

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Grete Ohak 185092IABB

Hardware and software setup for long term OCXO output frequency and temperature measurement

Bachelor's thesis

Supervisor: Erkki Arus
MSc EE

Marten Kask
MSc

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Grete Ohak 185092IABB

Riist- ja tarkvaralahendus OCXO sageduse ja temperatuuri pikaajaliseks mõõtmiseks

Bakalaureusetöö

Juhendaja: Erkki Arus
MSc EE

Marten Kask
MSc

Tallinn 2021

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Grete Ohak

17.05.2021

Abstract

In this thesis a measurement system is designed to measure a 10 MHz OCXO output frequency and temperature on a telecom product continuously for an extended period of time, up to 24 hours, to observe that there are no sudden frequency deviations. Specific hardware setup was created and software written to carry out the measurements and store results. The measured frequency and temperature values, together with a timestamp of each measurement, are stored in a comma-separated log file. The designed measurement system is verified functional and able to meet the thesis problem requirements.

This thesis is written in English and is 37 pages long, including 18 chapters and 17 figures.

Annotatsioon

Riist- ja tarkvaralahendus OCXO sageduse ja temperatuuri pikaajaliseks mõõtmiseks

Selle töö eesmärgiks on disainida sageduse ja temperatuuri mõõtmise süsteem, mis mõõdaks 10 MHz OCXO väljundsagedust 24 tunni jooksul. Süsteemi abil saab kindlaks teha, et ei esine märkimisväärsed kõrvalkaldeid 10MHz-st. Selle jaoks pannakse kokku riistvaraline lahendus ning luuakse programmikood mõõtmistulemuste kätte saamiseks ning salvestamiseks. Sageduse mõõtmistulemused ei tohiks olla mõjutatud temperatuurist, seega tuleb mõõta ka OCXO lähedal asuva komponendi temperatuuri ning tulemused samuti samas failis salvestada. Esialgu saab tulemusi analüüsida manuaalselt, et teada saada, kas loodud lahendus töötab ja väljundsagedus püsib ette antud limiitide piires. Kirja on pandud ka võimalikud edasiarendused tulevikuks.

Põhjus selle süsteemi loomiseks on kliendi täheldus telekommunikatsioonitoote piiratud funktsionaalsusest, mis võib olla seotud OCXO väljundsageduse võimaliku kõikumisega.

Loodud mõõtmise süsteem on testitud, täidab oma eesmärgi ning seda on võimalik kasutada antud lõputöös käsitletud probleemi lahendamiseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 37 leheküljel, 18 peatükki ja 17 joonist.

List of abbreviations and terms

COM	Communication Port
CSV	Comma-Separated Values
DBB	Debug Board
DTP	Device Test Platform
DUT	Device Under Test
IVI	Interchangeable Virtual Instruments
LMT	Local Maintenance Terminal
OCXO	Oven Controlled Oscillator
PC	Personal Computer
PCB	Printed Circuit Board
PPB	Parts Per Billion
PTC	Production Test Connector
RS232	Recommended Standard 232 for serial communication
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

Table of contents

Author’s declaration of originality	3
Abstract.....	4
Annotatsioon.....	5
List of abbreviations and terms	6
Table of contents	7
List of figures	8
1 Introduction	9
2 Background.....	10
2.1 Product.....	10
2.2 OCXO.....	10
2.3 Hardware setup	11
2.4 Software setup	13
3 Frequency measurement.....	15
3.1 Measurement accuracy	15
3.2 Single measurement.....	16
3.3 Continuous measurement	20
4 Temperature measurement	24
4.1 Measurement accuracy	24
4.2 Single measurement.....	24
4.3 Continuous measurement	26
5 Merging measurement data	28
6 Analysis	30
7 Possible future developments	34
8 Summary.....	35
References	36
Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis	37

List of figures

Figure 1. OCXO [2].....	10
Figure 2. Hardware setup.....	12
Figure 3. Device Test Platform.....	13
Figure 4. Driver initialization and communication	17
Figure 5. Collecting frequency measurements	18
Figure 6. Adding measurement results to the list and print out.....	18
Figure 7. Parallel running threads	20
Figure 8. Initializing with IVI driver	22
Figure 9. Getting measurements and checking if they are valid	23
Figure 10. Command output.....	25
Figure 11. Getting temperature with serial port	26
Figure 12. Getting temperature using COM port and dictionary	27
Figure 13. Merging frequency and temperature into CSV file.....	29
Figure 14. Data in CSV file.....	30
Figure 15. Product warm-up stage.....	31
Figure 16. Frequency and temperature change over 24h	31
Figure 17. Frequency change over 24h with upper and lower limit.....	32

1 Introduction

Most electronics today use crystal oscillators, mainly for microcontroller and processor clock frequency sources. A crystal oscillator generated output frequency depends on its temperature. In-order to get a table output frequency, Oven Controlled Oscillators (OCXO) are widely used, which automatically stabilize their crystal temperature [1]. However, even stable OCXOs can have output frequency stability issues in certain scenarios.

The problem of this thesis is that a customer observed performance issues (losing sync lock) with a telecom product which could be related to OCXO output frequency stability.

The purpose of the thesis is to design a measurement system with which to measure a 10 MHz OCXO output frequency and temperature on a telecom product continuously for an extended period of time, up to 24 hours and to observe that there are no sudden deviations. For this a suitable hardware setup should be created and specific software written to carry out the measurements and store results. The measured frequency and temperature values, together with a timestamp of each measurement, will be stored in a comma-separated log file. The results of the measurement shall be manually analysed to see if the designed solution works for the intended purpose and can be used to detect sudden OCXO output frequency deviations that exceed the required limits. The thesis also includes recommendations for possible future developments. Similar works probably have been done, but they are not available.

2 Background

2.1 Product

The product under investigation is a telecom product that features a 10 MHz OCXO for generating an internal clock reference. The reference 10 MHz clock is used to generate custom clock frequencies, for Transport Network and Radio Base Station Network, on the product. During production testing, it is mandatory to measure the 10 MHz OCXO output clock, and for this reason the clock output signal is routed to Production Test Connector (PTC). This is where the clock will be measured from, for the purposes of this thesis.

2.2 OCXO

The Oven Controlled Crystal Oscillator (OCXO) is a stable frequency generation device in electronics products [2]. The used quartz crystals have a natural output frequency dependence to its temperature. However, the temperature inside the product will fluctuate during its use which can cause frequency shifts. To compensate for temperature changes and keep the crystal close to constant temperature during use, and thus keep the output frequency stable, an internal heater together with its control circuit are used [3]. The OCXO components are housed in a hermetically sealed metal case, that also acts as an Electro Magnetic shielding. The OCXO is shown in Figure 1.

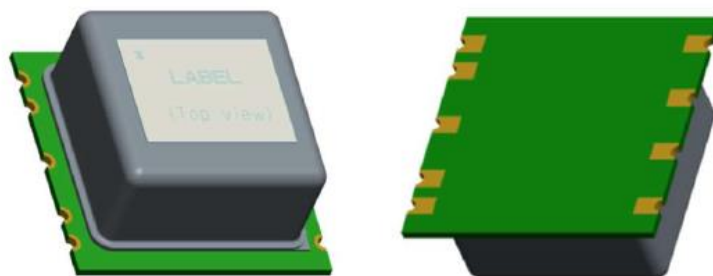


Figure 1. OCXO [3]

The OCXO component used in the product can be from multiple manufacturers and historically quality issues have been documented internally [4], which could possibly affect the output frequency stability of the component. In addition, there could be other issues that could affect the output frequency stability, like the ageing related frequency jumps that are described here [5]. The previously recorded issues, possible unknowns and the nature of the failure performance at the customer site are the main reasons for needing to verify the OCXO component.

2.3 Hardware setup

In order to measure the OCXO frequency stability, the hardware setup as depicted in Figure 2. Hardware setup Figure 2 was prepared. The setup can be described as in two parts, the product communication interface and the 10 MHz OCXO output signal interface.

The product communication interface hardware setup consists of a USB-RS232 adapter that connects to product Local Maintenance Terminal (LMT) port. The USB-RS232 adapter appears as a Communication port (COM port) in Windows and is used for UART serial communication with the product processor, running Linux, to send commands and received results. The commands sent are described in section 4.2.

The 10 MHz OCXO output signal interface consists of a Debug Board (DBB) that is connected to product PTC. To measure the frequency a Keysight 53230A frequency counter is used, that is connected to the PC via a USB port. Device Under Test (DUT) is the telecom product inside which the OCXO is mounted.

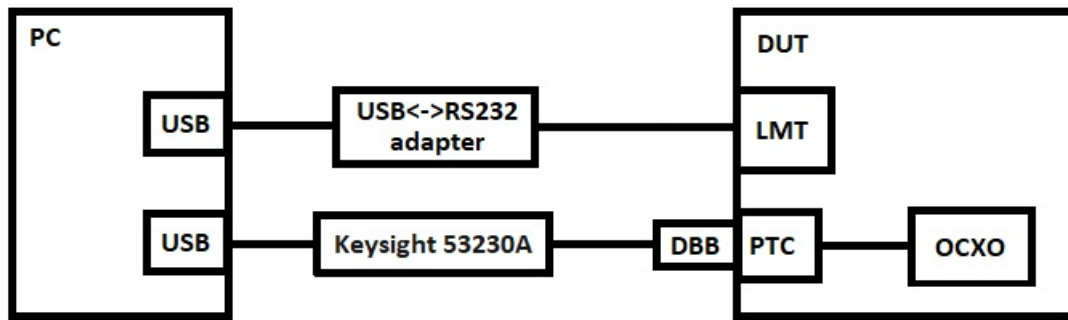


Figure 2. Hardware setup

2.4 Software setup

Device Test Platform (DTP), in Figure 3 is an automated testing framework used for both radio and digital products [6]. This platform makes test system development easier by allowing to re-use already existing code for various hardware operations.

DTP is a cross-platform application, developed using Microsoft C#.

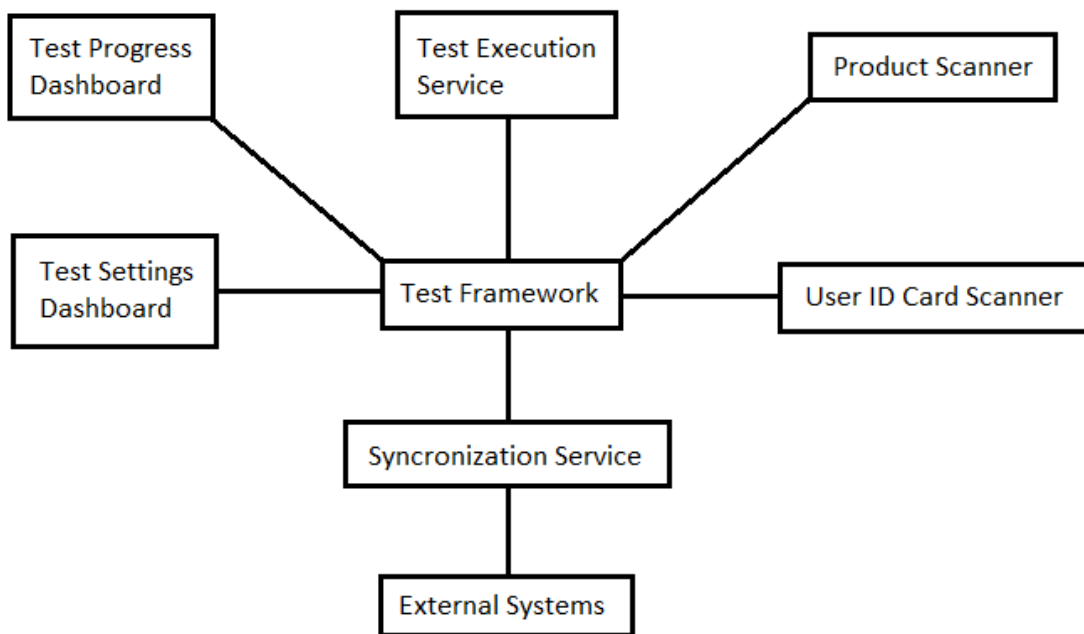


Figure 3. Device Test Platform

Test Framework is the core operations handler in the DTP. It exchanges data with all components of the DTP, processes all the received data and makes decisions. For example, it identifies the product when it receives product information from the product scanner, then it can load the correct test files for that product. It also identifies test users from the information received when the user ID card is scanned.

The Test Settings Dashboard shows the product platform information to the user. The information displayed includes product name, product revision and product description. The test and user settings can also be viewed and modified from there.

From the Test Progress Dashboard, the user will be able to view the revisions of the test files and the status of the running test. The test logs are also printed on this dashboard while the test runs. The user can abort, block, repeat or re-run tests from this dashboard.

Test Execution Service is the test execution engine that executes all the tests in the loaded test file.

Synchronization Service supports a two-way communication between the Test Framework and the external systems, for example, the automated mechanical test fixture. It translates information into a form that either the Test Framework or the External Systems can understand.

3 Frequency measurement

3.1 Measurement accuracy

The frequency counter works by counting the pulses from the input source and adding up the value until a predefined period of gate time is reached, after which the value will be stored to a register and read out. The gate time is based on an internal oscillator which is the reference against which the measurement is done. The instrument internal oscillator requires 45min warm-up time, to become stable and accurate [7]. To increase the accuracy of the frequency counter even further a very accurate rubidium oscillator is used as an external reference to the frequency counter. The following calculations are based on the instrument data sheet [8] where an example calculation is given. The values and calculations used below are for the designed measurement system.

$$T_{SS} = 20ps$$

$$K = 2 \text{ (for confidence 95\%)}$$

$$GateTime = 100ms$$

$$R_E = 4$$

$$SR = \frac{0,8 \times V_{pp}}{t_{RISE}} = \frac{0,8 \times 5,0}{10 \times 10^{-9}} = 400MV/s$$

$$T_{E(5V)} = \frac{(500\mu V^2 + E_N^2 + V_x^2)^{\frac{1}{2}}}{SR} = \frac{(500\mu V^2)^{\frac{1}{2}}}{400 \times 10^{-6}} = 1,25ps$$

$$\begin{aligned} Random &= \frac{1,4 \times (T_{SS}^2 + T_E^2)^{\frac{1}{2}}}{R_E \times GateTime} = \frac{1,4 \times [(20 \times 10^{-12})^2 + (1,25 \times 10^{-12})^2]^{\frac{1}{2}}}{4 \times 0,1} \\ &= 7,01 \times 10^{-11} \text{ parts error} \end{aligned}$$

$$Systematic = \frac{10ps}{GateTime} = \frac{10 \times 10^{-12}}{0,1} = 1 \times 10^{-10} \text{ parts error}$$

$$Timebase = Rubidium_{AGEING} + Rubidium_{TEMP} = 1 \times 10^{-9} + 0,5 \times 10^{-9} = 1,5 \times 10^{-9} \text{ parts error.}$$

$$\begin{aligned}
\text{Basic Accuracy} &= \pm[(K \times \text{Random}) + \text{Systematic} + \text{Timebase}] \\
&= (2 \times 7,01 \times 10^{-11}) + 1 \times 10^{-10} + 1,5 \times 10^{-9} \\
&= \pm 1,742 \times 10^{-9} \text{ parts error}
\end{aligned}$$

As result, $\pm 1,742$ parts error is $\pm 1,742$ ppb. The limit for frequency deviations was given as ± 100 ppb, so that small mistake ($\pm 1,742$ ppb) is barely noticeable. It is fine to say that used frequency counter is trustworthy.

3.2 Single measurement

The Keysight 53230A frequency counter user guide [7] provided instructions and an example code solution on how to use the instrument for frequency measurement. The example code contained code for initializing the instrument and carrying out a single measurement. However, it is important to point out that the example code was too reliant on the instrument manufacturer software, which is indicated by it using the `Agilent.Ag532xx.Interop` library. Having code that is dependant to one company, like in the current example, is not preferred since in the future there might be need to use the same code with an instrument from a different company, where this specific library might not exist, and this would break the code. However, the example code suits well for understanding the instrument and its operation.

The first measurements were done following the example code used driver initialization and communication code in Figure 4.


```

Ag532xx driver = null;

// Create driver instance
driver = new Ag532xx();

// Edit resource and options as needed. Resource is ignored if option
Simulate=true
//string resourceDesc = "GPIB0::3::INSTR";
string resourceDesc = "USB0::0x0957::0x1907::MY60020108::0::INSTR";
// resourceDesc = "TCPIP0::<ip or hostname>::inst0::INSTR";

string initOptions = "QueryInstrStatus=true, Simulate=false,
DriverSetup= Model=, Trace=false, TraceName=c:\\temp\\traceOut";

bool idquery = true;
bool reset = true;

// Initialize the driver. See driver help topic "Initializing the
IVI-COM Driver" for additional information
driver.Initialize(resourceDesc, idquery, reset, initOptions);
Console.WriteLine("Driver Initialized");

```

Figure 4. Driver initialization and communication

Before doing a measurement, the frequency counter needs to be initialized. Fetch() method is used to collect all measurement results but the execution will not continue unless Fetch() method has completed. However, for a 24-hour measurement it is a very risky design since there is no way to know which of the measurements failed in case of failure. Also, it will mean loss of all measurement data. The used code is provided in Figure 5.

```

List<double> measurementResults = new List<double>();

Console.WriteLine("Basic Frequency Measurement \n");
double[] Data;
int Channel = 1;
double Estimate = 1E6;
double Resolution = 1e-4;
driver.Frequency.Configure(Estimate, Resolution, Channel);
driver.Trigger.Count = 3; //At the moment measures 3 times
driver.Trigger.Delay = 3; //Seconds, time between measures
driver.Trigger.Source =
Agilent.Ag532xx.Interop.Ag532xxTriggerSourceEnum.Ag532xxTriggerSourceI
mmediate;
driver.Trigger.Slope =
Agilent.Ag532xx.Interop.Ag532xxSlopeEnum.Ag532xxSlopePositive;
driver.Measurement.Initiate();
driver.System.WaitForOperationComplete(100000); //Milliseconds, time
when operation finishes anyway
Data = driver.Measurement.Fetch();
int points = Data.Length;

```

Figure 5. Collecting frequency measurements

When Fetch() method has succeeded, then the measured results should be added to a list so they could be printed out and analysed. This provides an overview of what the instrument returned. The used code is shown below Figure 6.

```

Console.Write(" Measurement results: ");
for (int i = 0; i < points; i++)
{
    measurementResults.Add(Data[i]);
    Console.Write("{0};", Data[i]);
    //Adding results to the list and printing them
}

```

Figure 6. Adding measurement results to the list and print out

This method worked and returned the measured frequency results, but there was minimally a 1 second delay between each measurement. This was however not good enough to be considered a continuous measurement.

The above discussed solution was not suitable and as discussed, it had multiple problems. Firstly, the code is tied to a specific Keysight driver and their software library. Secondly, the Fetch() method was not suitable for 24h measurement since there was possibility of losing data in case of any issues. Thirdly, there is no way to stop the Fetch() method

execution, in case there should be a need. And finally, how to get the already measured data out in case of non-completion.

It is possible to stop the measurement using threads, shown in Figure 7, where two threads would run in parallel. One thread would handle the frequency measurements and the other waiting for Esc Key. This however was not a suitable solution either, as the `driver.Measurment.Abort()` left the frequency counter in a state where a new measurement was not possible to be started. This meant that, the frequency counter needed to be restarted after each abort, and this is not a suitable solution.

```

Thread measThread = new Thread(() =>
{
    Console.WriteLine("Started measuring, thread ID {0} ...",
        Thread.CurrentThread.ManagedThreadId);
    Data = driver.Measurement.Fetch();
});
measThread.IsBackground = true;
measThread.Start();

Console.WriteLine("Press Esc to stop measuring. Thread ID: {0}",
    Thread.CurrentThread.ManagedThreadId);
do
{
    if (Console.KeyAvailable)
    {
        if (Console.ReadKey().Key == ConsoleKey.Escape)
        {
            driver.Measurement.Abort();
            Console.WriteLine("Measurement aborted");
            break;
        }
    }
    else
    {
        Thread.Sleep(500);
    }
}
while (!measThread.Join(0));
int points = Data.Length;

Console.Write(" Measurement results: ");
for (int i = 0; i < points; i++)
{
    measurementResults.Add(Data[i]);
    Console.Write("{0};", Data[i]);
    //Adding results to the list and printing them
}

```

Figure 7. Parallel running threads

3.3 Continuous measurement

For a continuous measurement, the code needed to be made less dependent on the frequency counter software, for which the IVI driver was used. Actually, IVI driver is not a typical driver at all, but rather a set of standardized interfaces which instrument vendors

implement in their instrument software. As a result, a client application does not need to know, or care about, which vendors instrument it is using. The code about IVI driver and initializing is shown below in Figure 8.

```
const double Attenuation = 1.0D;
const double Impedance = 50.0D;
const int MeasTimeoutMs = 50;

IIviCounter freqCounter = null;

// Assume by default, the test has been succeeded, i.e. all
// measurements in the range.
bool testResult = true;

// While we don't have C# wrapper RCM Module for the Frequency Counter
// and we use the IVI interfaces and
// counter's COM component directly, the test takes relatively long
// time and we don't know how to set
// lease time for COM component's remoting object then we don't use
// here Platform's Resource Scheduler to
// instantiate the frequency counter. Instead, we instantiate it
// directly from Test Method with help of
// IVI Session Factory and release it when the Test Method finishes.

// Instantiate IVI Session Factory
IIviSessionFactory sessionFactory = new IviSessionFactoryClass();

// Instantiate Frequency Counter.
object freqCounterObj =
sessionFactory.CreateSession(_FreqCounterLogicalName);
freqCounter = freqCounterObj as IIviCounter;

// We need IOResource Descriptor to initialize the Frequency Counter.
// We use ConfigStore class from Platform for that
ConfigStore configStore = new ConfigStore(Logger as ILoggerGeneric2);
string ioResourceDescriptor =
configStore.GetHardwareAsset(_FreqCounterLogicalName);

// Initialize the Frequency Counter
freqCounter.Initialize(ioResourceDescriptor, true, true);

if (DebuggingOutputEnabled)
{
Output.TextOut("Number of channels: {0}", freqCounter.Channels.Count);
}

// We use the first channel.
```

```

string channelName = freqCounter.Channels.Name[1];
IIVICounterChannel channel = freqCounter.Channels.Item[channelName];

// Configure the channel
channel.Configure(Impedance,
IviCounterCouplingEnum.IviCounterCouplingAC, Attenuation);
channel.Slope = IviCounterSlopeEnum.IviCounterSlopePositive;

// Configure the frequency
// Get Estimate Frequency.
double estimateFreq = _expectedFreq;
int power = 0;
while (estimateFreq > 10.0d)
{
estimateFreq /= 10.0d;
power++;
}
estimateFreq = Math.Ceiling(estimateFreq);
estimateFreq = estimateFreq * Math.Pow(10.0d, power);

freqCounter.Frequency.ConfigureManual(channelName, estimateFreq,
_resolution);

freqCounter.Arm.Start.Type =
IviCounterArmTypeEnum.IviCounterArmTypeImmediate;

```

Figure 8. Initializing with IVI driver

After using IVI driver functionality, it became possible to do a continuous measurement with a do/while cycle. With a do/while cycle it was possible to get 3 measurement results per second, which was deemed continuous enough for current purposes. Then a timestamp was added to track the time of each measurement to know later when a possible deviation occurred. The limit for frequency deviations was given as ± 100 ppb. To easily observe when the measured result was out of the limits, a check was included for each measured value and “OUT” written to list. To use different data types in one list, Tuple list is used. The following Tuple list format is used: timestamp, measured frequency, limit check. The variable `_measTime` determines the length of the time to measure, and its value is given in a parameter file. The code about continuous measuring is shown in Figure 9.

```

var freqlist = new List<Tuple<string, double, string>>();
DateTime measStartTime = DateTime.Now;

do
{
    FlowControl.TestStopButton();

    freq = measFunction.Read(MeasTimeoutMs);

    // Check if the measured frequency is in range. If not and
    // _breakOnFirstFailure is set then break the measuring.
    // If measurement is out of range, set testResult to false.
    // Also, output it with Output.TextOutError.

    if (freq >= _expectedFreq - (_allowdDeviation * 10) && freq <=
        _expectedFreq + (_allowdDeviation * 10))
    {
        var time = DateTime.Now.ToString("s");
        freqlist.Add(new Tuple<string, double, string>(time, freq,
            inBorders));
    }
    else
    {
        var time = DateTime.Now.ToString("s");
        freqlist.Add(new Tuple<string, double, string>(time, freq,
            outOfBorders));

        if (_breakOnFirstFailure == true)
        {
            if (DebuggingOutputEnabled) {
                Output.TextOut("Measurement failed"); }
            testResult = false;
            break;
        }
    }
}
while (DateTime.Now - measStartTime < _measTime);

```

Figure 9. Getting measurements and checking if they are valid

4 Temperature measurement

4.1 Measurement accuracy

The temperature sensor D73002A4 is a high-accuracy remote temperature sensor monitor which has a built-in local temperature sensor [9]. The D73002A4 is physically located close to the OCXO. The temperature value used in this thesis is read from the local temperature sensor of D37002A4 which has the accuracy of $\pm 1^{\circ}\text{C}$ according to datasheet [9].

4.2 Single measurement

Since OCXO has a temperature dependency, and in-order to rule out any frequency deviations due to possible temperature fluctuations, the temperature of the product should also be measured. Since OCXO crystal will be kept at constant 77°C degrees [10] then an assumption is that this will be according to the specification and would not need to be measured. The requirement for the thesis assignment states to monitor PCB temperature sensor D37002A4, which is closest to OCXO, and which should be kept between 45°C - 55°C throughout the measurement process. For this the first task is to be able to read the temperature value of the D37002A4 temperature sensor, which can be done using an existing Testbox application. Testbox is a Linux application running on the product, used for running test related functions. The Testbox application is started using command line commands (Testbox commands), sent to the product via serial port. The challenge here is to create the code to send the necessary commands through the test system software to the product, over the serial port, which is a virtual COM port.

Using a simple Windows form application, it is possible to query info from COM port. The Testbox command used to read the temperature value from the D37002A4 sensor is `testbox temp_read D37002A4`. Response to this command, from the product can be seen in Figure 10.


```

[SERIAL]: textbox temp_read D37002A4
[SERIAL]: Sensor D37002A4 Temperature (C): 41
[SERIAL]: temp_read :: TESTBOX_PASS
[SERIAL]: [phv]#

```

Figure 10. Command output

As seen from the response message, it contains other information as well, so it is necessary to filter out and capture the temperature value we are interested in. Getting the temperature with COM port is shown in Figure 11.

```

public partial class Form1 : Form
{
    SerialPort comport = new SerialPort("COM3", 115200);

    public Form1()
    {
        InitializeComponent();

        comport.DataReceived += Comport_DataReceived;
        comport.Open();
    }

    private void Comport_DataReceived(object sender,
        SerialDataReceivedEventArgs e)
    {
        byte[] buf = new byte[comport.ReadBufferSize];

        (sender as SerialPort).Read(buf, 0, buf.Length);
        string str = System.Text.Encoding.UTF8.GetString(buf, 0,
            buf.Length);

        if (str.StartsWith("textbox"))
            return;

        string[] lines = str.Split(new char[] { '\n' });

        if (lines.ElementAt(1).Contains("temp_read ::
            TESTBOX_PASS"))
        {
            string[] data = lines.ElementAt(0).Split(new char[]
                { ':' });

```

```

        Console.WriteLine("temp: {0}",
            data.ElementAt(1).Trim());
    }
}

private void button1_Click(object sender, EventArgs e)
{
    comport.WriteLine("textbox temp_read D37002A4");
}

private void Form1_FormClosing(object sender,
    FormClosingEventArgs e)
{
    comport.Close();
}
}

```

Figure 11. Getting the temperature with serial port

4.3 Continuous measurement

As discussed in chapter [3.3] it is already possible to measure the frequency continuously which was defined as 3 times per second, but temperature reading is also needed to be made continuous. Since temperature fluctuates very slowly, due to the mass of the metal case-heatsink of the product, it is enough to read the temperature only once per minute, and this can be considered continuous. The Testbox command response message filtering and temperature value capturing are done using a regular expression (Regex). The timestamps and captured temperature values are stored in Dictionary. The product is described in the code as DUT. The used code is shown in Figure 12.

```

private static IDictionary<string, double> measTempList = new
    Dictionary<string, double>();

DateTime measStartTime = DateTime.Now;
bool testResult = true;
using (var dutHandle = GetDutCommunicationSession(_comPort))
{
    // Set up the DUT communication
    Session theDut = SetupDutCommunication(dutHandle);

    if (theDut == null)
    {

```

```

        return MethodResult.Error;
    }

    do
    {
        theDut.Clear();
        theDut.Timeout = this.Timeout / 1000.0;

        //Delay
        System.Threading.Thread.Sleep(1000);
        // Send the command
        string command = "testbox temp_read D37002A4 \n";
        theDut.Send(command);
        Output.TextOut("CMD: {0}", command.TrimEnd());

        string response;
        WaitForResponse(theDut, out response);
        Regex regexGetTemp = new Regex(@"Temperature.+(\d\d)",
            RegexOptions.Multiline | RegexOptions.CultureInvariant);
        Match matchTemp = regexGetTemp.Match(response);
        int temp = -1;

        if (matchTemp.Success)
        {
            temp = Int32.Parse(matchTemp.Groups[1].Value);
        }

        var time = DateTime.Now.ToString("s");
        measTempList.Add(time, temp);

        if (DebuggingOutputEnabled)
        {
            Output.TextOut(TextOutColor.Green, "DBG:
            lastWaitResult: {0}", this.LastWaitResult);
        }
    }
    while (DateTime.Now - measStartTime < _measTime);
}

```

Figure 12. Getting the temperature using COM port and dictionary

5 Merging measurement data

The frequency values in Tuple list are: DateTime, frequency, inBorders/outOfBorders. The temperature values in a Dictionary are: DateTime, temperature. A Dictionary was chosen since it is easy to match frequency and temperature while using DateTime as a key. The DateTime from Tuple list is used as a key to match the DateTime from Dictionary, to merge the DateTime, frequency, temperature and inBorders/outOfBorders as one line. The order of data in the line is such to give a good overview to the user on what date and time each measurement was done.

DateTime matching is only possible when the frequency measurement class and temperature measurement class are running in parallel. The command to run the two classes in parallel is handled by the test system.

When the predefined period for measurement has passed (24h), the measurements need to be stored from lists to a file. The matching of DateTime-s and sorting the data to a line is done before writing each line to file. The file format shall be .csv since it is smaller in size and faster to handle, it is easy to read without formatting and can be imported to MS Excel for additional analysis. The name and location of the file are handled by the test system and defined in the parameter list file. All the logic behind is shown in Figure 13.

Writing to file is done in frequency class and this means the Dictionary from the other class needs to be imported. For this getTemperatureDictionary() method is used. The value of variable _dumpMeasurements determines if the file shall be created or not, and it is defined by the input parameter.

```

var measTempList = mComportConnection.getTempertureDictionary();

if (_dumpMeasurements == true)
{
    using (StreamWriter writer = new StreamWriter(_measFile, true))
    {
        for (int i = 0; i <= freqlist.Count - 1; i++)
        {
            var date = freqlist[i].Item1;
            var freqValue = freqlist[i].Item2;
            var isInBorder = freqlist[i].Item3;

            if (measTempList.ContainsKey(date))
            {
                writer.WriteLine("{0};{1};{2};{3}", date,
                    freqValue, measTempList[date], isInBorder);
            }
        }
    }
}

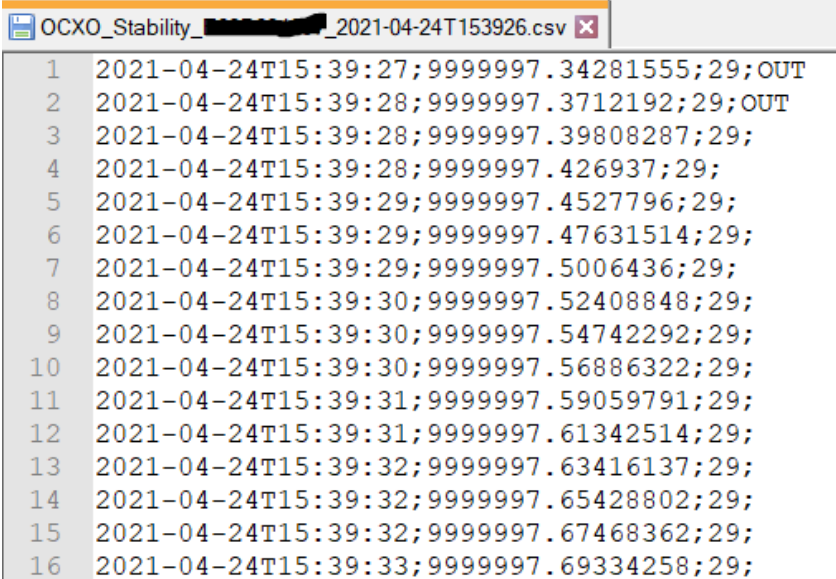
```

Figure 13. Merging frequency and temperature into CSV file

6 Analysis

The code execution results in a .csv file as shown in Figure 14 with timestamp, measured frequency in MHz and temperature value in °C. The fourth column in the file is for marking the measured result as out of the ± 100 ppb deviation limit, in which case there will be a marking “OUT”. In Figure 14 there are two results with “OUT” which are caused by low OCXO temperature during its warm-up stage. However, these measured values show that the code to detect and mark the “OUT” text in case the measurement is out of the ± 100 ppb deviation limit, works and is detecting and marking the deviations properly.

In the .csv file, the values are separated by semicolons and each row represents one measurement. The filename is automatically generated and includes the product serial number for future result tracking, which is blacked out in the Figure 14 below.



Line	Timestamp	Frequency (MHz)	Temperature (°C)	Status
1	2021-04-24T15:39:27	9999997.34281555	29	OUT
2	2021-04-24T15:39:28	9999997.3712192	29	OUT
3	2021-04-24T15:39:28	9999997.39808287	29	
4	2021-04-24T15:39:28	9999997.426937	29	
5	2021-04-24T15:39:29	9999997.4527796	29	
6	2021-04-24T15:39:29	9999997.47631514	29	
7	2021-04-24T15:39:29	9999997.5006436	29	
8	2021-04-24T15:39:30	9999997.52408848	29	
9	2021-04-24T15:39:30	9999997.54742292	29	
10	2021-04-24T15:39:30	9999997.56886322	29	
11	2021-04-24T15:39:31	9999997.59059791	29	
12	2021-04-24T15:39:31	9999997.61342514	29	
13	2021-04-24T15:39:32	9999997.63416137	29	
14	2021-04-24T15:39:32	9999997.65428802	29	
15	2021-04-24T15:39:32	9999997.67468362	29	
16	2021-04-24T15:39:33	9999997.69334258	29	

Figure 14. Data in CSV file

The chosen CSV format made it easy to import the result file to Excel for analysis. After importing, the measurement data can be filtered and plotted on a graph to show frequency and temperature change over time. In Figure 15, a graph of the first hour of testing is shown. The product is at room temperature of 21°C when powered on, but during the software loading and initialization phase, its temperature rises to about 29-30°C and this is what is shown in the first measurement. The Figure 15 shows temperature rise to the target level of 50°C during which the frequency also stabilizes.

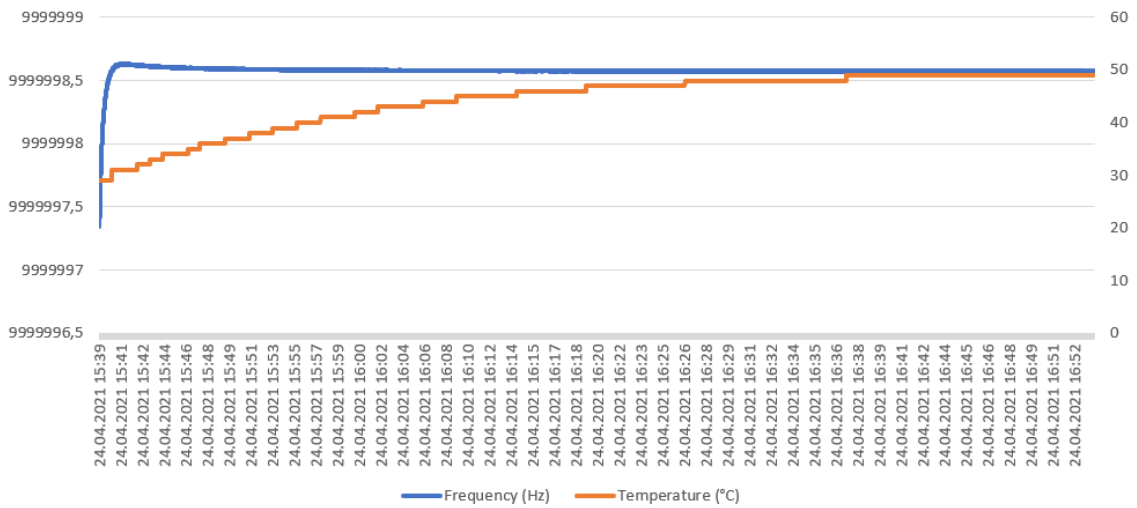


Figure 15. Product warm-up stage

The purpose of the test is to measure the frequency deviation over the period of 24h after the product has reached 50°C, which is shown in Figure 16. Here it can be seen that the temperature fluctuates between 50-51°C while the output frequency is quite stable, but still showing very minuscule average drift while becoming more stable towards the second part of the measurement.

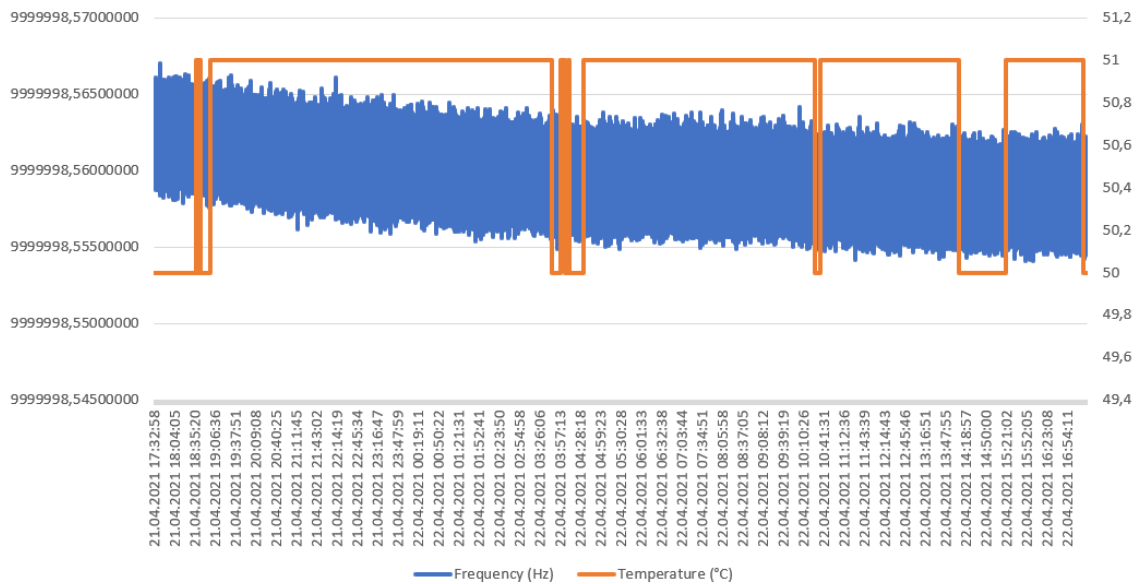


Figure 16. Frequency and temperature change over 24h

Then a check on the measured data was done to see its Min and Max deviation, for which Excel Min and Max functions were used. The values are:

Measured Min = 9999998,55404948Hz Measured Max = 9999998,56699854Hz

The calculated ± 100 ppb deviation limits are:

Lower limit (-100ppb) = 9999997,377Hz Upper limit (+100ppb) = 9999999,377Hz

To better visualize the measured frequency deviation, as shown in Figure 16, in relation to the deviation limits, a graph Figure 17 is presented. From here, it can be seen that the frequency deviations are barely visible at all, and very far from the limits. This is however, an expected outcome since the product used was a known good reference sample.

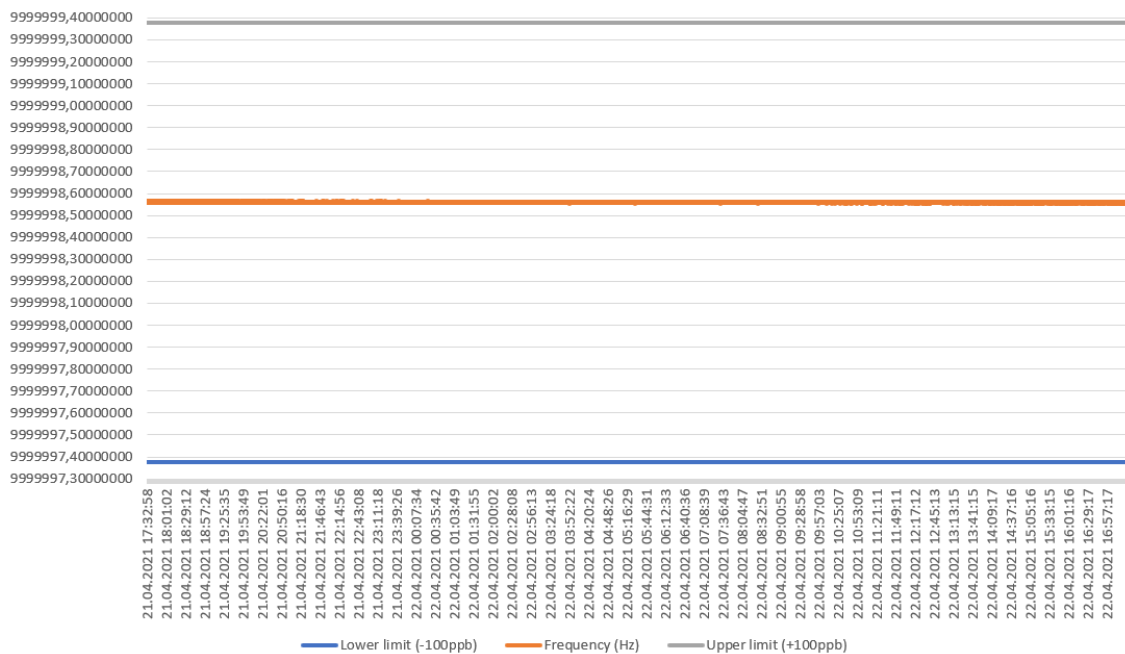


Figure 17. Frequency change over 24h with upper and lower limit

From the presented data and its analysis, it can be seen that different parts, and the entirety of, the program code fills its functions. The timestamp was verified by comparing the start and end timestamps to start and end times from the test program execution software log. The temperature values were based on the temperature sensor, and the values read out by the program code were seen rising as expected and no abnormalities were observed. The frequency measurements were done with a calibrated instrument and the values measured for a single measurement matched the values on the screen of the instrument. The measured values were also seen varying as expected and no abnormal behaviours were observed in the data. In the few measurements where the frequency was

out of the limits, the “OUT” mark was printed to the CSV file as expected. And finally, the CSV file creation code created the file, with proper formatting, data and name, as expected. There were no problems for the code to measure for up 25h straight and then writing all the data to file in the end in one go.

This proves that the designed hardware and software solution is functioning as intended, is tested and demonstrated to function properly, and can be used to try and solve the problem of the thesis, to measure the product OCXO output frequency and temperature over 24h.

7 Possible future developments

Monitoring OCXO internal temp as well, in addition to PCB temp sensor.

The thesis task description states to measure the board temperature via a board temp sensor that is close to the OCXO, to know that the OCXO environment temperature is stable. However, since the OCXO frequency depends on its internal temperature, then when only monitoring the board temp sensor, an assumption is made, that the OCXO internal temperature is stable when the environment is stable, while in reality, it is not known. To be able to rule out possible frequency deviations due to OCXO internal temperature deviations, then the OCXO internal temperature should also be measured and stored to file.

Progress counter to know how far test progress is and how much time is left.

The current implementation of the temperature and frequency measurement is done such that the data is collected to system RAM and stored to file once the specified time period is succeeded. This long term measurement shows up in the test system as one test item and while it is in progress, there is no way to know how long time has passed or how long time it will still need to run before completion. This information could be very valuable to the user, who otherwise could make incorrect conclusions about the progress of the test and try to abort the test execution. To solve this problem, a progress indication printout functionality could be added to the program code.

Parallel measurement for 6 products.

Due to the measurement time period being very long (24h) and the reality of possibly needing to test a lot of products in this test system, the limiting factor will be the throughput of the system. Since the measurement time is fixed, the only way to increase throughput is to test multiple products in parallel in the same test system. The used Device Test Platform supports testing 6 products, so the changes needed for the designed test system to support parallel testing, would be changes to the written program code and the IVI driver regarding instrument addressing. This would also mean separate instruments for each product.

8 Summary

The topic of this thesis came from the need to solve a real-life problem that the company is facing. The problem is that a customer observed performance issues with a telecom product that were proposed to be related to OCXO output frequency stability. In this thesis a test system, consisting of suitable hardware and software setup, was designed and verified. The hardware setup consisted of a PC, the frequency counter, the product and necessary cable connections. The focus in the thesis was on writing the software for single and continues measurement of frequency and temperature of the product, and on how to store the results to a file. The measurements were carried out on a sample reference product and the results were collected. The results file was analysed in Excel and graphs showing the frequency and temperature changes over time presented. The individual parts of the program code, and the whole code itself, were verified to fill the intended functions. Based on this the designed test system is found to be suitable for carrying out the OCXO output frequency stability measurement, in order to help solve the problem.

References

- [1] J. Lim, K. Choi, H. Kim, T. Jackson and D. Kenny, "Miniature Oven Controlled Crystal Oscillator (OCXO) on a CMOS Chip," IEEE, 2006.
- [2] H. Zhou, C. Nicholls, T. Kunz and H. Schwartz, "Frequency Accuracy & Stability Dependencies of Crystal Oscillators," Carleton University, 2008.
- [3] Company, "OCXO Specification," Internal document, 2019.
- [4] Company, "Vtemp mis-trigger report," Internal document, 2019.
- [5] M. Koyama, Y. Watanabe, H. Sekimoto and Y. Oomura, "An Experimental Study of Frequency Jumps," IEEE, 1996.
- [6] Company, "Test Platform," Internal document, Tallinn, 2019.
- [7] K. Technologies, "Keysight 53220A/53230A 350 MHz Universal Frequency Counter/Timer User Guide," Keysight Technologies, Bayan Lepas Free Industrial Zone, 11900 Penang, Malaysia.
- [8] K. Technologies, "53200A Series RF/Universal Frequency Counter/Timers Data Sheet," Keysight Technologies, Dallas, 2020.
- [9] T. Instruments, "TMP461 High-Accuracy Remote and Local Temperature Sensor with Pin-Programmable Bus Address Datasheet," Texas Instruments, Dallas, 2015.
- [10] Company, "Test requirements for products," Internal document, 2020.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis¹

I Grete Ohak

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Hardware and software setup for long term OCXO output frequency and temperature measurement”, supervised by Erkki Arus and Marten Kask
 - 1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
 - 1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

17.05.2021

¹ The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.