TRLO II User Documentation

Martin Bajzek, Andrea Jedele

M.Bajzek@gsi.de , A.Jedele@gsi.de

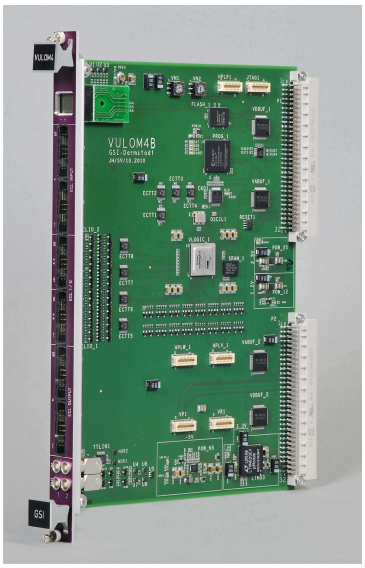
June 3, 2025

Contents

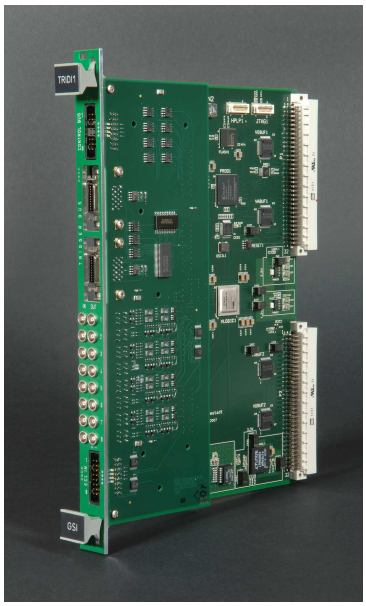
1 Introduction

TRLO II is an FPGA firmware [1] for the GSI modules VULOM and TRIDI. The VULOM and TRIDI modules are primarily used as a trigger logic unit and as scalers. Features include, but are not limited to, scaler monitoring, identifying which input triggered within an event, coincidence logic units, white rabbit scaling, etc. The objective of this manual is to familiarize the user with how to set up a VULOM and common uses.

In all the following examples, all the commands started with `:$>` are to be issued on any Linux PC. Commands starting with `~ >` are to be run on the embedded device, such as the RIO4 VME controller, or a GSI-standard X86L machine connected via USB to the MVLC[10] controller. For all of the commands, VULOM can be replaced with TRIDI. This manual will focus on VULOM, given it is the more common module.



(a) VULOM4b



(b) TRIDI

2 Setting up

This guide walks the users through compiling the TRLO II companion programs, flashing the firmware onto a VULOM/TRIDI module and some initial tests.

Code is hosted with `git`.

```
:$> git clone /u/johansso/trloii.git
```

The upstream repository having the most up-to-date code is `https://gitlab.com/chalmers-subexp/trloii`, which is a private repository. Ask permission to join group.

Enter the `trloii` directory.

```
:$> cd trloii
```

Find out the latest .tar firmware file name by visiting:
`http://fy.chalmers.se/~f96hajo/trloii/firmwares.html`

At the time of writing, it is `f13d071c`. Download the tarball locally:

```
:$> wget https://fy.chalmers.se/~f96hajo/trloii/trloii_firmwares_XXXX.tar.gz
```

where `XXXX` is replaced with the correct firmware hash. Extract the tar in the `trloii` directory:

```
:$> tar -zxf trloii_firmwares_XXXX.tar.gz
```

Enter the `trloctrl` directory and run the Perl script:

```
:$> cd trloctrl
:$> ./find_firmwares.pl
```

Go back to main `trloii` directory and run `make` there.

```
:$> cd ../
:$> make -j4
```

Log in to your embedded device machine (usually the RIO4). Go to the same working directory where the `trloii` was git-cloned.

```
~ > cd trloii
~ > cd trloctrl
~ > make
```

This might take some time. Compile the flash utility next:

```
~ > cd ../flash
~ > make
```

3 UNIX the support software

Once building of `trloii` is done, to have ease of use of various programs, it is common to set certain environment variables [3]. What needs to be known is if the VULOM module in question is of type VULOM4B or just VULOM4.
Go again to the `https://fy.chalmers.se/~f96hajo/trloii/firmwares.html`. Depending on VULOM type in question, in the **File** column, find the rows `vulom4_trlo/vlogic_1.rbt` , or `vulom4b_trlo/vlogic_1.rbt`. Make note of the bold 8 hexadecimal letters at the end of **md5sum** entry `(XXXXXXXX)`. At the time of writing, it is `1409285e` for VULOM4b.

Make a new file, `env.csh` in the main directory where `trloii` is located:

```
:$> touch env.csh
```

Edit the `env.csh` with the following lines, where you change `XXXXXXXX` to a proper hash. Note that `$WORKING_DIR` should point to the working directory. Either type-out the full path for `$WORKING_DIR` or pass `` `pwd` ``, but then sourcing will not work from outside the working directory.

```
setenv ARCH_SUFF `gcc -dumpmachine`_`gcc -dumpversion`
setenv WORKING_DIR `pwd`
setenv TRLOII_PATH $WORKING_DIR/trloii
setenv VULOM4_FW XXXXXXXX
setenv TRLOII_FLASH $TRLOII_PATH/flash/bin_$ARCH_SUFF/vulomflash
setenv VULOM4_CTRL $TRLOII_PATH/trloctrl/fw_${VULOM4_FW}_trlo/bin_$ARCH_SUFF/trloctrl
```

Source the file.

```
~ > source env.csh
```

One can also add the environment variables (but with full path, not `` `pwd` `` for `WORKING_DIR`) to the `~/.tcshrc` script which gets loaded during login. Also an example repository with the `env.csh` and `env.sh` data available. It is in `https://git.gsi.de/r3b/daq/env`.

4 VME base-address

To communicate with any VME module, the embedded controller needs to know the module's VME base-address. For GSI VULOM these are the `VN1` and `VN2` rotary switches on the backside of the module. The base address is then the hexadecimal number: `0x VN2 VN1`. The default configuration from the GSI EE is 0x300000 or `VN2=3` and `VN1=0`. For multiple VULOMs or VME modules, it is imperative that the base addresses are unique.

5 vulomflash utility

The `vulomflash` program allows flashing or inspection of specified firmware stored inside the VULOM's flash memory. Suppose that the base address is the default `3` (`VN2=3` and `VN1=0`) in all the following examples. With the environment variables set up from Section 3 execute:

```
~ > $TRLOII_FLASH --addr=3 --readprogs
```

Identifier of the loaded firmware is found in the rightmost column. If this shows all `Rng` sections empty, or shows wrong firmware version stored in `Rng 0`, then store the firmware (corresponding `.rbf` file) into a free region `N` of the flash:

```
~ > $TRLOII_FLASH --addr=3 --prog=N $TRLOII_PATH/fw/vulom4b_trlo/vlogic_4b.rbt
```

Check that the region `N` is now flashed with a proper image:

```
~ > $TRLOII_FLASH --addr=3 --readprogs
```

Load this firmware into the FPGA:

```
~ > $TRLOII_FLASH --addr=3 --restart=N
```

It is important to first load the new firmware into a prog region that is not 0. Therefore, the progs can be tested before overwriting the default, which is in region 0.

Now `Rng 0` should show the correct firmware md5sum in the rightmost column. If this is not the case then force the flashing into region 0. **Careful with the following command.**

```
~ > $TRLOII_FLASH --addr=3 --prog=0 $TRLOII_PATH/fw/vulom4b_trlo/vlogic_4b.rbt --force
```

Flashing should be done only once, or when the new firmware version gets released.

6 trloctrl program

The main `trloii` program is the `trloctrl`. Start first by poking the VULOM. The following command should not incur a SIGBUS error:

```
~ > $VULOM4_CTRL --addr=3
```

Result should be something like:

```
hwmap_mapvme.c:419: LOG: Virtual address for TRLO II @ VME 0x03000000 is 0x3005e000.
LOG: TRLO: MD5SUM: 0x1409285e (CT: 63bb1d44 = 2023-01-08 19:45:08 UTC)
```

If the message in the last line states

```
signal SIGBUS received.
```

then the VULOM was not acknowledged. The most common issue is either wrong firmware (see Section 5) or wrong base address (see section 4)

Check all the command options with a `--help` flag.

To see real-time updates of scaler values, execute:

```
~ > $VULOM4_CTRL --addr=3 --mux-src-scalers
```

Columns labelled `Value` show the count status of each of the scalers, while the `Diff` columns show increment per second.

More about `trloctrl` in the following sections ...

7 TRLO II syntax

Configurations of connections and internal logic can be loaded either from setup files (conventional filename extension `.trlo`) or issued from the shell. Setup file syntax is as following: all expressions must be contained in various named `SECTION(name)` blocks and must terminate with a semicolon `;` . There are three types of expressions:

- Signal multiplexing, which assigns a source to a destination with the `<=` operator.

```
destination <= source;
```

- Setup register assignment which assigns a value to a certain parameter with the `=` operator.

```
parameter = value;
```

- Alias definition. Declare an additional alias (name) for a specified signal/parameter/number with the `:=` operator.

```
new_name := signal|parameter|value;
```

The control flow inside a `SECTION()` block doesn't exist. All the instructions are executed at the same time (similar to HDL) in the FPGA. Multiple identical/conflicting instructions prioritize the order in which `SECTION`'s are executed, with later setup overwriting the older one. Usually before setting a new configuration, the previous setup should be cleared to avoid conflicts:

```
~ > $VULOM_CTRL --addr=3 --clear-setup
```

Finally, to load a `SECTION(section)` into the VULOM, from a file `example.trlo`:

```
~ > $VULOM_CTRL --addr=3 --config=example.trlo "section"
```

To load multiple sections, just pass them as additional arguments to the `$VULOM4_CTRL` program.

```
~ > $VULOM_CTRL --addr=3 --config=example.trlo "section1" "section2"
```

8 Practical examples

Create a file named `vulom.trlo`.

```
:$> touch vulom.trlo
```

8.1 Pulsers

Suppose we wish to create a 1 kHz pulser on a lemo output of the VULOM. All that is sufficient is to write the following `SECTION` in the `vulom.trlo` file:

```
SECTION(pulser) {
    period(1) = 1 kHz;
    LEMO_OUT(1) <= PULSER(1);
}
```

Execute the section:

```
~ > $VULOM_CTRL --addr=3 --config=vulom.trlo pulser
```

Each signal can either be a source or a destination. Easiest to understand are the `LEMO_IN(1|2)` and `LEMO_OUT(1|2)`. We can send an internally generated signal to a `LEMO_OUT(1)` but not into the input `LEMO_IN(1)`. All output signals from the VULOM are either ECL or NIM type.

The TRLO II in the VULOM runs on a 100 MHz clock, so each pulse will be 10 ns long by default. If we wish to stretch the 1 kHz pulser, we need to route it first into a stretcher. The section above can be rewritten as:

```
SECTION(pulser) {
    period(1) = 1 kHz;
    GATE_DELAY(1) <= PULSER(1);
    stretch(1) = 1 us;
    LEMO_OUT(1) <= GATE_DELAY(1);
}
```

This will output 1 us long pulses, instead of them being 10 ns long. TRLO II hosts 4 independent pulsers, labelled with array indices 1,2,3,4. The firmware also allows one to set the pulser in terms of time (ns, us, ms, s) or frequency (MHz, kHz, Hz). To simulate CFD responses, a periodic pulser is a bad choice. TRLO II however also provides one poisson-like pulse generator.

```
SECTION(pulser) {
    prng_poisson(1) = 1 kHz;
    GATE_DELAY(1) <= PRNG_POISSON(1);
    stretch(1) = 1 us;
    LEMO_OUT(1) <= GATE_DELAY(1);
}
```

8.2 Digital logic

Logic inside VULOM is presented with the Logic Matrix Unit, `LMU`. The entry-point to `LMU` are 8 different (1..8) `LMU_IN(n)` gates. There are 8 different independent outputs (1..8) `LMU_OUT(n)` of the unit.

The Logic Matrix Unit is essential anytime an output from a several inputs is generated. As a basic example, we can take an input and pass it as an output

```
SECTION(pass_signal_along) {
    LMU_IN(1) <= ECL_IN(1);
    LMU_OUT(1) <= LMU_IN(1);
    LEMO_OUT(1) <= LMU_OUT(1);
}
```

Note: The input signal is solely being passed through the module, which is on the time-order of 50 ns. The signal will be passed through even if the module is in deadtime. For deadtime implementation, see Section 10.7

For a more complex example, to make a logical AND of three different ECL inputs:

```
SECTION(and_gate) {
    LMU_IN(1) <= ECL_IN(1);
    LMU_IN(2) <= ECL_IN(2);
    LMU_IN(3) <= ECL_IN(3);
    LMU_OUT(1) <= LMU_IN(1) and LMU_IN(2) and LMU_IN(3);
    LEMO_OUT(1) <= LMU_OUT(1);
}
```

A specific `LMU_OUT` can be composed of only logical AND's or only logical OR's. One cannot mix AND's and OR's in the same unit.

Another way to make coincidence is via the `input_coinc` unit

```
SECTION(and_gate_coinc) {
    input_coinc_mask(1) <= ECL_IN(1), ECL_IN(2), ECL_IN(3);
    input_coinc_level(1) = 3;
    LEMO_OUT(1) <= INPUT_COINC(1);
}
```

The `input_coincidence_level` can be varied to allow a different minimum number of coincidence channels needed. There is only 1 `input_coinc` gate available.

Another example - to make a logical OR of two ECL inputs:

```
SECTION(or_gate) {
    LMU_IN(1) <= ECL_IN(1);
    LMU_IN(2) <= ECL_IN(2);
    LMU_OUT(1) <= LMU_IN(1) or LMU_IN(2);
    LEMO_OUT(1) <= LMU_OUT(1);
}
```

Alternative way is to utilize the `all_or` unit:

```
SECTION(all_or_gate) {
    all_or_mask(1) <= ECL_IN(1) | ECL_IN(2);
    LEMO_OUT(1) <= ALL_OR(1);
}
```

There are 2 `all_or` units available.

9 Other examples

9.1 Generating a long gate

Suppose we wish to create a spill-like emulator with a certain duty-cycle and fan out these three signals:

- `BOS` 200 ns pulse at the beginning of the spill
- `EOS` 200 ns pulse at the end of the spill
- Beam gate

Naive attempt:

```
out_bos          := ECL_OUT(12);
out_eos          := ECL_OUT(13);
out_beam_gate    := ECL_OUT(14);
spill_period     := 3000 ms;
spill_on_length  := 2000 ms;
output_length    := 200 ns;

SECTION(beam_gate_mimic) {
    period(1) = spill_period;
    GATE_DELAY(1) <= PULSER(1);
    stretch(1) = spill_on_length;
    out_beam_gate <= GATE_DELAY(1);

    // BOS
    GATE_DELAY(2) <= PULSER(1);
    stretch(2) = output_length;
    out_bos <= GATE_DELAY(2);

    // EOS
    GATE_DELAY(3) <= PULSER(1);
    delay(3) = spill_on_length;  // oops ??
    stretch(3) = output_length;
    out_eos <= GATE_DELAY(3);
}
```

Oops, doesn't work. Delay time is too big for `EOS`. To remedy it, pass the beam gate into a `GATE_DELAY` that resets on trailing edge. The correct example:

```
out_bos          := ECL_OUT(12);
out_eos          := ECL_OUT(13);
out_beam_gate    := ECL_OUT(14);
spill_period     := 3000 ms;
spill_on_length  := 2000 ms;
output_length    := 200 ns;
```

```
SECTION(beam_gate_mimic) {
    period(1) = spill_period;
    GATE_DELAY(1) <= PULSER(1);
    stretch(1) = spill_on_length;
    out_beam_gate <= GATE_DELAY(1);

    // BOS
    GATE_DELAY(2) <= PULSER(1);
    stretch(2) = output_length;
    out_bos <= GATE_DELAY(2);

    // EOS
    GATE_DELAY(3) <= GATE_DELAY(1);
    restart_mode(3) = TRAILING_EDGE;
    stretch(3) = output_length;
    out_eos <= GATE_DELAY(3);
}
```

9.2 Manual deadtime implementation

Suppose some signal on any of the connections: `LEMO/ECL_IN , LMU_IN/OUT` comes with some random pattern, and we wish to limit the minimum possible interval between two consecutive pulses by adding a quasi deadtime. In this example, the free signal comes into `LEMO_IN(1)`.

```
free_input       := LEMO_IN(1);
deadtimed_signal := LMU_OUT(1);
deadtimed_output := LEMO_OUT(1);
deadtime_length  := 300 us;
output_length    := 200 ns;

SECTION(manual_deadtime) {
    // Pulses that get "accepted" spawn a deadtime gate.
    GATE_DELAY(1) <= deadtimed_signal;
    stretch(1) = deadtime_length;

    LMU_IN(1) <= free_input;     // 'free' signal
    LMU_IN(2) <= GATE_DELAY(1);  // deadtime gate
    deadtimed_signal <= LMU_IN(1) and not LMU_IN(2); // alias for LMU_OUT(1).
    lmu_restart_mode(1) = LEADING_EDGE | GATE_ENABLE;
    lmu_stretch(1) = output_length; // width of the output `deadtimed_signal`

    // Send the output of the logic to the lemo output.
    deadtimed_output <= deadtimed_signal;
}
```

10 Trigger logic

VULOM works in tandem with the TRIVA module to properly deadtime the readout. We can construct complex trigger logic with it as well.

Main components of trigger logic are the *trigger patterns* (TPAT's) and *trigger types*.

10.1 Nomenclature

Free trigger is a trigger request that the DAQ system receives and it either gets accepted or rejected. *Accepted trigger* gets created when the free trigger arrives outside of DAQ deadtime, and then also survives the potential downscale veto. This then prompts the VME controller to initiate a readout.

10.2 Trigger types & trigger patterns

The GSI TRIVA [4] module has four Master-Trigger ECL inputs which allow encoding of 15 different **trigger types**. MBS [5] and drasi [6] VME DAQ systems' user readout code (nurdlib / `f_user.c`) dispatches different readout actions based on the received trigger type. In other words, one can in this way select which readout actions to perform if prompted by different trigger types. Usually, trigger type=1 is reserved for 'physics' triggers, when the whole system gets readout. Then types 2, 3 are related to synchronisation procedures which are user-defined. Types 12, 13 by convention we reserve for beginning-of-spill and end-of-spill triggers, respectively. Types 14, 15 are software generated and shall always be reserved for beginning of acquisition and end of acquisition.

The **Trigger pattern** (from now on called **TPAT**) is a feature of TRLO II and adds another layer of distinction between triggers, even for the same trigger type. If multiple different free triggers arrive within a specific `accept_window_len` window relative to the first accepted trigger, and survive downscale, then these additional triggers also get encoded into a trigger pattern word which the user can read out[1] from the VULOM. Sixteen different TPATs are available and their status in an event will be encoded in lower 16 bits of the TPAT word. If a trigger assigned to TPAT=N gets accepted within the acceptance window, then the N-th bit of the TPAT word shall be `1`, otherwise it is `0`. Which in pseudo-code would mean:

```
TPAT_STATUS[N] = (TPAT_WORD >> (N-1)) & 1 , where N=1,2, ... ,16
```

If TPAT's mapped to different trigger types arrive within the same `accept_window_len` time and survive downscale, then the trigger type of the accepted trigger will be the highest type received during this window.

Exit point of the `TRIG_LMU` trigger logic matrix of the VULOM are the 16 gates `TRIG_LMU_OUT(n)` which represent each of the TPATs in order. Each of `ECL_IN`s of VULOM can be directly sent to any of the `TRIG_LMU_OUT(n)`. To map any other different signal, *e.g.* an `LMU_OUT(n)`, one first has to send it to a `TRIG_LMU_AUX(m)` gate, where m=1,2,3,4, and then use the aux to input of the trigger logic matrix.

In the simplest example, one can map each of the `ECL_IN`s to a TPAT in order. Then assign them all to the physics trigger type=1

[1] Check section 13 for readout guide.

```
SECTION(trigger_matrix) {
    // Set up the TPAT's:
    TRIG_LMU_OUT(1) <= ECL_IN(1);
    TRIG_LMU_OUT(2) <= ECL_IN(2);
    TRIG_LMU_OUT(3) <= ECL_IN(3);
    /* and so on ... */

    // Map the TPAT's to the trigger type:
    // tpat_trig (TPAT) = trigger_type ;
    // Remember not to map it to trigger_type=14,15 , these are reserved!
    tpat_trig(1) = 1;
    tpat_trig(2) = 1;
    tpat_trig(3) = 1;
    /* and so on ... */

    // Lastly, enable the tpats, with the specific += operator
    // Alternative syntax: tpat_enable = mask 0xffff;  with `mask` keyword
    tpat_enable += 1;
    tpat_enable += 2;
    tpat_enable += 3;
    /* and so on ... */

    // Want to downscale a TPAT?
    // Syntax: tpat_red(TPAT) = r;
    // Downscale factor is 2 to the power of 'r'. r=0 means no reduction
    tpat_red(1) = 0;
    tpat_red(2) = 0;
    tpat_red(3) = 0;
    /* and so on ... */
}
```

TPATs can be turned off, such that they do not get encoded nor can they then create an accepted trigger, with the `-=` operator. This is analogous to how the `+=` operator enables them, as shown in example above.

10.3 Trigger LMU input delays and stretches

More complex trigger selection requires coincidences to enter as TPATs. Combining signals to `TRIG_LMU_OUT(n)` allows coincident logic (with `and` , or `&` operator) between different inputs to the gate. Sometimes the inputs to a specific `TRIG_LMU_OUT(n)` arrive with a delay relative to each other, or with a significant jitter. TRLO-II allows stretching of these inputs, so that coincidences can be achieved more consistently. Each of the `ECL_IN`'s can also be delayed before entering into the coincidence.

Let's have an example. Suppose `ECL_IN(1)` and `ECL_IN(2)` are CFD discriminated outputs of two PMTs of two separate plastic scintillators. The time-of-flight between them is on average 100 ns, with a 10 ns jitter. Suppose `ECL_IN(1)` arrives 100 ns earlier. Instead of sending the inputs to different `GATE_DELAY`s, we can align the coincidence like this:

```
SECTION(trigger_coinc) {
    // Delay ECL_IN(1) to coincide in time with ECL_IN(2)
    // Number `1` in brackets is associated to a replica of ECL_IN(1)
    // which enters the trigger logic
    trig_delay(1) = 100 ns;

    // Stretch both ECL_IN inputs to the trigger logic to mitigate the jitter
    trig_stretch(1) = 100 ns;
    trig_stretch(2) = 100 ns;

    // Require a coincidence to fire the TPAT=1
    TRIG_LMU_OUT(1) <= ECL_IN(1) and ECL_IN(2);
}
```

10.4 Trigger alignment

10.4.1 Data intake and analysis program

The trigger alignment is done using the tracer feature, which acts as a soft-scope for the `TRIG_LMU_OUT` inputs. It samples a 512-entry buffer (*i.e.* a 5120 ns time period) around internal triggers. The internal soft-scope triggers are cycled to look for all possible pairs of coincidences, as well as individual signals.

```
~ > $VULOM_CTRL --addr=3 --tracer > tracer001.txt
```

The `tracer001.txt` is an arbitrary name, which has to be changed every time the tracer recording is run again. Otherwise, an error message will appear and the program does not execute.

To end the program, press ctrl+c, making sure to collect enough statistics on each input.

To analyze, run the analyse.bash script using the command

```
:$> ./analyse.bash tracer001.txt
```

analyse.bash has the following input:

```
in=$1
out=${1}.analysed

$TRLOII_PATH/bin/align_analyse < $in > $out
less $out
```

Replace `tracer001.txt` with the name of you input file. The output file is generated in a file tagged with `.analysed` after the input name.

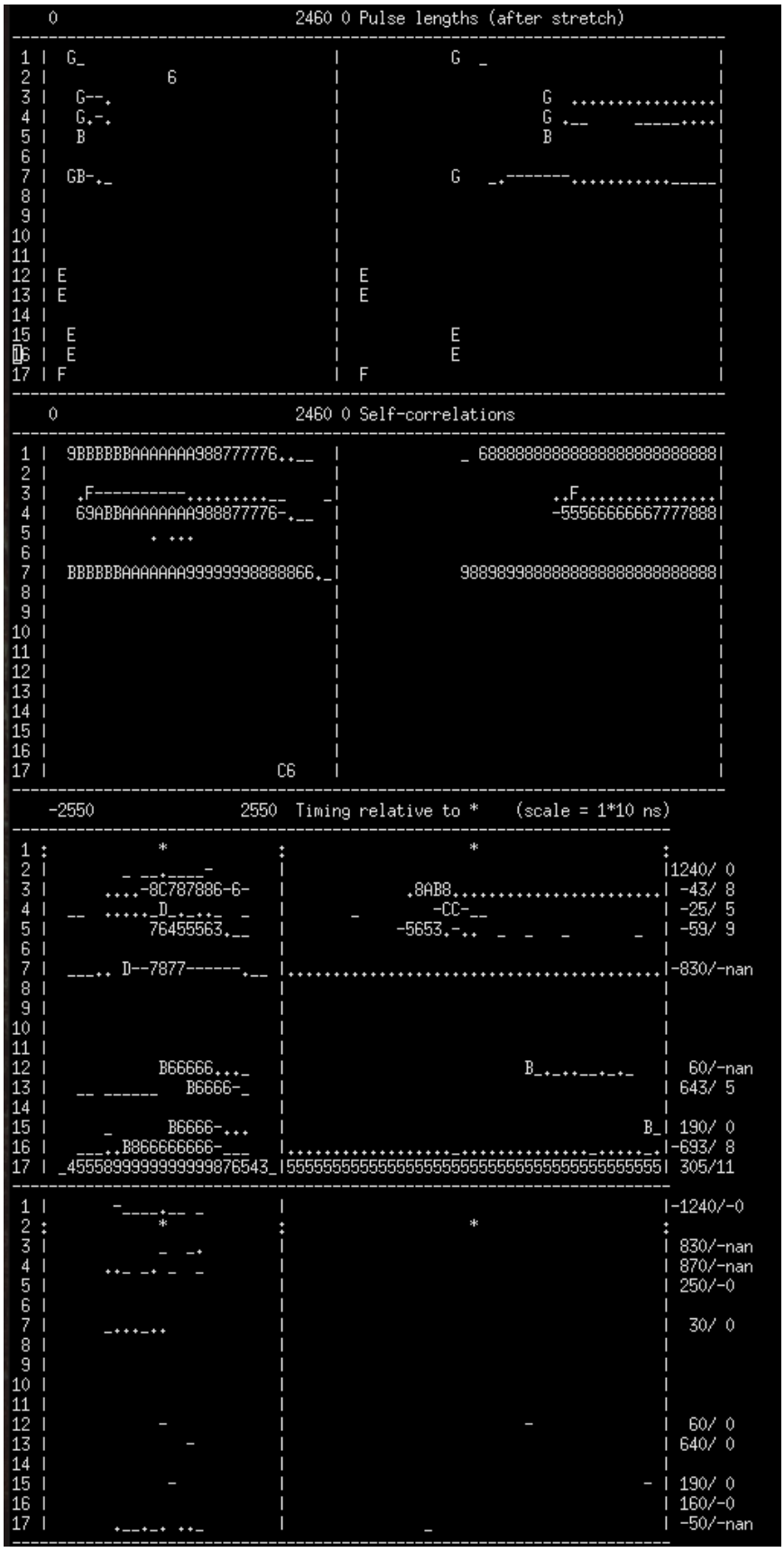


Figure 2: Abbreviated output from analyse.bash in tracer001.txt.analysed.

10.4.2 How to read the output

The output is seen in Figure 2. The 1st section labelled `Pulse lengths` corresponds to the length of the trigger sampled. This should be equal to the value set for trig_stretch(#), where # is the ECL input.

The second section, labelled `Self-correlations` corresponds to the trigger timing relative to other signals of itself. It is good to make sure the timing of the output is appropriate and that there is no 'ringing' signals (*i.e.* double triggers) which can ruin downscale counts and thus, cross-section measurements.

In the 3rd section, we note that rows 1-17 are labelled twice. In the program this is repeated 17 times [2] corresponding to the trigger alignment relative to the channel with the `*` in it. Channels 1-16 correspond to `ECL_IN(1..16)` and 17 corresponds to `LEMO_IN(1)` that enter the trigger logic matrix after delaying and stretching.

In each section, values are plotted on the left and right hand side, separated by the middle vertical | line. The values to the left are zoomed-out values. The scale corresponds to the values on the top left of the header. For example, in the 1st section labelled `2460 0 Pulse lengths (after stretch)`, there is a 0 and 2460. The 0 corresponds to the start of the timing and the 2460 is the width of the total gate in ns. There are 31 bins. The width of each bin is

$$\frac{2460\,\text{ns}}{31\,\text{bins}} \sim 80\,\text{ns} \tag{1}$$

For the 3rd section on, the number of bins is 25. Here the scales goes from -2550 to 2550 (indicated in the top left header). Therefore, each bin corresponds to approx. 200 ns.

The number, letter and character combinations corresponds to the relative number of trigger samples in binary logarithmic form. In other words, a value of G corresponds to approx. 2^G (or 2^{16}) events. **Be careful, these values are not absolutes, but relatives.** The lowest values 2^0, 2^1 and 2^2 is represented by a _, followed by a . and then a -. Generally, the values with at least numbers are generally good values.

The second set of numbers to the right of the vertical line | corresponds to the zoomed-in values. The scale here is 10 ns. Starting in the 3rd section, it is labeled to the right of the label (scale = 1*10 ns).

10.4.3 Setting the correct stretch and delay values

The first question to consider is what is the minimum trigger condition of the system? For example, if we have a start scintillator, chances are all events must have that start scintillator as a minimum triggering condition. The alignment of all other triggers should be done relative to that signal.

Let us assume here that the minimum bias condition is in `ECL_IN(1)`. Therefore we will use the first set of 17 values in section 3 for the trigger alignment, relative to `ECL_IN(1)`.

The first step is to ensure that the trigger window is correct. In the example shown in Fig. 2, let us look at rows 3, 4 and 5. Start by looking at the values on the left-hand side of the vertical line |. When we look at input 3, there are a bunch of numbers and one `C`. The `C` is the most important value since the bin contains 16 times more triggers than the surrounding bins. When we view the zoomed-in values on the right, we see an 8, A, B and 8 preceded and followed by a few periods. This means that the trigger is the width of the number and letter combination with

[2] The figure shown is only a screenshot, for legibility purposes only the trigger alignment for the first two ECL_IN inputs is shown.

a few spurious triggers that fall outside this window. Since every bin is 10 ns, we can assess that the trigger needs to be stretched at least 40 ns. Since we have no other set of triggers collectively appearing, we can stretch the trigger to ensure we are capturing more events[3]. Here, a value of 100 ns is acceptable. The same applies for trigger 4 and 5.

To stretch the trigger window, change the corresponding `trig_stretch(i)` to the respective value.

The next step is to determine the delay values. Here, it is useful to look at the value on the far right, which is the average offset and its standard deviation. For the third, fourth, and fifth triggers, we see that the trigger is within 100 ns of the relative trigger. Since the first trigger should exist in all of our trigger combinations, we would like to ensure that the other triggers come before the trigger for the minimum triggering condition is set. Therefore, we would like the average to be negative.

It is also important to ensure that the widths of the triggers overlap. For example, if the signal is stretched 100 ns and the offset is greater than 100 ns, the signal will come and disappear before the minimum trigger condition is met. In that case, the VULOM will acknowledge two separate hits with two separate trigger patterns, or the latter one will just get deadtime vetoed.

In general, a delay value of 0 is not desired. Therefore, a minimum value of 10 ns should be used. The slowest trigger should be placed at 10 ns and all other triggers should be delayed relative to that.

To change the delay, change `trig_delay(i)` to the appropriate value.

This process is usually run several times. It is also encouraged to use this tool during dry runs or on a small set-up, to become familiar with how to use it.

10.5 Pending triggers

Triggers that the DAQ is forced to accept are called *pending triggers*. Pending triggers encode `TPAT=0` and are used to accept rare events such as beginning and end of spill triggers, synchronisation and diagnostic triggers. To prompt a pending trigger type `n = 12`, for example, the syntax is the following:

```
TRIG_PENDING[12] <= signal;
```

where `signal` can be any `ECL_IN`, `LEMO_IN` or even a mux output, such as `PULSER` or `LMU_OUT`. If multiple trigger types are pending, then the highest one will have priority. For example, the following sequence:

```
period(1) = 1 Hz;
TRIG_PENDING[5] <= PULSER(1);
TRIG_PENDING[6] <= PULSER(1);
TRIG_PENDING[7] <= PULSER(1);
TRIG_PENDING[8] <= PULSER(1);
```

will result in a series of 4 consecutive events of trigger types: $8 \Rightarrow 7 \Rightarrow 6 \Rightarrow 5$ every one second, regardless of the current TPAT encoding or DAQ status, if there are no higher pending triggers. If a pending trigger falls within the DAQ deadtime, or while the TRLO II is already processing TPAT's via the fast-path, then that pending trigger will get accepted the next time TRIVA releases its deadtime.

[3]If the trigger window is too large, then there is a possibility for random triggers to enter instead of an actual event of interest. This will cause the trigger to be less selective

10.6 Outputting the accepted trigger

Once the trigger survives first the deadtime veto and then the downscale veto, it is accepted. To output the accepted trigger out of the trigger logic matrix, TRLO II offers the following signals:

- `sum_out_mask` , fast master start signal.
- `MASTER_START` , generated slower master start signal.
- `ACCEPT_TRIG(i)` , fires only if trigger type = `i` got accepted.
- `ACCEPT_PULSE` , 1-cycle (10 ns) pulse after an accepted trigger.

`sum_out_mask` is the preferred option as it is the fastest pulse that gets sent out of the trigger matrix, once the trigger decision has taken place.
NOTE 1: unlike other signals, `sum_out_mask` is mapped with the right arrow `=>` operator. By default the signal length is 10 ns. Stretching is possible with the `sum_out_stretch` setup register.
NOTE 2: if a pending triggers are used, then only `ACCEPT_PULSE` will output also the accepted triggers generated by the pending requests.

10.7 TRIVA interfacing (`No touchy!` section)

Finally, to enable communication between VULOM and TRIVA, a consistent cabling scheme has been adopted. Connect upper ECL_IO (9-16) of VULOM to ECL_OUT of TRIVA, and also lower ECL_IO (1-8) of VULOM to ECL_IN of TRIVA. The following block must be present and consistent:

```
SECTION(triva) {
    fast_busy_len = 1000ns;
    DEADTIME_IN(1) <= ECL_IO_IN(4);
    ECL_IO_OUT(1) <= ENCODED_TRIG(1);
    ECL_IO_OUT(2) <= ENCODED_TRIG(2);
    ECL_IO_OUT(3) <= ENCODED_TRIG(3);
    ECL_IO_OUT(4) <= ENCODED_TRIG(4);
    accept_window_len = 100 ns;
}
```

with the possibility to only adjust `accept_window_len`. DAQ deadtime can also be monitored with the `DEADTIME` signal which can be directly sent to any `ECL_OUT` or `LEMO_OUT`.

11 Using the command line

11.1 Setting commands

The same commands discussed in the above sections can also be implemented using the command line. For example, to add a pulser

```
~ > $VULOM4_CTRL --addr=3 "LEMO_OUT(1)=PULSER(1);" "period(1)= 1 kHz;"
```

Note: This will overwrite the current configuration. If a file is loaded and sections executed, then the input from the file or command line will overwrite parts of the current configuration.

11.2 Monitoring tools

There are several features `$VULOM4_CTRL --help` that are of particular interest.

- `--clear-setup` ⇒ clears the current setup.
- `--print-config` ⇒ prints the configuration (ECL_IN, ECL_OUT, ALL_OR, GATE_DELAY, etc.). If the config is not cleared after restarting a VME crate, the default values (ECL_IN(1)) are set.
- `--trig-status[=n]` ⇒ prints every `n` seconds (by default, every 1 s) the TPAT status and scalers. The values are displayed before and after deadtime, and after reduction (if applied).
- `--mux-src-scalers[=ptn]` ⇒ prints the VULOM scalers counting signals at all inputs and from all internal units updating every 1 s. Convenient to see what features are available. The `=ptn` flag prints an `*` next to active input/outputs. Allows the users to easily recognise if an input is perpetually on.
- `--alias` ⇒ always used in conjunction with another argument. Aliases the name of the channels when reading in `mux-src-scalers`. Used for multidetector experiments.

For example:
```
~ > $VULOM4_CTRL --addr=3 --alias=losnrolu="ECL_IN(1)" --mux-src-scalers{=ptn}
```
To exit any monitoring tool, press ctrl+c.

12 Sampler utility

TRLO II also allows sampling of ECL or LEMO inputs on a VULOM, and plotting the timing differences between consecutive hits. This gives a rather straightforward microspill structure spectrum. To sample an `ECL_IN(1)` and save to a local file `local.smpl` execute:

```
~ > $VULOM4_CTRL --addr=3 --sampler=ecl-in-1 > local.smpl
```

Print the file on the screen:

```
$> cat local.smpl
```

The upper drawing represents the Δt distribution between two consecutive signals, in log-log scale. The bottom drawing represents the ratio (in logarithmic scale) of the sampled distribution time difference bin and the corresponding bin that an ideal Poissonian distribution with the same mean rate would produce. Therefore, if the sampled distribution is a true Poissonian, then the bottom curve would be flat.

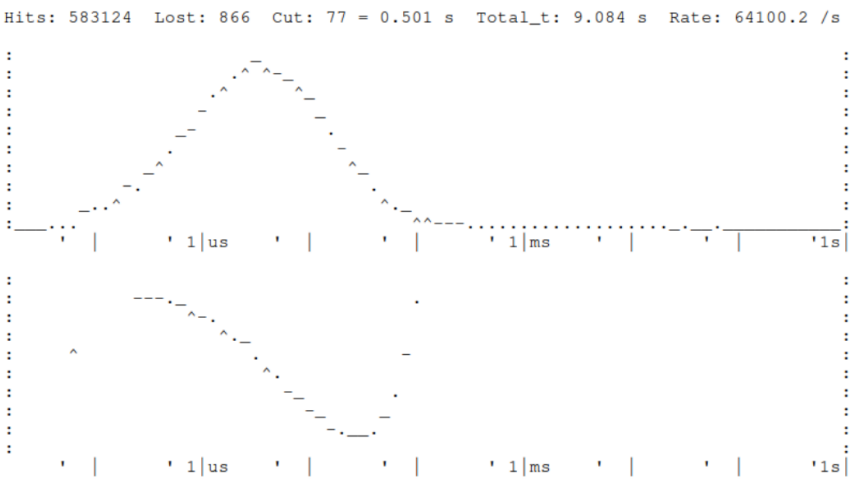


Figure 3: Microspill structure represented as an ASCII art.

13 TRLO II readout

In all the previous sections, we have mainly the interface guide to the `trloctrl` program and explained the corresponding functionalities of the module. VULOM/TRIDI with the TRLO II firmware can also serve as a readout module. There are multiple output registers that can be read out, which correspond to different scalers, timestamps, FIFO buffers (multi-entry buffers) or status registers. These can be latched into the output buffers either *on-event*, such as accepted trigger creation, or they can be *pulsed* via VME. Pulsed means that during the readout of the module, a single-cycle VME write to the pulse register with the offset `0xa000` and the appropriate bitmask, makes the latch. A complete list and description of the output registers (in beige) is given on the main documentation website [1]. An example of a minimal C code to read out certain registers is given in: `https://git.gsi.de/r3b/daq/krab/-/tree/sync_check`. The relevant files are: directory `sync_check`, `Makefile`, `f_user.c`. Such bare code is commonly not presented to the users as there is a standard readout library - *nurdlib*[9].

14 MVLC-based system (GSI-EE adaptation)

Mesytec[10] MVLC controller (via USB2/3) can also serve as an interface to TRLO II. The TRLO II wrapper for GSI x86l- DAQ machines is hosted by the GSI EEL group and standard aliases are provided upon login. At the time of writing, this feature is supported only by the version 7x (development version) of GSI MBS. The standard TRLO II path for GSI X86L- Debian 11/12 machines are provided upon login. The TRLO II firmware corresponding to version `1409285e` or higher is required. Software works stably for MVLC firmware releases `0039` and higher.

All the `trloctrl` utilities are supported, although with the older syntax where all the operators `=>`, `<=` and `:=` are to be replaced with the simple `=` symbol.

14.1 Examples

An example of a TRLO II setup (`.vme`) file is given in: `https://git.gsi.de/m.bajzek/frs-daq/-/blob/sec_s150apr25/mvme/workspaces/frs_daq/vulom.vme`. The different project based on TRLO II with MVLC is `https://git.gsi.de/m.bajzek/microspill`.

14.2 Circumventing DAQ readout access limitations

All the commands work fine while the data acquisition process isn't running. When the acquisition runs, the readout process asserts the USB bus and no VME reads or writes are possible by other programs. To access the `--mux-src-scalers` and `--trig-status` utilities of `trlo_ctrl`, the scalers are read out as a different trigger type, converted into standard GSI LMD data stream, which then the *ucesb*[8] unpacker catches and serves to a standalone program. The project and the corresponding `.vme` file are hosted on `https://git.gsi.de/m.bajzek/mvlc-trloii-unpacker`.

14.2.1 Acessing VULOM registers during MBS acquisition

When the MBS acquisition process `m_read_mvme` runs, the read/writes can still be issued from the MBS dispatcher via `write mvlc` or `read mvlc` commands. More documentation on [11], p13. The commands are issued one at a time within the event readout process, after the event data has been processed from the single USB readout. As such, different VME writes can be saved into a `.scom`[5] sequence file and issued together to the dispatcher. This is commonly used to change the trigger configuration, without the need to restart the acquisition, as used by the *FRS Trigerbox* program: `https://git.gsi.de/m.bajzek/frs-daq/-/tree/sec_s150apr25/scripts/trigcontrol` which creates a `.scom` file that writes to the `trig_enable` and `trig_red(j)` registers the corresponding bitmask and values. An example of a compiled sequence file:

```
write mvlc 0x03000010 0
write mvlc 0x030096ec 0x7fff
write mvlc 0x030096ac 1
write mvlc 0x030096b0 0
write mvlc 0x030096b4 0
write mvlc 0x030096b8 3
write mvlc 0x030096bc 0
write mvlc 0x030096c0 0
write mvlc 0x030096c4 0
write mvlc 0x030096c8 0
write mvlc 0x030096cc 0
write mvlc 0x030096d0 0
write mvlc 0x030096d4 0
write mvlc 0x030096d8 0
write mvlc 0x030096dc 0
write mvlc 0x030096e0 0
write mvlc 0x030096e4 0
write mvlc 0x030096e8 0
```

, where the first write is a dummy write.

15 Credits

All credits go to Håkan T. Johansson, developer of the TRLO II firmware and associated programs. A much more detailed description of the framework is given in his webdocs [1], alongside different examples and presentations. Credits to Jochen Frühauf, who provided the source and helped to understand the original TRLO gateware. Credits to LeCroy for the design of the logic matrix [7] CAMAC module 2365 which served as inspiration for the TRLO and TRLO II projects. We would also like to acknowledge the contribution of Jörn Adamczewski-Musch who ported the TRLO II companion programs and readout libraries to the MVLC USB-based controller platform. Big thanks to our seniors, the original users of the firmware: Hans T. Törnqvist, Bastian Löher, Stephane Pietri and Haik Simon; whose examples and configurations we inherited and studied. Thanks to Philipp Klenze as a colleague, contributor and current main user of TRLO II alongside ourselves.

References

[1] H. T. Johannson - *TRLO II - flexible FPGA trigger control*, `https://fy.chalmers.se/~f96hajo/trloii/`

[2] mesytec GmbH & Co. KG - *MVLC - Low latency VME controller and Trigger module*, Release 1.14.1.1

[3] How to Set and List Environment Variables in Linux, `https://linuxize.com/post/how-to-set-and-list-environment-variables-in-linux/`

[4] J. Hoffmann, N. Kurz, M. Richter, *TRIVA, VME Trigger Module*, `https://www.gsi.de/fileadmin/EE/Module/TRIVA/triva7_1.pdf`

[5] R. Barth et. al., *GSI Multi-Branch System User Manual*, `https://www.gsi.de/fileadmin/EE/MBS/Gm_mbs_i_2.pdf`

[6] H. T. Johansson - *[drasi] — data acquisition*, `https://fy.chalmers.se/~f96hajo/drasi/doc/`

[7] Teledyne LeCroy, *2365 Octal Logic Matrix*, `https://www.teledynelecroy.com/lrs/dsheets/2365.htm`

[8] H. T. Johansson - *ucesb*, unpack & check every single bit, `https://fy.chalmers.se/~f96hajo/ucesb/`

[9] H. T. Törnqvist et. al., *NUstar ReaDout LIBrary - Nurdlib*, `https://web-docs.gsi.de/~land/nurdlib/`

[10] mesytec - MVLC VME Controller - `https://www.mesytec.com/products/nuclear-physics/MVLC.html`, mesytec.com

[11] J.A. Musch, N. Kurz, S. Linev, *GSI Data Acquisition System MBS Release Notes V7.0*, `https://www.gsi.de/fileadmin/EE/MBS/gm_mbs_rel_70.pdf`