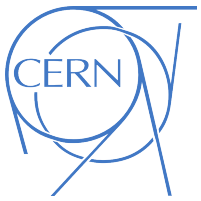

New Interface for ATLAS L1 Trigger Menu Testing

Summer Student Project 2023

Jan Kočka, supervised by Ondřej Penc

1 Introduction

The ATLAS trigger system has the important purpose of reducing the data flow from the experiment to the data center while preserving anything interesting. This is important as with LHC proton-proton collisions, there are hundreds of million events per second, even if storing each event took only a couple of kilobytes, that would end up in us producing several petabytes per day. With runtimes measured in weeks and months and these being underestimates this is a problem.

The ATLAS trigger currently consists of two main stages, the Level-1 (L1) trigger and the Higher-Level-Trigger (HTL). [1] Combined, the two stages reduce the rate from the mentioned $\sim$ 100 MHz to $\sim$ 100 Hz[2].

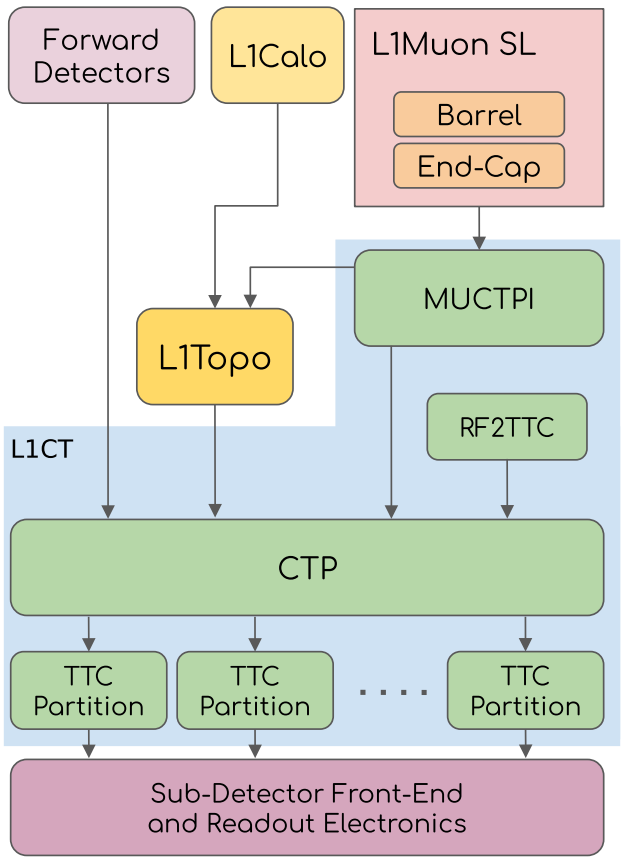

Figure 1: Overview of the ATLAS L1 trigger, my project and this report only concerns the area highlighted in blue labeled L1CT. Anything before or after is treated as input and output.

[1] Though the HLT is further subdivided, and for the LHC High-Luminosity upgrade a new L0 trigger will be introduced[1].

1.1 ATLAS L1 Central Trigger (L1CT)

The L1 trigger is intended to reduce the rate from ~ 40 MHz to ~ 1 kHz using specialized hardware (FPGAs) to handle the high input rate. It consists of multiple components (see fig. 1), the L1 Calorimeter Trigger (L1Calo) and L1 Muon Trigger (L1Muon SL) gather (a part of) the data from detector. That is then handed to the L1 Central Trigger (L1CT) (a group of devices, highlighted in fig. 1) which makes the decision on whether to pass the event to HLT and so possibly keep it.

L1CT contains the logic and central control of the L1 trigger, it is further subdivided into a couple of components (described later) and its main output is an L1 Accept (L1A) signal. This consists of 512 boolean values, called *trigger items*, each representing a particular type of event that we accept, if any are true the event is passed to HLT.

The MUCTPI, which has recently been upgraded [3], prepares the muon information for the CTP. The CTP is another collection of devices which does the final trigger action. Its output is the final L1A, and its input is a collection of *threshold* values, these are boolean values which represent physical conditions. For example `2MU3V` and `3MU3V` each mean that there are two and three muons with transverse momentum over 3 GeV.

As the CTP only does the final, relatively simple logic, in order to have more complex trigger conditions, the L1 Topological Processor (L1Topo) also takes the inputs from L1Calo and MUCTPI and provides advanced topological conditions to the CTP. The CTP also provides other crucial functions, like providing timing signals to all the L1 devices, dealing with busy signals from the detectors etc.

1.2 Trigger Menu Testing

As the L1 trigger is essential to the operation of ATLAS, the L1CT team has a duplicate setup of the L1CT in a lab to test any new development or setup before using it at ATLAS. My project was to work on what is called the trigger menu testing, trigger menu referring to the physics configuration of the trigger (for example accept any event with 3 muons).

These are comprehensive tests of the L1CT, we fully setup the hardware (in the lab), load generated inputs and check the L1A output, along with some intermediary states. This is a check that the trigger menu obtained from the physics team does what was intended, any new menu is tested before being used at ATLAS Point 1 (P1). This also checks the menu compilation process (handled by the L1CT team) which converts a menu specification into hardware configuration files. Finally, it also verifies that the hardware is correctly setup (the process involves a large amount of wires). Though this is currently not as useful, as we only run the tests in the lab, with some further work this may be used to test the setup at P1 too.

Figure 2 shows a diagram of the L1CT devices that are a part of the trigger menu testing. As a final bit of hardware explanation, the CTPIN devices on the diagram are multiplexers that pass through some of the thresholds from L1Calo. [1] There are currently 4 types of tests, all following a similar process and typically all ran just after each other. These each test a group set of trigger items which use a particular set of the CTPCore input signals, they are as follows:

- PIT tests the inputs that go through the CTPIN multiplexers.

[1] This is needed as the CTPCore has a limited number of inputs

- DIR tests the direct electrical inputs into the CTPCore (red lines in fig. 2)
- OPT tests the optical inputs which are the remaining red input lines going to CTPCore.
- MUO tests the inputs coming through the gray lines from MUCTPI to the CTPcore (along with the MUCTPI)

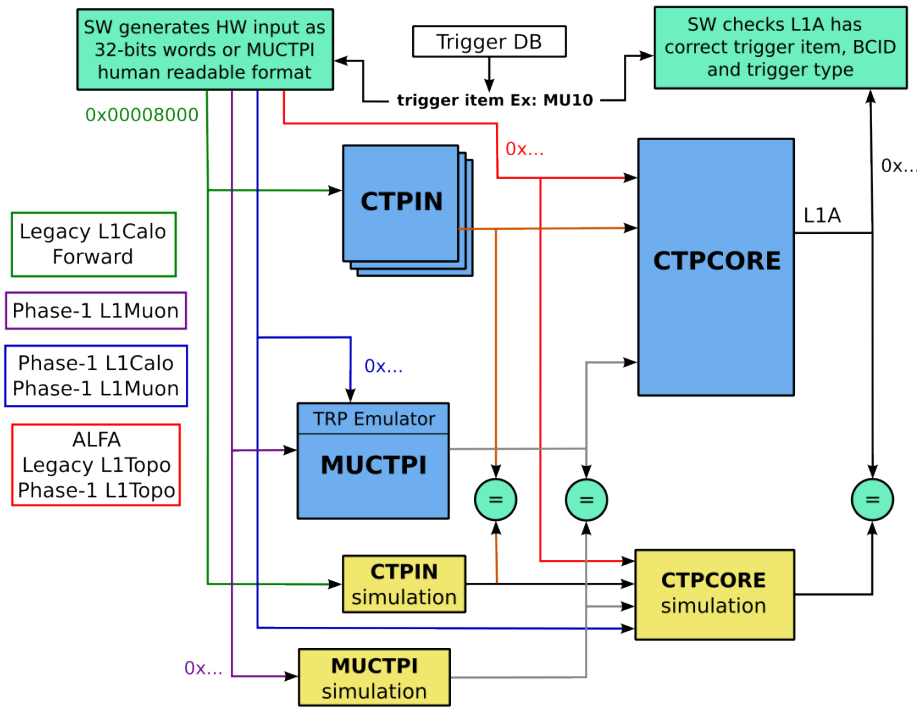

Figure 2: Diagram of the components tested during trigger menu testing. Shows the flow of information during the test and how exactly the hardware components are connected. The boxes on the left list what types of information are sent over the connections.

The actual testing process consists of the following steps, generate the inputs (also referred to as patterns) for a particular menu. Prepare the hardware (different components needed for different test types) and load the patterns. Then finally obtain the output and some intermediary states[1] and evaluate the results against what is expected.

Up until now this process was done by one of a couple of experts familiar with the procedure by running $\sim$ 20 commands or so and a Python script according to a manual. This worked well, though required said experts, and the process is quite repetitive and so suitable for being automated. This is what my project was, developing a new software to run the tests in a user-friendly, reproducible and flexible manner, along with optimizing it where possible (mainly the original Python code).

[1]These are compared against simulations of the hardware, as visualized in fig. 2 using '=' symbols.

2 The New Software

A key part of the software is that most of the hardware is controlled using executables on a couple of computers connected to a network. As such the new software had to be well suited to running external commands and be transparent about it so that the process is debuggable when things go wrong. Managing this well, along with the software being user-friendly (and developer-friendly) were the main two goals of the project. Next in line was a configuration system, as there is some variability mainly concerning how the hardware is setup and what is to be ran. Before, most options were hard coded and described in the manual, so instead providing a unified configuration which is used throughout the testing process (both when generating inputs and when running the tests) was important.

2.1 Using the Software

To demonstrate how the software is used I describe some of the key parts of running it, for more details see the `README` document in the git repository[4]. The software provides a single executable `TriggerMenuTestScript`. As the software implements multiple related but separate functions, they are separated into what is called *actions* (see listing 1). A list of actions (or a string of letters each representing one action) is the only required argument to the software. From a developer's view, these are each implemented by a single function taking two arguments (see section 2.3) and more can be easily added.

- `get-menu`: This downloads the specified menu's JSON files using `TriggerMenuRW` to the path specified by `menu.dir`, it uses Athena for it too.
- `compile-menu`: This requires the menu files from `get-menu`. It uses the trigger menu compiler bash scripts in `.../menuCompiler/run3/` to compile the menu (in two steps) and prepare the files for upload. Creates the `compilation.dir` directory.
- `upload-menu`: Copies the files from `compile-menu` to P1 using rsync and prints instructions for finishing the upload to database manually.
- `generate-patterns`: Generates the input patterns using C++ code from the repository, downloads hardware files from P1 database and runs a simulation for the MUO test. These are stored in the directories as specified by the `patterns` config section.
- `init-test-dir`: This creates the test directory (as specified by `test.dir`), this mainly consists of copying files from other projects in the L1CT release and preparing a config file to be used by `testCtpcorePlusReadout` to get the final test readouts of all the test types.
- `init-hardware`: Fully prepares the hardware, resets the CTPCore and its timing.
- `run-tests`: Runs all the specified tests, this is the most complex action, for more detail see the code in the `run_tests.py` file.
- `evaluate-tests`: Evaluates all the tests, this is fully implemented in Python. It processes files in the `test.readout_dir` directory (created during `run-tests`) to print results of all the tested items and a summary.

Listing 1: A list of all the implemented actions, the mentioned options come from the configuration as covered later.

There are also *special actions*, instead of providing a list of actions, the user can specify

one of these special actions along with any arguments for it. These implement various auxiliary functions to make it easier to use the software – among others the `clean` special action removes previous logs or the more complex `run-subscript` special action can run a bit of Python code used during the testing process if one wants to do a test manually.

2.2 Configuration

As mentioned an extensive, unified configuration system was one of the key features of the new software. In order to satisfy that and still maintain ease of use we opted for a joint configuration file and command line argument system. The software internally has a `default_config.json` configuration file which has all the configuration options in it with their default values, see listing 2. JSON format was chosen as it is used elsewhere in ATLAS TDAQ and as it is possible to group configuration options as needed.

```
{
    "database": "TRIGGERDB_RUN3",
    "smk": -1,
    "tests": "all",
    ...
    "log": { ... },
    "system": {
    "project_data_dir": "./data",
    "project_scripts_dir": "./scripts",
    ...
    },
    "menu": {
        "dir": "./menu",
        "athena_version": "Athena,23.0.33"
    },
    "compilation": { ... },
    "patterns": { ... },
    "test": { ... },
    "evaluation": { ... },
    ...
}
```

Listing 2: Structure of the configuration JSON file

The user can then provide another configuration JSON file (either using the `-c` command line argument or placing it under `./config.json`) which can override any of these. Further, all of these configuration options can be further overridden from the command line arguments (according to their type). All of them are present in the `--help` message with their names being slightly changed (underscores and dots representing nesting are replaced by '-' symbols) to keep with how command line arguments are typically formatted.

To further touch on ease of use and the developer's point of view, a key part here is that all of this functionality is implemented generally, only using the mentioned default configuration. This means that if a developer wants to add a new configuration option, all they need to do is add it to the default configuration with the right type (boolean,

number or string) and everything else is taken care of. The fully parsed configuration is then stored in a `DotDict` class instance which extends the `dict` type to allow for convenient dot syntax access and a couple other usage options.

After processing the configuration files and command line arguments, the configuration is set and then treated as a constant global and passed to all the actions (note that it is not built for special actions). So the `get-menu` action uses the `menu.dir` option to create it and put its output there, but the `compile-menu` action uses the files there as input.

2.3 The IOManager Class

This is a major part of the software, taking care of all output, logging and running external commands. I mention it here to provide more detail on how the code is structured and to somewhat demonstrate its developer-friendliness. An instance of `IOManager` along with an object storing all the configuration options discussed above are the only two arguments passed to any action function. This then has to be able to do all the tasks an action can do[1] which is really just printing and running external commands.

The external commands are ran through a `run` method. This has to be passed a string which is the command to run and this will be run using the Python `subprocess.Popen` class. The method then takes care of handling output, printing it and logging it to a log file and also returning it in case it is needed. It also checks if the command return value and a there are `run_while` and `run_until` methods can be used to repeatedly call a function until a particular condition is satisfied. These have been designed to be rigorous and not fail unpredictably and have been tested with somewhat contentious commands like running `ssh -c *` style commands.

The class also provides some convenience features, like centrally handling printing, colors of the output and a verbosity flag. The instance has a verbosity attribute that it gets from the config, and every method that produces output to the terminal will have a verbosity parameter and the printing will only be done if that parameter is less than the verbosity attribute.

2.3.1 Logging

As mentioned before, it was a key requirement that the software is transparent about what commands are being ran and what are their results. In order to achieve that, the executed commands and their output is printed to the terminal. However, this can be overwhelming, and so when `IOManager` is instantiated it creates a new log directory for itself (according to the config). Where it stores a list of the actions passed to the software, the used configuration, a *main log* and all output from all commands that were ran.

Again, for more details see the documentation, but as key points, the main log contains a list of all executed commands (without their output) and some extra log messages prefixed with '#'. This makes it clear what exactly was done and it is close to a script that if ran directly would perform the same job, though there are some caveats to that. Finally, the command outputs are by default stored separately - each command has its own new file with the file name being chosen based on the command itself. This makes it relatively easy to find any errors that happened during the execution of the commands.

[1] With the exception of subscripts, small largely independent bits of Python code that can be ran using the `run-subscript` special action.

3 Summary

The new software and its features have been tested throughout the summer as they were implemented and there have been no issues, it is now ready for adoption. It takes 1–3 commands (depending on the user) to run all the tests and they can run in $\sim$ 15 minutes and the menu compilation and upload to P1 database can be done more easily. The main goal of making the process easy enough for other members of the L1CT team is to be tested, but we have reason to believe it will be so.

Besides that the unified configuration and the general layout of the code should prove a solid base for further changes and developments. All steps to make work reflecting future hardware upgrades as seamless as possible were taken. Aside from that that various smaller improvements were made to the testing process, using ssh jumps for passing authentication, noticeably speeding up the python parts of the code etc.

References

[1]G. Aad et al., *Technical Design Report for the Phase-I Upgrade of the ATLAS TDAQ System* (2013).

[2]F. Pastore, "The atlas trigger system: past, present and future", Nuclear and Particle Physics Proceedings **273-275**, 1065–1071 (2016).

[3]A. Koulouris et al., "Commissioning of the new muon-to-central-trigger-processor interface at atlas", Journal of Instrumentation **18**, C03020 (2023).

[4]*TriggerMenuTest*, `https://gitlab.cern.ch/atlas-l1ct/TriggerMenuTest` (visited on 09/21/2023), github repository.