TCT-Analysis Framework Developers Guide

Mykyta Haranko

mykyta.haranko@gmail.com

July 5, 2017

Abstract

This guide was written for users and developers of the TCT-Analysis framework to understand basic aspects of the program and to make them able to extend its functionality for further data analysis.

Contents

1. Introduction

The Transient Current Technique (TCT) is widely used for studying the detector response after high irradiation fluences. The main idea is to use laser to produce charge carriers in the sensor, which is similar to the energy loss process of a minimum ionizing particle crossing the detector[1]. Then waveforms are stored in a binary files and processed using different methods.

TCT-Analysis is a framework proposed by Hendrik Jansen and developed to make the data analysis process fast and unified for all the laboratories using this technique. The basic idea was to create one open source framework, to which people can easily add their analysis and share it with other people.

To implement new modules developer don't have to go deeply into the code and can go directly to the Sections 3.3, 5.

2. Usage

This section gives simple instructions on downloading, installation and usage of the framework, also gives a brief description of the configuration file.

2.1. Downloading and Installation

There are two options available - console and graphical version of the framework. User can download pre-compiled Windows version or compile it from source.

2.1.1. Pre-compiled version

The pre-compiled version is distributed only for Windows, for the other operating systems it has to be compiled from source. TCT-Analysis requires ROOT 5.34.XX to be installed. Download needed version from `https://root.cern.ch/` and add it to the PATH environmental variable (usually offered by ROOT installer).

The next step is downloading the latest release of the framework from:

`https://github.com/DESY-FH-ELab/TCT-analysis/releases`

Windows pre-compiled versions are stored in *.zip file. To install the program one needs to unzip the archive to the desired directory. To start the program run **TCT-Analysis-QT.exe** in the **TCT-Analysis/bin** folder.

The last release was linked against ROOT 5.34.32. Tested on Windows 7, Windows 8.1 with ROOT 5.34.XX. Compiled using msvc2013 compiler.

2.1.2. Compiling from source using CMake

To compile the framework from source using CMake next prerequisites have to be satisfied:

- ROOT 5.X or 6.X has to be installed, ROOTSYS variable has to be defined.

- CMake 2.6 or older has to be installed and added to the PATH variable.

- (if GUI needed, **recommended**) Install Qt4 or Qt5.

- If you are going to use Qt5, export Qt5 installation dir, for example `export CMAKE_PREFIX_PATH=/opt/Qt5/5.7/gcc_64:$CMAKE_PREFIX_PATH`. For Windows you need to set `CMAKE_PREFIX_PATH` environmental variable.

- (optional, for single waveforms acquisition) If LeCroy RAW data files converter needed - put the external LeCroyConverter lib to the *external/LeCroyConverter/lib/libLeCroy.so.*

After satisfying dependencies, follow the next steps for both Windows and Linux operating systems:

1. Checkout the latest release of the program from the Github repository:

   `https://github.com/DESY-FH-ELab/TCT-analysis/releases`

   Or go to the master branch to get the latest version in development

   `https://github.com/DESY-FH-ELab/TCT-analysis`

2. Go to the **TCT-Analysis/build** directory.

3. Run "**cmake ⟨options⟩ ..**" command with the next available options.
   a) **-DWITH_GUI=ON** – compiles GUI version of the framework. Default – OFF.
   b) **-DWITH_LECROY_RAW=ON** – links against LeCroyRAW Converter library. Default – OFF.

4. Run **make install**.

5. Go to the **TCT-Analysis/bin** directory.

6. Run **tct-analysis** or **tct-analysis.exe** executable.

2.1.3. Compiling from source using qmake

To compile the framework from source using qmake next prerequisites have to be satisfied (compilation with GUI):

- For Linux: gcc 4.6 or later. For Windows: MSVC2015 or later.

- ROOT 5.X or 6.X has to be installed.

- Install Qt4 or Qt5.

After satisfying dependencies, follow next steps for both Windows and Linux operating systems:

1. Checkout the latest release of the program from the Github repository:

   `https://github.com/DESY-FH-ELab/TCT-analysis/releases`

   Or go to the master branch to get the latest version in development

   `https://github.com/DESY-FH-ELab/TCT-analysis`

2. Open **TCT-Analysis.pro** file in **Qt Creator**.

3. Follow instructions to configure compilation of the program.

4. Change **TCT-Analysis.pro** file according to the location of ROOT libraries and include files.

5. Press **Ctrl+B** to build the program.

6. Repeat previous steps for **tbrowser.pro** file to compile **TBrowser**.

7. Both executables (**tct-analysis** and **tbrowser**) has to be placed in the **TCT-Analysis/bin** folder.

8. Create **default.conf** file in **TCT-Analysis/bin** folder containing:

   DefaultFile = ../testanalysis/lpnhe_top.txt

9. Run the program. Note: be sure that Qt libraries are present in the PATH variable (LD_LIBRARY_PATH for Linux) or in the execution folder, otherwise program may fail to start, the same is for ROOT libraries.

2.2. First Run

The following folder structure is present in the base folder:

- **./bin** – contains executables and linked libraries
  - **./bin/default.conf** – contains name of the config file loaded by default
  - **./bin/execution.log** – log file of the program execution and data analysis
  - **./bin/tct-analysis.exe** – execution of the program
  - **./bin/tbrowser.exe** – TBrowser execution
- **./results** – here program stores output data by default
- **./testdata** – folder with test data, contains one data file with focus search at lpnhe folder
- **./testanalysis** – folder with sample program configuration files

- **./testsensor** – folder with sample sensor configuration files.

Depending on the selected options of compilation from Section 2.1 user can have either console or graphical version of the program.

In case of the console version of framework, file with configuration has to be specified:

```
./tct-analysis -af <path-to-configuration-file>
```

for example

```
./tct-analysis -af ../testanalysis/lpnhe_top.txt
```

In case of the graphical version, configuration file specified in **bin/default.conf** will be loaded by default. In Figure 1 main window of the program is shown.
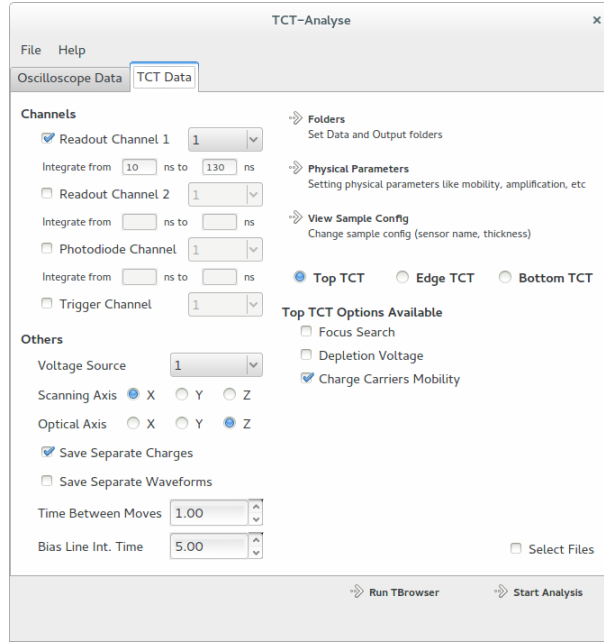

Figure 1: Main window of the framework graphical version.

To Open/Save new configuration file follow the corresponding file dialogue in the **File** menu at the top.

In Figure 2 structure of the main window explained with key control elements signed.

- To change data folder or output folder press **Folders** button.

- **Physical Parameters** button is used to set physical quantities used for certain analysis.

- **View Sample Config** used to change sensor name, thickness or general sensor configuration file.

- **Run TBrowser** button is used to view output root files with analysed data.
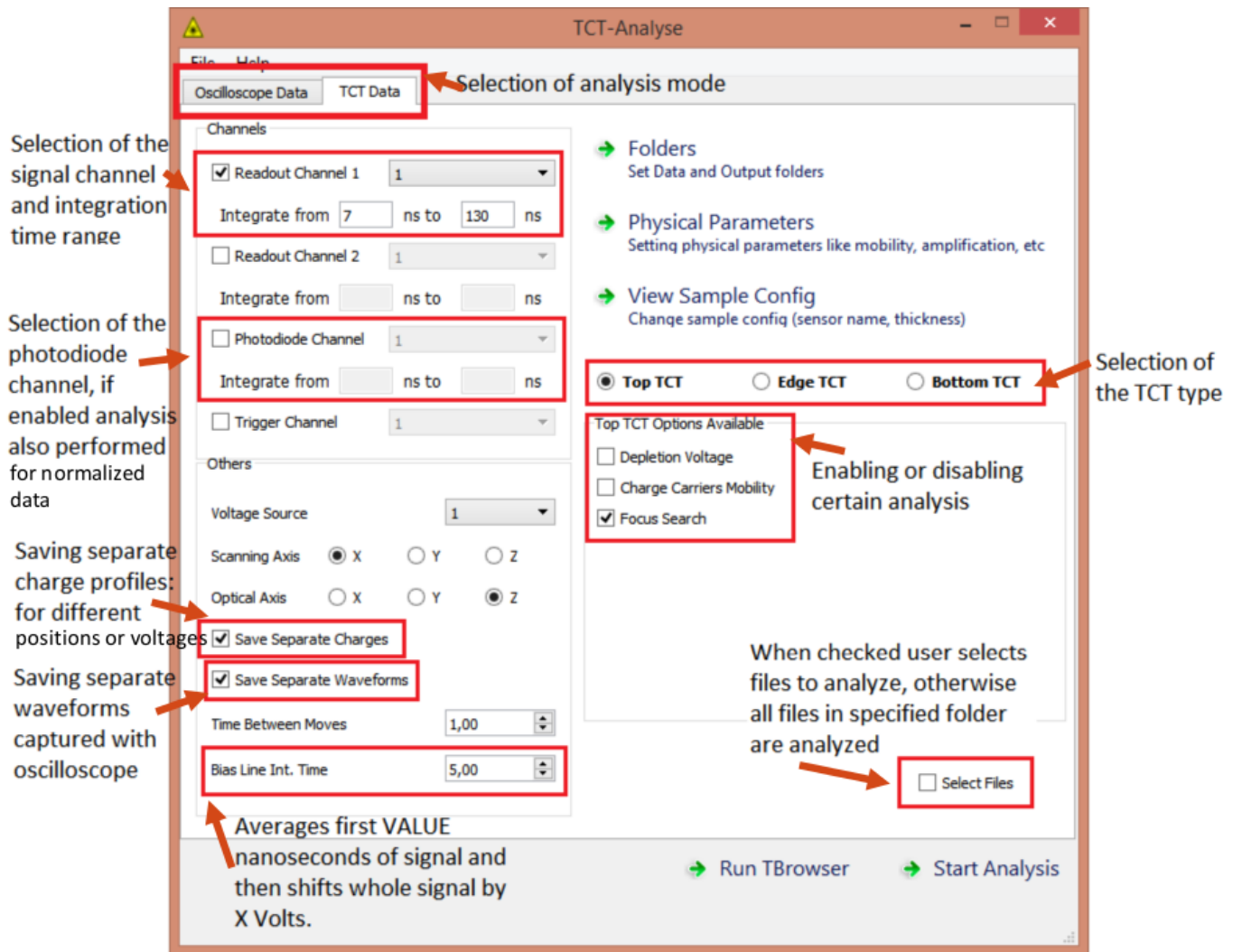
Figure 2: Structure of the main window.

- To run analysis click **Start Analysis** button. If **Select Files** is checked, one need to select *.tct data file to analyse, otherwise all data files in the specified folder analysed.

The program can operate in two modes: **TCT Data**, which analyses scans taken using PSTCT program from Particulars and **Oscilloscope Data**, which processes waveforms taken directly from the oscilloscope.

Normally for the graphical version user don't have to interact with the configuration *.txt file, but in case of the console version, configuration files are saved with comments, explaining all the parameters. Example of the configuration file is attached in Appendix A.

2.3. Sample Data File

Test analysis configuration file is present at the Github repository (**lpnhe_top.txt**) and provided with the corresponding data file (**top_focus_precise.tct**), containing sample focusing scan data.

To run analysis in the console program version simply execute the next command:

```
./tct-analysis -af ../testanalysis/lpnhe_top.txt
```

In the graphical version this configuration file is loaded by default. Click **Start Analysis** to run the program. Analysis Progress window will be displayed containing current progress on data analysis (Figure 3). To view the results click **Run TBrowser** button.
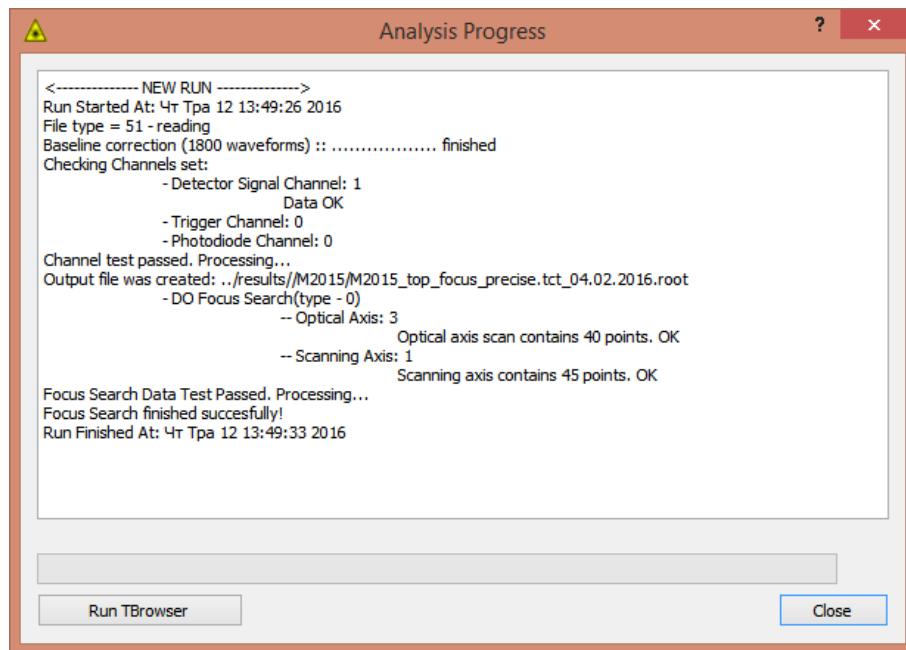
Figure 3: Analysis Progress window.

By default results will be placed in

./results/M2015/M2015_top_focus_precise.tct_04.02.2016.root,

this name consists of sample name, data file name and scanning date.
Internal data structure is the next:

- **sample_signals** – contains sample signals from all the channels

- **detector_signals** – contains all the signals from all the channels if *Save Separate Waveforms* is checked

- **Scan Name** – contains data from certain analysis

For the **FocusSearch** scan the next graphs are available:

- **ChargeVsDistance** – MultiGraph with all charge profiles on one graph

- **./charges** folder – contains waveforms for different positions along the optical axis

- **FWHM** – contains width of the falling edge from the charge profiles fit this graph is the most important for the focus search

- **MinCharge** – can be used to cross check the FWHM fit. Contains minimal charge values when laser is lighting on the strip (usable when the strip width is comparable to the beam width)

- **Missalignment** – can be used to find the misalignment of the optical axis with respect to the scanning axis

To find additional information on the scan results interpretation see Section 4 (partially, Section 4.1.1 for Top TCT analysis output file interpretation).

3. General Code Review

General program flow is the same for both graphical and console versions. In the following sections the console version of the program will be described by default, excepting remarks that will be made during the discussion.

The program can operate in two modes: **TCT Data** which analyses scans taken using PSTCT program from Particulars and **Oscilloscope Data** which processes waveforms taken directly from the oscilloscope.

After starting of the analysis process, the program follows next steps:

1. Reading of the configuration file with `util` class. It contains `void` `parse(std::ifstream &cfgfile)` method, which parses the configuration file and stores Keyword-Value pairs in `std::map<std::string, std::string> _id_val` map.

2. Parameters need to be converted to the corresponding type and stored in memory. Several classes of the same structure are used:

   - `mode_selector` - is responsible for the program operation mode selection.
   - `sample` - is responsible for the sensor parameters storage.
   - `tct_config` - is responsible for the program operation in TCT Data mode.
   - `analysis` - stores configuration for the single waveforms from the oscilloscope analysis, also performs this analysis.

   Each class mentioned above takes `util` class as the argument, then scans keywords for needed parameters and saves their values.

3. The analysis started in the selected mode. The programs scans data folder for needed file formats (*.tct files for TCT Data mode).

4. `Scanning` class dynamic instance created.

5. Method `bool Scanning::ReadTCT(char* filename, tct_config* config)` called to analyse the data, passing file path and configuration class as the arguments. This method manages reading of the data file, runs the analysis and saves the result to the output file, for the details see Section 3.1

6. Steps 4-5 are repeated for all the files found.

3.1. Scanning Class

`TCT:Scanning` class manages reading of the data file, runs the analysis and saves the result to the output file. All the managing is done inside the `bool Scanning::ReadTCT(...)` method. In the source file one can see, that there are two different definitions of its header: in graphical version it sends the progress of the data analysis to the interface.

TCT data file is read using `TCTReader` class borrowed from Particulars and described in Section 3.2. `ReadTCT` method checks the file for presence of the data in specified channels using `CheckData()` method, creates *.root output file using `CreateOutputFile()` method, saves separate waveforms (if specified) and sample waveforms using `Separate_and_Sample()` method. After all the preparations it tries to run all the registered analysis modules checking if they are enabled and are of the same TCT type as selected.

3.2. TCT Data Reader

`TCTReader` file contains two classes: `TCTReader` and `TCTWaveform`. First one is responsible for reading of the data file, getting the histograms, etc. `TCTWaveform` is used to operate with multiple waveforms. Both classes were borrowed from Particulars.

The most important methods of the `TCTReader` class are:

- `TCTReader::TCTReader(char *FileNameInp, Float_t time0, Int_t Bin)` – constructor method, runs the read out of the data file

- `void TCTReader::ReadWFsBin(Float_t time0)` – reads the waveforms

- `TH1F *TCTReader::GetHA(Int_t ch , Int_t x, Int_t y, Int_t z, Int_t nu1, Int_t nu2)` – returns the TH1F histogram for certain channel, position and voltage.

- `TCTWaveform *TCTReader::Projection(int ch, int dir,int x,int y,int z, int nu1, int nu2, int num)` – projects the data into the `TCTWaveform`, which can be analysed later.

- `void TCTReader::CorrectBaseLine(Float_t xc)` – function corrects the baseline (DC offset) of all waveforms. It averages the signal in range (0,xc) and then shifts the signal by the mean value.

- `void TCTReader::PrintInfo()` – prints the file info.

3.3. TCT Module Class

`TCTModule` is the base class for module implementation. It contains basic functions for plotting, integrating the histograms, checking the data, etc. It is run from `Scanning` class using the `bool TCTModule::Do(TCTReader *in_stct, TFile *in_rootfile)` method. To see the example of usage of the methods go to Sections 4, 5.

It consists of several base methods:

- `TCTModule::TCTModule(tct_config* config1, const char* name, TCT_Type type, const char* title)` – module constructor method.

  - `tct_config* config1` – instance of the `tct_config` class

  - `const char* name` – name of the class used internally, without spaces.

  - `TCT_Type type` – type of the TCT measurement, can be: _Top, _Edge, _Bottom.

  - `const char* title` – title of the analysis, displayed everywhere.

- `bool TCTModule::Do(TCTReader *in_stct, TFile *in_rootfile)` – method called from outside to manage analysis process.

- `virtual bool TCTModule::CheckModuleData()` – **has to be re-implemented by developer**, checks the data is usable for the analysis. For example, focusing scan has to contain at least 10 points along the scanning axis to be able to fit the charge profile.

- `virtual bool TCTModule::Analysis()` – **has to be re-implemented by developer**, performs full analysis process.

There are several methods that have to be implemented when new `TCTModule` has additional analysis parameters, see Section 5 for additional information. These methods are used for the graphical version of the program and described below:

- `virtual void PrintConfig(std::ofstream &conf_file)` – used to save parameter values to the file.

- `virtual void AddParameters(QVBoxLayout* layout)` – used to add these parameters to the interface.

- `virtual void FillParameters()` – used to set the values of parameters to the interface widget.

- `virtual void ToVariables()` – used to read the values from the widget and save to variable.

To save time for implementing of new modules several methods were implemented and have to be used (when not specified argument supposed to be input):

- `void SwitchAxis(Int_t sw, Int_t& nPoints, Float_t& step, Float_t& p0)`
  – used to briefly set the scanning axis.

    - `Int_t sw` – input, axis to set.

    - `Int_t& nPoints` – output, variable containing the number of points.

    - `Float_t& step` – output, the axis step.

    - `Float_t& p0` – output, the starting point on axis.

- `void CalculateCharges(Int_t Channel, Int_t Ax, Int_t numAx, Int_t scanning, Int_t numS, TGraph **charges, Float_t tstart, Float_t tfinish)` – calculating the charge profiles.

    - `Int_t Channel` – channel with signal.

    - `Int_t Ax` – variable axis, optical axis for focusing, voltage for voltage scans.

    - `Int_t numAx` – number of points for axis specified in previous step.

    - `Int_t scanning` – the scanning axis id.

    - `Int_t numS` – the scanning axis number of points.

    - `TGraph **charges` – output, resulting charge profiles.

    - `Float_t tstart` – integration time start, nanoseconds.

    - `Float_t tfinish` – integration time start, nanoseconds.

- `TGraph** NormedCharge(TGraph** sensor, TGraph** photodiode, Int_t numP)` – normalizing charge profiles using photo diode signals.

    - `TGraph** sensor` – sensors charge profiles.

    - `TGraph** photodiode` – charge profiles from photo diode.

    - `Int_t numP` – number of profiles.

- `void FindEdges(TGraph* gr, Int_t numS, Float_t dx, Double_t &left_edge, Double_t &right_edge)` – searching for the sensor edges, useful in Edge TCT for defining sensor position.

    - `TGraph* gr` – charge profile used to find edges.

    - `Int_t numS` – number of scanning points.

    - `Float_t dx` – step.

    - `Double_t &left_edge` – output, position of the left edge.

    - `Double_t &right_edge` – output, position of the right edge.

- `void FindEdges(TGraph** gr, Int_t numP, Int_t numS, Float_t dx, Float_t* left_pos, Float_t* left_width, Float_t* right_pos, Float_t* right_width)` – the same as previous one, but for multiple charge profiles.

- `TGraph* GraphBuilder(Int_t N, Float_t *x, Float_t *y, const char *namex, const char *namey, const char *title)` – used to build graphs from x,y points, return TGraph pointer.

  - `Int_t N` – number of points.

  - `Float_t *x` – x-axis.

  - `Float_t *y` – y-axis.

  - `const char *namex` – title of the x-axis.

  - `const char *namey` – title of the y-axis.

  - `const char *title` – title of the graph.

- `void GraphBuilder(Int_t N, Float_t *x, Float_t *y, const char *namex, const char *namey, const char *title, const char *write_name)` – the same as previous one, but writes the graph directly to the output *.root file.

  - `const char *write_name` – name of the graph in the output file.

- `TGraph* GraphBuilder(Int_t N, Double_t *x, Double_t *y, const char *namex, const char *namey, const char *title)` – the same as the first GraphBuilder, but for Double_t arrays.

- `void GraphBuilder(Int_t N, Double_t *x, Double_t *y, const char *namex, const char *namey, const char *title, const char *write_name)` – the same as the second GraphBuilder (with writing to the file), but for Double_t arrays.

- `void GraphSeparate(Int_t N, TGraph **gr, const char *dir_name, const char *namex, const char *namey, const char *title, const char *name_0, Double_t *name_1)` – writes separate charge profiles to the file

  - `Int_t N` – number of charge profiles.

  - `TGraph **gr` – charge profiles to write.

  - `const char *dir_name` – directory to write.

  - `const char *namex` – title of the x-axis.

  - `const char *namey` – title of the y-axis.

  - `const char *title` – title of the profiles.

  - `const char *name_0` – name of the profile to write to the file, consists of string and number. This is string.

  - `Double_t *name_1` – this is number.

- `void GraphSeparate(Int_t N, TGraph **gr, const char *dir_name, const char *namex, const char *namey, const char *title, const char *name_0, Float_t *name_1)` – the same as previous, but for Float_t numbers to write (last argument).

- `void MultiGraphWriter(Int_t N, TGraph **gr, const char *namex, const char *namey, const char *title, const char *write_name)` – used to output the multi-graph with all the charge profiles.
    - `Int_t N` – number of charge profiles.
    - `TGraph **gr` – charge profiles to write.
    - `const char *namex` – title of the x-axis.
    - `const char *namey` – title of the y-axis.
    - `const char *title` – title of the graph.
    - `const char *write_name` – name to write to the file.

- `void SetFitParameters(TF1* ff, Double_t p0, Double_t p1, Double_t p2, Double_t p3)` – simply sets the fit parameters values to the fit function, used to save space.
    - `TF1* ff` – fit function.
    - `Double_t p0` – parameter 0.
    - `Double_t p1` – parameter 1.
    - `Double_t p2` – parameter 2.
    - `Double_t p3` – parameter 3.

- `Double_t GraphIntegral(TGraph *gr, Double_t x1, Double_t x2)` – integrates the graph in x1 to x2 range.

- `Double_t abs(Double_t x)` – return the absolute value of x.

- `void ChargeCorrelationHist(TGraph** sensor, TGraph** photodetector, Int_t numO)` – build a charge correlation histogram to check the correlation of sensor and diode signals.
    - `TGraph** sensor` – sensor charge profiles.
    - `TGraph** photodetector` – photo detector charge profiles.
    - `Int_t numO` – number of graphs.

To add the new analysis (module) to the program, one need to inherit it from the `TCTModule` class and place corresponding files to the **./src/modules** and **./include/-modules** directories. The next is registering of the module, it has to be done in **./src/tct_config.cc** file by calling RegisterModule method. For details, follow tutorial in Section 5.

4. Implemented Modules

4.1. Top TCT

Top-TCT is defined by illuminating of the sensor top surface. Top-TCT is used to extract such quantities as the depletion voltage, velocity of the charge carriers, which is used for calculation of the mobilities of electrons and holes.

4.1.1. Focus Search

Idea

(a) Scan along X axis.

(b) Changing of the beam spot size due to the different optical distances.
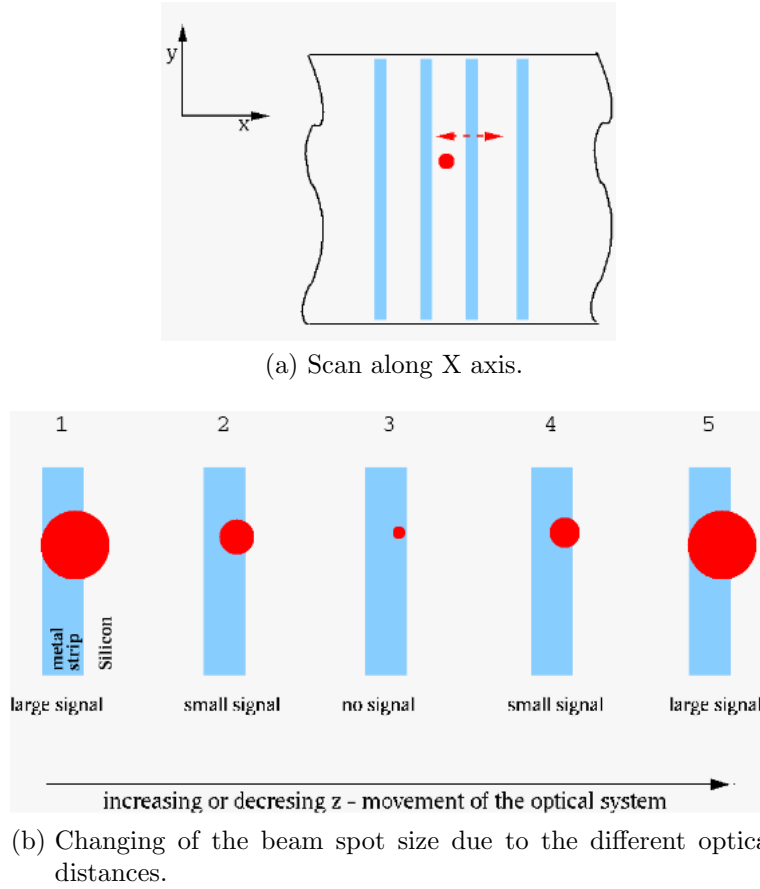
Figure 4: Schematic representation of the focus search method for a strip detector.[2]

To find the focus for a strip sensor one need to follow instructions proposed in Ref.[2]. The basic idea is shown in Figure 4(a): when moving the beam spot along the X axis, less charge will be observed at the strip region - part of light will be reflected from the strip. Then, by changing the optical distance, the size of the beam spot can be changed (Figure 4(b)). In Figure 5 the curves for focused and un-focused beam are shown. If the beam is out of focus, charge measured using the sensor, starts to increase slowly when scanning along the X-axis. This curve could be fitted using the error function, and the

FWHM value can be extracted from the fit. When the beam spot gets smaller, the same happens to the FWHM value.
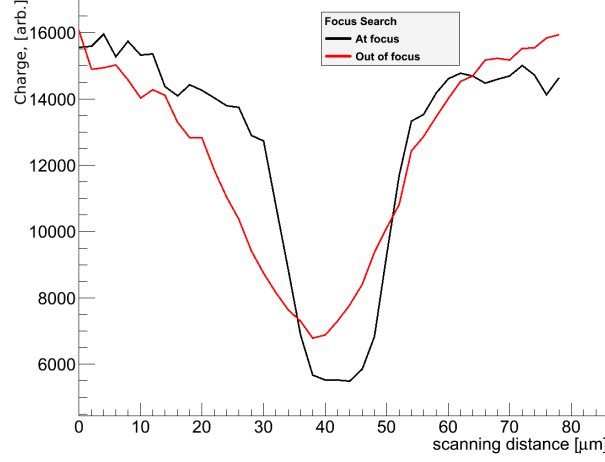

Figure 5: Scans for focused and un-focused beam, Top-TCT.

The FWHM is plotted for different optical distances and fitted with the second-order polynomial, see Figure 6. The position with the smallest value of the FWHM is defined to be a focus position.
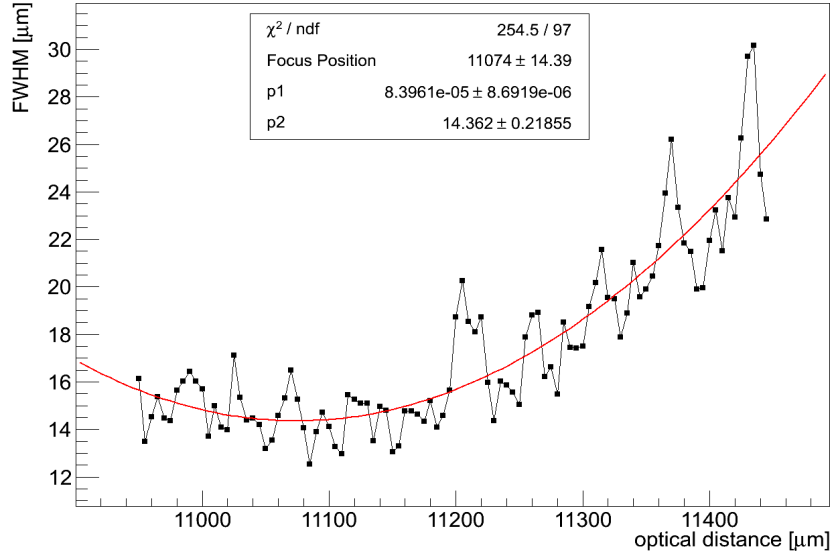

Figure 6: FWHM values for different optical positions, Top-TCT.

Realization

Top TCT Focus Search is realized in the `ModuleTopFocus` class inherited from the `TCTModule` class. The `CheckModuleData()` class method checks that scanning axis contains more than 10 scan points to be sure that charge profiles can be fitted. The `Analysis()` class method performs charge profiles building and fitting.

16

4.1.2. Depletion Voltage

Idea

To find the depletion voltage, the charge collected at certain position between strips for the different bias voltages has to be integrated. After reaching the depletion voltage charge stops to increase and becomes constant. Plotting the charge with square root of voltage, the dependency can be obtained, which has to be fitted with two lines - one fits the rising part, second one - the constant part, when all charge is collected (detector is fully depleted). The intersection of these curves gives the value of the full depletion voltage. For the sample sensor from Particulars Figure 7 shows the depletion voltage value of 33.6 V.
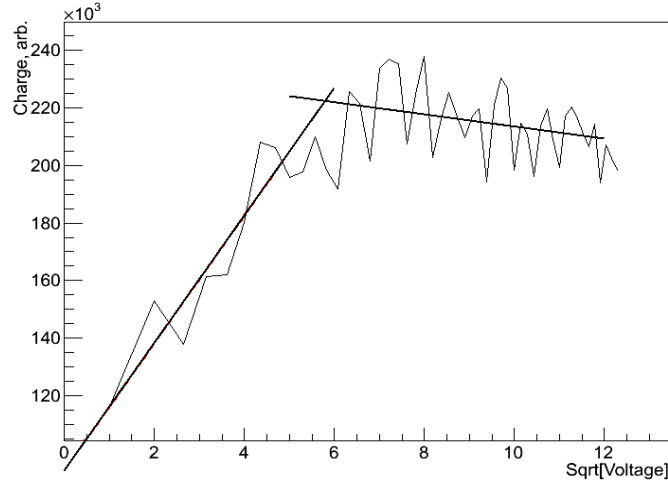

Figure 7: Depletion voltage search, Top TCT.

Realization

Top TCT Depletion Voltage search is realized in the `ModuleTopDepletion` class inherited from the `TCTModule` class. The `CheckModuleData()` class method checks that voltage scan contains at least 7 scan points to be sure that charge dependency vs. voltage can be fitted. The `Analysis()` class method performs charge dependency building and fitting.

4.1.3. Mobility

Idea

The TCT is used to calculate the charge carrier mobility. The 80% of the red light is absorbed after 5 μm of silicon[3] - this is used to measure the charge carrier mobility of different type. After applying positive voltage to the strips (p-type bulk sensor from Particulars) and negative to the back side, holes drifting to the back side and electrons to the strips. Shooting from the top side: electrons are collected immediately, and the transit time is the time of the hole drift.
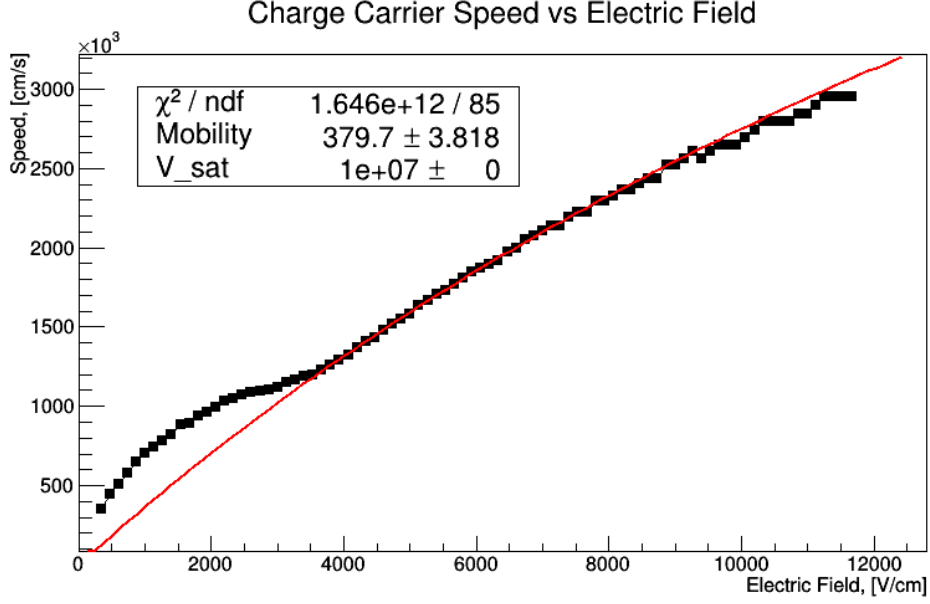
Figure 8: The average hole drift velocity for p-type sensor from Particulars.

The average hole drift velocity through the sensor is given by $v_h = \frac{W}{t_{transit}}$, where $t_{transit}$ is the duration of the signal, W is the thickness of the sensor. For the sample sensor from Particulars the average hole drift velocity is shown in Figure 8. The fit function is given by (1), according to the (2), where v_{sat} is the holes saturation velocity, μ_0 is the low-field mobility, E is the electric field. The fit results in a value of $379.7\frac{cm^2}{Vs}$, with assumption of detector thickness 300 μm and saturation velocity in silicon of $v_{sat} = 1 \times 10^7 \frac{cm}{s}$.

$$f(E) = \frac{p_0 E}{1 + \frac{p_0 E}{v_{sat}}}. \tag{1}$$

$$v_h(E) = \frac{\mu_0 E}{1 + \frac{\mu_0 E}{v_{sat}}}. \tag{2}$$

Realization

Top TCT Mobility search is realized in the `ModuleTopMobility` class inherited from the `TCTModule` class. The `CheckModuleData()` class method checks that voltage scan contains at least 7 scan points to be sure that velocity dependency vs. voltage can be fitted. The `Analysis()` class method performs velocity dependency building and fitting.

4.2. Edge TCT

4.2.1. Focus Search

Idea

To find the focus with Edge TCT the same approach is used as for the Top TCT measurements. Instead of scanning and passing the strips, scan is done along the entire detector thickness. The charge profile falling and rising edge are fitted, as it is shown in

Figure 9(a). Red arrow shows the direction of the scan. FWHM values are taken from the fits and plotted for different optical distances, then fitted as it shown in Figure 9(b).
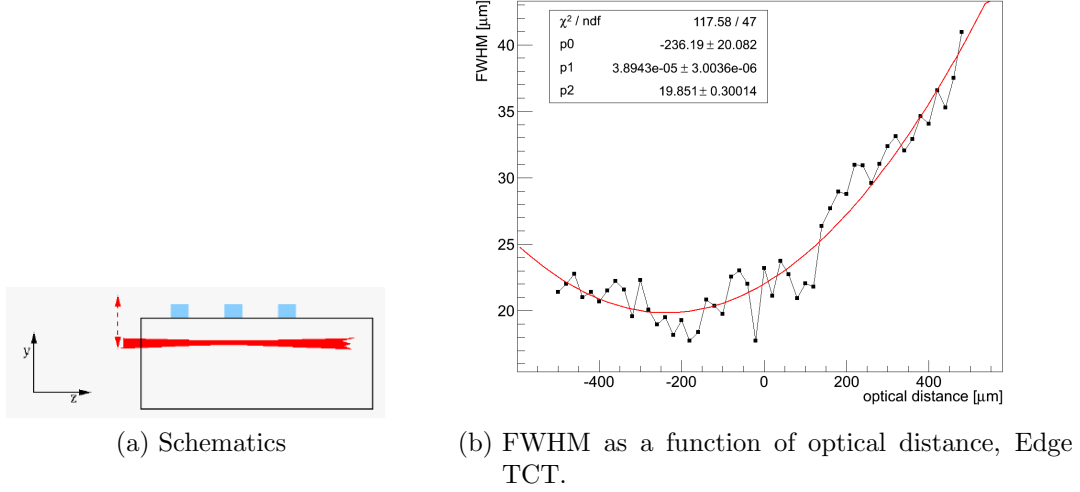

(a) Schematics

(b) FWHM as a function of optical distance, Edge TCT.

Figure 9: Edge TCT laser beam focusing.

Realization

Edge TCT Focus Search is realized in the `ModuleEdgeFocus` class inherited from the `TCTModule` class. The `CheckModuleData()` class method checks that scanning axis contains more than 10 scan points to be sure that charge profiles can be fitted. The `Analysis()` class method performs charge profiles building and fitting.

4.2.2. Depletion Voltage

Idea

The charge profiles for different bias voltages in arbitrary units are shown in Figure 10. When the detector is not fully depleted the signal near the bottom of the detector still can be observed due to the different doping concentrations in the n and n+ layers.
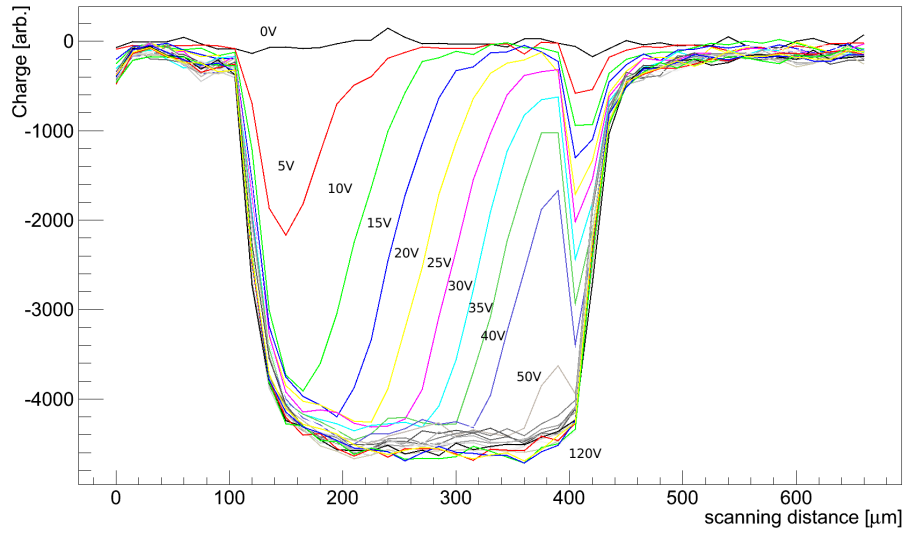
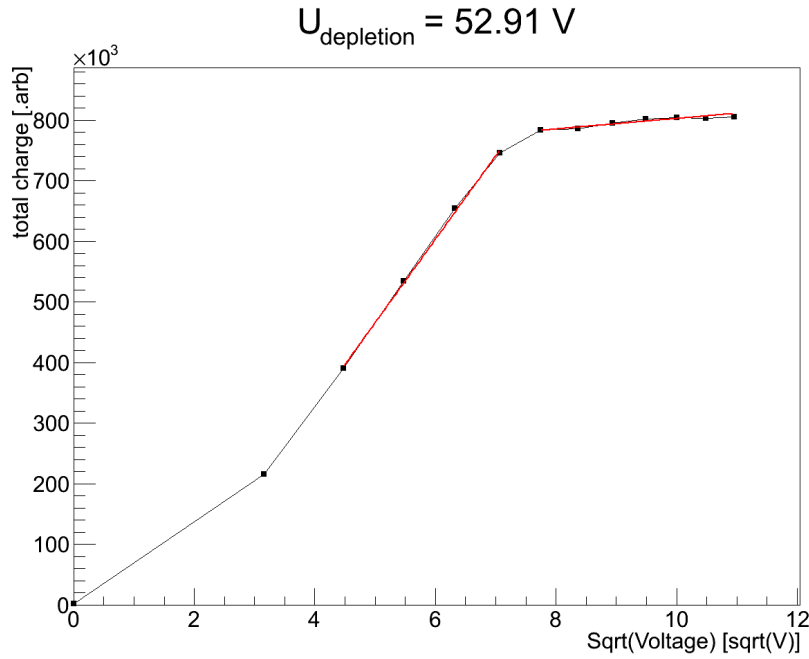Figure 10: Charge profiles for different voltages, Edge TCT.

Figure 11: Depletion voltage of the CE2339 sample sensor.

Depletion voltage was found using the same method, as it was used for the Top TCT measurements, see Section 4.1.2, but the charge is also integrated through all the detector thickness. As it shown in Figure 11, obtained value is 52.91 V and is consistent with the value of 49 V measured using the probe method.

Realization

Edge TCT Depletion Voltage search is realized in the `ModuleEdgeDepletion` class inherited from the `TCTModule` class. The `CheckModuleData()` class method checks that

scanning axis contains more than 10 scan points and voltage scan contains at least 7 scan points to be sure that charge dependency vs. voltage can be fitted. The `Analysis()` class method performs charge dependency building and fitting.

4.2.3. Electric Field Profiles

Idea

The induced current in the detector can be expressed by (3), where e_0 is the elementary charge, $N_{e,h}$ is the number of created e-h pairs near the strip, A is the amplifier amplification, $\tau_{eff,e,h}$ is the effective trapping time, $v_{e,h}$ is the drift velocity and E_ω is the weighting field. For simple pad detectors term $\overrightarrow{v}_{e,h}(t) \cdot \overrightarrow{E}_\omega$ is simply $\frac{v_{e,h}}{W}$, where W is the detector thickness.

$$I_{e,h}(t) = Ae_0 N_{e,h} \exp(-\frac{t}{\tau_{eff,e,h}})\overrightarrow{v}_{e,h}(t) \cdot \overrightarrow{E}_\omega. \tag{3}$$

According to the Ref. [5], Prompt Current Method is used to obtain the electric field profiles. The measured current amplitude immediately after charge carrier generation $(\exp(-\frac{t}{\tau_{eff,e,h}}) \approx 1)$ can be expressed as:

$$I(y, t \sim 0) = Ae_0 N_{e,h} \frac{v_e(y) + v_h(y)}{W} = Ae_0 N_{e,h} \frac{\mu_e(y) + \mu_h(y)}{W} E(y). \tag{4}$$

From this point there could be two ways: one way to extract the electric field profiles is to use the estimation of the charge carriers number from the laser power, measured using the photodetector. But the problem is that one need precise values of the amplification, $N_{e,h}$ - which is difficult to estimate due to the loses in the optical system, in the silicon, etc. The second way is to use formula (5) as the constraint and to solve numerically the equation (4). That was done using the bisection method[6].

$$V_{bias} = \int_0^W E(y)dy. \tag{5}$$

As the result the electric field profiles were extracted and are presented In Figure 12.

Also the velocity profiles could be extracted using equation (6), where v_{sat} is the holes saturation velocity, μ_0 is the low-field mobility, E is the electric field.

$$v_h(E) = \frac{\mu_0 E}{1 + \frac{\mu_0 E}{v_{sat}}}. \tag{6}$$

Realization

Edge TCT Edge Field Profiles search is realized in the `ModuleEdgeField` class inherited from the `TCTModule` class. The `CheckModuleData()` class method checks that scanning axis contains more than 10 scan points to be sure that charge profiles can be fitted. The `Analysis()` class method performs charge profiles building, numerical calculations and output. Also field profiles module contains additionally re-implemented functions and described more advanced in Section 5.
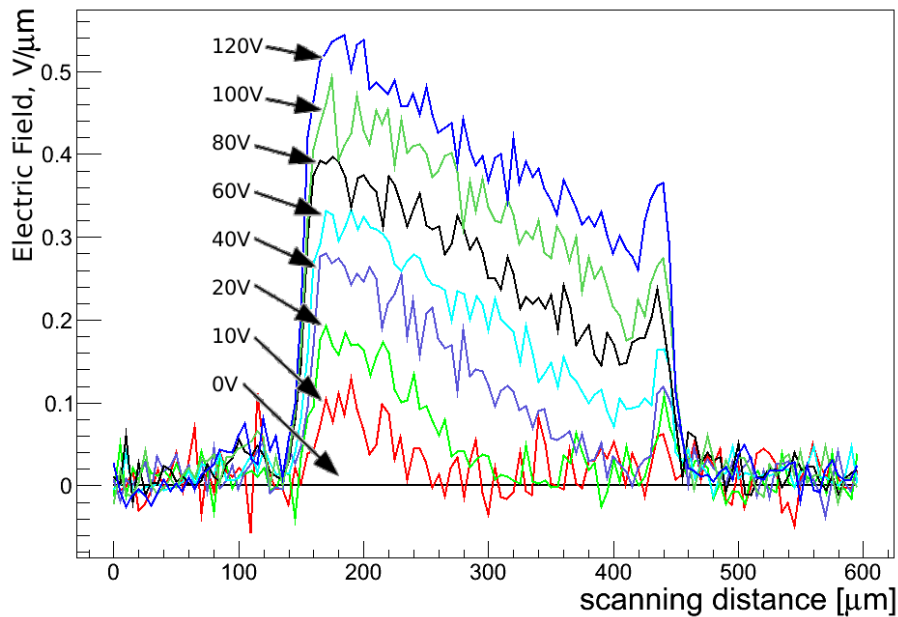
Figure 12: Field profiles for different voltages, Edge TCT.

5. Tutorial on TCTModule Implementation

This section is a step by step tutorial on module implementation using `ModuleEdgeField` class (See Section 4.2.3) as the example.

New module creation consists of several steps:

1. Module implementation and placement to the **modules** directory.
   Field profiles module was selected due to the presence of several features such as the `_EV_Time` parameter which is used for averaging of the signal during the first moments of evolution - typically less than $1ns$. This parameter is included in the header file, which contains variable definition and set/get methods:

```
public:
            ...
       void SetEV_Time(float value) { _EV_Time =
          value; }
       float GetEV_Time() { return _EV_Time; }
private:
            ...
       float _EV_Time;
```

As parameter has to be displayed in the GUI, additional methods has to be re-implemented according to Section 3.3:

```
public:
         ...
#ifdef USE_GUI
```

```
            void PrintConfig(std::ofstream &conf_file);
            void AddParameters(QVBoxLayout* layout);
            void FillParameters() { ev_time->
                setValue(GetEV_Time()); }
            void ToVariables() { SetEV_Time(ev_time->
                value()); }
            QDoubleSpinBox* ev_time;
#endif
```

QDoubleSpinBox* ev_time is a graphical widget responsible for parameter input. FillParameters and ToVariable manipulate with this widget values. PrintConfig is responsible for saving the configuration file with brief explanation of it's meaning:

```
void ModuleEdgeField::PrintConfig(std::ofstream
   &conf_file) {
    conf_file<<"\n#Averaging␣the␣current␣for␣electric␣
        field␣profile␣from␣F_TLow␣to␣F_TLow+EV_Time";
    conf_file<<"\nEV_Time\t=\t"<<GetEV_Time();
}
```

AddParameters(QVBoxLayout* layout) is used to set the layout of the interface:

```
void ModuleEdgeField::AddParameters(QVBoxLayout
   *layout) {
    QHBoxLayout* hlayout = new QHBoxLayout;
    hlayout->addStretch();
    hlayout->addWidget(new QLabel("Integration␣time"));
    ev_time = new QDoubleSpinBox();
    ev_time->setMinimum(0);
    ev_time->setMaximum(1000);
    ev_time->setSingleStep(0.01);
    ev_time->setDecimals(4);
    hlayout->addWidget(ev_time);
    hlayout->addWidget(new QLabel("ns"));
    layout->addLayout(hlayout);
}
```

The desired parameter input is shown in Figure 13. Horizontal Box Layout used to place the labels and a spin box. The CheckModuleData() method is easily
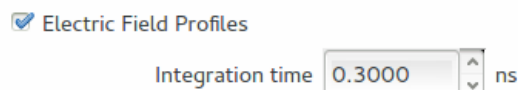

Figure 13: EV_Time parameter input.

understandable and will be skipped in this tutorial. The `Analysis()` method is deeply commented, see **./src/modules/ModuleEdgeField.cc** file.

2. Registering module in the **tct_config.cc** source file.
Following discussion is on the **tct_config.cc** file. Module has to be included at the beginning of the file:

```cpp
#include "modules/ModuleEdgeField.h"
```

When parsing the config file, line to register the module has to be added:

```cpp
if(i.first == "EdgeVelocityProfile")
   RegisterModule(new
   ModuleEdgeField(this,"EdgeVelocityProfile",_Edge,
   "Electric Field Profiles"),
   static_cast<bool>(atoi((i.second).c_str())));
```

Also EV_Time parameter has to be set. It has to be added in the second `for` loop, to avoid setting parameter for the non-registered module.

```cpp
if(i.first == "EV_Time")
   ((ModuleEdgeField*)GetModule("EdgeVelocityProfile"))
   -> SetEV_Time(atof((i.second).c_str()));
```
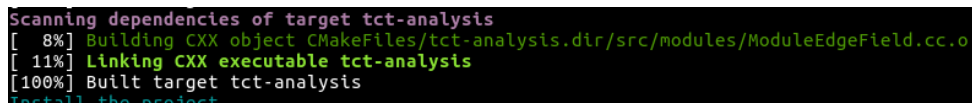
3. Manual adding of the enabling string to the configuration *.txt file.
Has to be done in any configuration file, because module will be registered only if the following line will be present

```
EdgeVelocityProfile = 0
```

Parameter EV_Time does not have to be added, default value $0.3ns$ will be assigned by default.

4. Running CMake inside the **build** directory to add module to the build list. Follow building tutorial in Section 2.1.

5. Code recompilation. Make sure, that needed module was compiled, see Figure 14.

Figure 14: Compilation of new module.

6. Conclusion

To conclude I would like to express my highest gratitude to Hendrik Jansen(CMS, DESY) for proposing the idea of this framework and sharing of his experience in Transient Current Technique. Also I would like to express my gratitude to ATLAS group(LPNHE,

Paris) in the face of Giovanni Calderini, Marco Bomben and Giovanni Marchiori for working with me on establishing TCT test bench at LPNHE and giving an opportunity to continue working on the framework.

In case of additional questions or ideas on improving the documentation (also the framework), please contact me mykyta.haranko@gmail.com

A. Sample configuration file

Here sample TCT-Analysis configuration file is presented. Follow the comments to understand logic.

```
#comments have to start with a
#put group key words in []
#always specify ID, TAB, "=", TAB, value

[General]
ProjectFolder   =           ./
DataFolder      =           ../testdata/lpnhe
Outfolder       =           ../results/

#Set acq mode.
# 0 - taking the sets of single measurements (*.txt or *.raw files by oscilloscope). Settings are in
     [Analysis]
# 1 - taking the data from *.tct file produced by DAQ software. Settings are in [Scanning]
Mode      =       1

[Analysis]
MaxAcqs =         100
Noise_Cut       =         0.005
NoiseEnd_Cut    =         0.005
S2n_Cut =         3
S2n_Ref =         2
AmplNegLate_Cut =           -0.02
AmplPosLate_Cut =         0.015
AmplNegEarly_Cut      =           -0.02
AmplPosEarly_Cut      =           0.02
DoSmearing      =         0
AddNoise        =         0
AddJitter       =         0
SaveToFile      =         1
SaveSingles     =         1
PrintEvent      =         4294967295
LeCroyRAW       =         0

[Scanning]
#Channels of oscilloscope connected to detector, photodiode, trigger.
#Put numbers 1,2,3,4 - corresponding to channels, no such device connected put 0.
CH_Detector     =         1
#Turning on of the Photodiode channel also adds normalisation to all scans
CH_Photodiode   =         0
CH_Trigger      =         0
#Set optical Axis. 1-x,2-y,3-z
Optical_Axis    =         3
#Set scanning Axis. 1-x,2-y,3-z
Scanning_Axis   =         1
#Set voltage source number (1 or 2)
Voltage_Source  =         1
#Time between stage movements in seconds.
Movements_dt    =         1
#Set the integration time in ns to correct the bias line.
#Program averages the signal in range (0,value) and then shifts the signal by the mean value.
CorrectBias     =         5
#Perform next operations. Analysis will start only if all needed data is present:
# 0-top,1-edge,2-bottom
TCT_Mode        =         0

#Scanning over optical and perpendiculr to strip axes
#(or along the detector depth in case of Edge TCT), fitting the best position.
Focus_Search    =         0
#Depletion Voltage
EdgeDepletionVoltage    =         1
#Electric Field Profiles
EdgeVelocityProfile     =         1
#Averaging the current for electric field profile from F_TLow to F_TLow+EV_Time
EV_Time =         0.3
#Depletion Voltage
TopDepletionVoltage     =         0
#Charge Carriers Mobility
TopMobility     =         1

#Integrate sensor signal from TimeSensorLow to TimeSensorHigh - ns
TimeSensorLow   =         10
TimeSensorHigh  =         130
#Integrate photodiode signal from TimeDiodeLow to TimeDiodeHigh - ns
TimeDiodeLow    =         46
TimeDiodeHigh   =         51

#Save charge, normed charge and photodiode charge for each Z, voltage
SaveSeparateCharges     =         1
#Save waveforms for each position and voltage
SaveSeparateWaveforms   =         0

[Parameters]
```

```
#low−field  mobility  for  electrons ,  cm2*V^−1*s^−1
Mu0_Electrons    =        1400
#low−field  mobility  for  holes ,  cm2*V^−1*s^−1
Mu0_Holes        =        450
#saturation  velocity  cm/s
SaturationVelocity       =         1e+07
#  amplifier  amplification
Amplification    =        300
#  factor  between  charge  in  sensor  and  photodiode  due  to  light  splitting :  Nsensor/Ndiode
LightSplitter    =        9.65
#  resistance  of  the  sensor  and  diode  output , Ohm
ResistanceSensor         =         50
ResistancePhotoDetector  =         50
#  pohotodetector  responce  for  certain  wavelength , A/W
ResponcePhotoDetector    =        0.7
#  electron−hole  pair  creation  energy , eV
EnergyPair       =        3.61

[ Sensor ]
SampleCard       =         ../ testsensor /SC_M2015 . txt
```

References

[1] http://www.desy.de/f/students/2015/reports/MykytaHaranko.pdf

[2] http://particulars.si/downloads/ParticularsProcedures-FocusFind.pdf

[3] Green MA, Keevers MJ. Optical properties of intrinsic silicon at 300 K. Progress in Photovoltaics: Research and Applications. 1995 ;3:189 - 192.

[4] Jiaguo Zhang DESY-THESIS-2013-018

[5] G.Kramberger et al, IEEE Transactions on Nuclear Science, vol. 57, no. 4, August 2010.

[6] https://en.wikipedia.org/wiki/Bisection_method