National Institute of Nuclear Physics and Engineering
Horia Hulubei

NuGecko manual

A program initially written by:
Bastian Lher

Modified and adapted by:
Lucian Stan

Contents

Chapter 1

Introduction

Chapter 2

The main interface

Chapter 3

The Plugins

3.1 Packing: Write to file

This plugin can be found in the processing category.

It is the most important plugin in the current acquisition system. It takes the data from all the modules, correlates it and then writes it to file in the GASPware format.

The plugin requires a configuration file to be specified. This configuration file tells the plugin which channels correspond to which detector. It can be loaded by clicking the "Choose the configuration file" button. A pop-up window than appears, which allows the user to navigate to the location of the file.

The configuration file starts, on the first row, with the number of parameters for each type of detector. So, finding "3 2 5" on the first row means there are 3 parameters for the first type of detectors, 2 for the second and 5 for the third. There must be as many entries as the number of detector types.

Afterward, each detector is specified on its own row. The first value is the detector number inside its type. So, if there are 2 detectors of one type and 4 detectors of the second, they will be numbered as "1 2 1 2 3 4". Counting starts from 1.

The following values represent the channels which belong to this detector. The number must correspond to the index of the input of the plugin that channel is connected to. If the configuration file specifies "1 4 6 8", the plugin will pack the data from the 1st, 4th, 6th and 8th channels as a single detector with 4 parameters. The number of values here MUST be equal to the number of parameters of the type specified on the first row of the configuration file. Counting starts from 0.

The last value represents the detector type. It must be smaller or equal to the number of values on the first row of the configuration file. Counting starts from 1.

Do note that the configuration file MUST be correct, as no safeguards have been put in place in case of it being wrongly written. An example of a configuration file comes with the Gecko files and is called "det".

The write folder can be chosen by pressing the "Choose the write folder" button. A pop-up window than appears, which allows the user to navigate to the location where he wants the data to be written. He can also create new folders in the pop-up window.

The write folder is the location where the data files, the logbook and the raw data files (if applicable) will be placed. It can be changed at any time, but an already started run will remain in the folder it started to be written.

The user can choose the name of the run file in an editable field in the plugin interface. This is how the files will be named on the disk.

The plugin automatically constructs the run number, starting from "001". If a Run with that exact name already exists, it will automatically jump to the next available number. If there already exist a number of Runs, with some numbers missing, it will fill in those numbers.

The Run is automatically changed on certain conditions. The first condition is the user starting the run manually, either after opening the program or after a stop. The second condition is a time condition. If a certain number of hours have passed since the run started, it will be changed. The number of hours is set by an editable field. The values can be set between 1 and 10 hours. The third condition is related to the amount of data written in that Run. If it exceeds a certain value, the run will be changed. The value can be set through an editable field. It can be set between 200 and 4000 Mbytes.

The plugin calculates how much data was written to the disk in total, how much was written in the current file, how much time passed since the current run started and how much space is available on the disk.

The main function of the plugin is to collect the data from the inputs, correlate it and write it to a file.

The plugin takes the data from the inputs, with as many as 256 inputs possible. The plugin is made especially for Block Transfer Reads. This means that it does not take a single signal from each input, but a large number of signals from each input, as they were read in the Block Transfer. Each input signal consists of the input value and the timestamp.

After it read the data, the plugin starts parsing through it in order to recreate the events from the signals from the various inputs. It first finds the minimum timestamp from the signals that have not yet been processed. Then it searches for other timestamps that are within a certain distance of the minimum timestamp.

This coincidence interval is given by an editable field in the interface. The value is in multiples of the VME clock period, which is of 62.5 ns. The coincidence time should never be larger than the event busy time.

After an event is reconstructed, it is memorized in a cache, in order to be written to file.

The plugin writes in the GASPware format. All the write files are segmented into 16k blocks of data. The first block is the Run header. It contains information such as the number of detectors and parameters and number of detectors. It is otherwise empty.

The data blocks follow. Each one has a 16 word block header at the start. The events follow afterwards.

Each event begins with a header word. This is composed by the xF000 value plus the number of words in the event. A second value is added, currently linked to the pulsing. If pulsing is used and the beam is on, it'll be a 0001. If the beam is off or the pulsing is not used, it will be a 0000.

After that, the number of each type of detector that gave a signal follows. Each type of detector has a word, regardless if any of those detectors gave a signal or not. So, if the acquisition is made of HPGe and LaBr detectors and 1 HPGe and 1 LaBr fired, it will show "0001 0001". If two HPGe and no LaBr detectors fired, it will show "0002 0000".

After that, the values for each parameter of the detectors that sent a signal is written. For each detector that sent a signal, its number is added, followed by a word for each of its parameters. If one of the parameters didn't have a signal, a 0000 is added.

The detectors are written in their numbering order. Detectors of the first type are written first, followed by those of the second type, and so on.

If the raw write option is selected, the plugin will also write the raw input data to another file. It will be placed in a "Raw" folder inside the write folder. These values can then be used to reconstruct the events if something went wrong with the event reconstruction.

The plugin also events the digital logbooks, where any relevant information is written.

The experiment name, written in the Experiment editable field, is written to the logbook every time it is changed.

Any stop or start commands given by the user are also written to the logbook, along with the date and time of said command.

The opening and closing of a file is also written to the logbook, with the relevant time and date.

If pulsing is started, the start and stop of the beam is also recorded, with the relevant time and date.

The plugin also calculates the trigger rate, albeit not precisely. It counts the number of events it has processed. At certain intervals, it sends this number to the Run Manager, and also calculates an average trigger rate, which is written to the logbook every hour.

Finally, the user can add notes to the logbook by pressing the "Add a note to the logbook" button.

A pop-up appears, in which the user can put down what should be recorded.

The files which define this plugin are eventbuilderBIGplugin.cpp and eventbuilderBIGplugin.h and can be found in /plugin/pack.

3.2 Processing: MADC32Processor

This plugin can be found in the Processing category.

The main function of this plugin is to take the raw data from a single MADC32 module and process it, so that it may be of further use in other plugins in the program.

For each data block that comes through the input, the plugin starts parsing it event by event. An event consists of a header, data from all the channels that received a signal during that trigger, and an end word.

For each event, the plugin extracts the channels, their energies and the event timestamp. It then sends the energy and timestamp to the output corresponding to the channel that received the signal.

These functions used to be a part of the MADC32 Demux. However, the Demux is ran as part of the Run Thread, and there were suspicions that the running of these functions would add up to the dead time of the acquisition. Thus, they were moved as independent plugins, running in the Plugin thread, independent of the acquisition cycle itself.

There is no setting that must be specified upon creation of the plugin.

The plugin has one input and 32 outputs. The input must be connected to a MADC 32 module, and the outputs represent the 32 inputs of the module, in the same order. So ”out 0” will have the data from the first channel of the MADC 32 module, which is also labeled 0 on the module.

Each MADC 32 module in the acquisition must have its own MADC 32 processor plugin defined.

The plugin does not have any settings that can be changed by the user. The only setting it has is the data length format, which the plugin takes directly from the module to which it is connected.

The files which define this plugin are madc32Processor.cpp and madc32Processor.h and can be found in /plugin/processing.

It is possible that some modifications will be necessary in order to make Gecko compatible with CBLT read modes.

The plugin is not prepared to handle extended timestamps. Modifications may be necessary if this function should ever be used

3.3 Processing: MTDC32Processor

This plugin can be found in the Processing category.

The main function of this plugin is to take the raw data from a single MTDC32 module and process it, so that it may be of further use in other plugins in the program.

For each data block that comes through the input, the plugin starts parsing it event by event. An event consists of a header, data from all the channels that received a signal during that trigger, and an end word.

For each event, the plugin extracts the channels, their energies and the event timestamp. It then sends the energy and timestamp to the output corresponding to the channel that received the signal.

These functions used to be a part of the MTDC32 Demux. However, the Demux is ran as part of the Run Thread, and there were suspicions that the running of these functions would add up to the dead time of the acquisition. Thus, they were moved as independent plugins, running in the Plugin thread, independent of the acquisition cycle itself.

There is no setting that must be specified upon creation of the plugin.

The plugin has one input and 32 outputs. The input must be connected to a MTDC 32 module, and the outputs represent the 32 inputs of the module, in the same order. So ”out 0” will have the data from the first channel of the MADC 32 module, which is also labeled 0 on the module.

Each MTDC 32 module in the acquisition must have its own MTDC 32 processor plugin defined.

The plugin does not have any settings that can be changed by the user. The only setting it has is the data length format, which the plugin takes directly from the module to which it is connected.

The files which define this plugin are mtdc32Processor.cpp and mtdc32Processor.h and can be found in /plugin/processing.

It is possible that some modifications will be necessary in order to make Gecko compatible with CBLT read modes.

The plugin is not prepared to handle extended timestamps. Modifications may be necessary if this function should ever be used

3.4 Aux: Fanout

This plugin can be found in the Aux category.

Its main function is to take in an input of a certain type, be it Int or double, and provide a certain number of outputs with the same data in the same format.

The Fanout plugin is mainly used to take the data from the Processor plugins and make it available on more channels, which can then be used for different purposes, like plotting or writing to the disk.

When created, the number of outputs to be created must be specified. This number cannot be changed after its creation. The plugin will have a single input and the specified number of outputs.

The type of data to handle is also specified upon creation, by choosing either the FanoutInt or the FanoutDouble plugin type.

The data that comes into the input is then sent to all the created outputs. The data that comes out of any of the outputs is identical to that which was sent to the input.

The plugin has no other functions or settings.

The files which define this plugin are fanoutplugin.cpp and fanoutplugin.h and can be found in /plugin/aux.

There are no planned additions, improvements or features for this plugin. However, it is planned to investigate whether it is possible to bypass the need for this plugin.

3.5 Aux: Int —>Double

This plugin can be found in the Aux category.

Its main function is to take a number of inputs with Int format data, and to provide the same number of outputs with the same data, but in the Double format.

The IntToDouble plugin was mainly used for the Cache plugins, which were requiring Double format inputs. However, since the Cache plugins have been converted to take Int format inputs, it no longer serves any meaningful purpose.

This plugin is a prime candidate for removal from the program, unless another use can be found for it.

When created, the number of channels to be created must be specified. This number cannot be changed after its creation. The plugin will have the specified number of inputs and outputs.

The inputs and outputs are connected on a one-to-one basis. This means that the Int format data inserted into, say "In 2", will be found in the Double format in the "Out 2" output.

The files which define this plugin are inttodoubleplugin.cpp and inttodoubleplugin.h and can be found in /plugin/aux.

The plugin has no other functions or settings.

There are no planned additions, improvements or features for this plugin.

Chapter 4

The Modules

Chapter 5

The Interface

Chapter 6

The core of the program

Figure 6.1: Ultraviolet wavelenght picture of the Sun taken by SOHO. Image courtesy of NASA.

Bibliography

[1] http://nssdc.gsfc.nasa.gov/planetary/factsheet/sunfact.html

[2] Broggini, C. (2003). "Nuclear Processes at Solar Energy". Physics in Collision, Proceedings of the XXIII International Conference. p. 21.

[3] http://solar-center.stanford.edu/vitalstats.html

[4] J. R. Kuhn, R. Bush, M. Emilio, I. F. Scholl, The Precise Solar Shape and Its Variability Science 28 September 2012: Vol. 337 no. 6102 pp. 1638-1640

[5] Garcia, R.; et al. (2007). "Tracking solar gravity modes: the dynamics of the solar core". Science 316 (5831): 15911593.

[6] Mitalas, R. and Sills, K. 1992, "On the photon diffusion time scale for the sun", The Astrophysical Journal, Vol. 401, p. 759-760.

[7] Aschwanden, M. J. (2004). Physics of the Solar Corona. An Introduction. Praxis Publishing.

[8] Yuming Wang; et al. (2011). "Statistical study of coronal mass ejection source locations: Understanding CMEs viewed in coronagraphs", Journal of Geophysical Research, Vol. 116

[9] Andrei Zhukov; (2011) "EIT wave observations and modeling in the STEREO era", Journal of Atmospheric and Solar-Terrestrial Physics, Vol. 73, p. 1096-1116

[10] http://aia.lmsal.com/, Lockheed Martin instrument website for AIA

[11] http://lasco-www.nrl.navy.mil/index.php?p=content/lasco_desc