to a device*

*:return: None*

*"""*

*devfilter = hid.HidDeviceFilter(vendor\_id=self.vendor\_id, product\_id=self.product\_id)*

*hid\_device = devfilter.get\_devices()*

*try:*

*self.device = hid\_device[0]*

*except IndexError:*

*raise DeviceIsNotConnected('Device is not connected!')*

*self.device.open()*

*self.device.set\_raw\_data\_handler(self.sample\_handler\_count)*

*self.\_send\_init\_packet()*

*def \_send\_init\_packet(self):*

*"""*

*Private function that send initialization package to a device*

*:return: None*

*"""*

*report = self.device.find\_feature\_reports()*

*buffer = [0x00] \* 6*

*buffer[0] = 0x00*

*buffer[1] = 0x60*

*buffer[2] = 0x09*

*buffer[5] = 0x03*

*report[0].set\_raw\_data(buffer)*

*report[0].send()*

*def receive\_package(self):*

*"""*

*Function that waits for the package received event and handles device disconnection*

*:return: None*

*"""*

*if self.device.is\_plugged():*

*sleep(0.02)*

*return self.data*

*else:*

*raise DeviceIsNotConnected('Device was unplugged')*

*if \_\_name\_\_ == '\_\_main\_\_':*

*try:*

*ourdevice = ut372device()*

*except DeviceIsNotConnected:*

*print('Device is not connected!')*

*print(ut372device.receive\_package())*