ALPIDE Characterisation Software Framework - Manual

rev. 1, August 4, 2016

1 Introduction

The software framework described in this document offers a lean software environment, which facilitates the development of test routines for the ALPIDE chip and ALPIDE modules.

The guidelines for the architecture development were the following:

- Simplicity of application development: the framework offers a package of building blocks that allow the user to easily implement applications for tests. Each test is supposed to be implemented in its own small standalone application.
- Independence from the readout system: as far as the generic functions of the readout boards are concerned, the implementation is transparent to the application, i.e. the high level implementation does not depend on which readout board is used.
- As a consequence the same Alpide implementation and (within limits) the same test routines can be used with all readout cards / test systems.

These requirements were addressed by the following class structure:

- An abstract base class TReadoutBoard, which all readout boards are derived from. Common functionality is declared in TReadoutBoard, special functionality of the different cards in the derived classes.
- A class TAlpide containing the basic chip interface and the register map.
- Two helper classes TAlpideDecoder and TAlpideConfig for event decoding and standard configuration commands, resp.
- A class TConfig containing configuration information.

This document describes how the framework can be used to implement test applications. Even if the framework is compatible with the test of both single chips and modules, this version of the document focuses on the test of single chips with the DAQ board. The first part of the document gives a step-to-step guide to the application development, the last part serves as a reference for the implemented methods.

2 User Manual

2.1 Initialisation of the Setup

The initialisation of the setup is the only part of each application, which necessarily depends on the type of readout board used. In general the initialisation consists of three steps (four in case of the DAQ board):

1. Generate a config object, which contains all information on the type of the setup and the settings of chip(s) and readout board(s).
2. Create a readout board object.
3. Create the chip object(s) and the connections between chip and board. The latter consists in passing the chip object a pointer to the readout board and "telling" the readout board object, which chip (identified via the chip ID) is connected to which control interface and which data receiver (non-trivial only for composed structures modules, HICs, staves).
4. For the DAQ board: power on the chip.

A standard initialisation for a setup with one DAQ board and a single chip with ID 16 is given in the following example (detailed explanations of the used methods in the reference part):

```
// vectors containing all chips and boards
// (in this example only one)
std::vector <TReadoutBoard *> fBoards;
std::vector <TAlpide *>       fChips;
TConfig                      *fConfig;

// create default config for one chip with ID 16
fConfig = new TConfig (16);

// USB helper-methods that create a DAQ board and push it into fBoards
InitLibUsb    ();
FindDAQBoards (fConfig, fBoards);

if (boards.size() != 1) {
  std::cout << "Problem in finding DAQ board" << std::endl;
  // throw exception
}

// create the chip object and the connections board <-> chip
fChips. push_back(new TAlpide (config->GetChipConfig(16));
fChips. at(0) -> SetReadoutBoard (boards.at(0));
fBoards.at(0) -> AddChip         (16, 0, 0);
```

```
// DAQ-board specific instructions
TReadoutBoardDAQ *myDAQBoard = dynamic_cast<TReadoutBoardDAQ*> (boards.at(0));
int               overflow;

if (myDAQBoard) {     // otherwise board was not of type TReadoutBoardDAQ
  if (! myDAQBoard->PowerOn(overflow) ) {
    std::cout << "LDOs did not switch on" << std::endl;
    // throw exception
  }
}
```

2.2 Interaction with the Readout Board

The basic configuration of the readout board is done automatically according to the information in the config, when the board object is constructed (configuration for readout see below). For more specialised configurations of the board each readout board can be configured directly using the generic `ReadRegister` and `WriteRegister` method or the methods of the derived readout board classes (see list in reference section). To access the methods of the derived classes the readout board has to be cast on the appropriate class. Checks of the object type can be done using a dynamic cast, e.g.

```
TReadoutBoardDAQ *myDAQBoard = dynamic_cast<TReadoutBoardDAQ*>(myReadoutBoard);
if (myDAQBoard) ...
```

or alternatively with the method `GetBoardType()`.

2.3 Chip Configuration

Also the configuration of the chip can be done directly using the methods `WriteChipRegister`, `SendOpCode` and `ReadChipRegister`. The register addresses can be given either directly as integers or using the names defined in `TAlpide.h`, e.g. `Alpide::REG_VCASN`. Additionally several methods for standard configuration procedures (masking, CMU configuration...) have been defined in AlpideConfig. All methods in AlpideConfig are used with the chip object as parameter, e.g.

```
// unmask all pixels
AlpideConfig::WritePixRegAll (myChip, Alpide::PIXREG_MASK, false);
// select row 5 for injection
AlpideConfig::WritePixRegRow (myChip, Alpide::PIXREG_SELECT, true, 5);
```

A command sequencer that can be used to send a predefined series of commands to the chip is also foreseen.

2.4 Data Taking

In this framework data taking is separated from the data decoding, giving the possibility to record either the raw data or hit information. Two configuration methods exist to define the sequence of pulse and strobe that is sent to the chip as well as the trigger source. After configuration one method (`TReadoutBoard::Trigger (int nTriggers)`) can be used to send triggers to the chip or module, a second one (`TReadoutBoard::ReadEventData()`) to retrieve the data event by event from the readout board objects. The two methods can be called one after the other (for low number of triggers, e.g. 50 triggers in a threshold scan) or can be put in two parallel threads, which is recommended for larger number of triggers to avoid filling the readout board buffer.

Event decoding is performed by two (static) helper classes `AlpideDecoder` and `BoardDecoder`, which decode the chip event or the readout board header and trailer, resp. To do so the buffer returned by `ReadEventData()` is first passed to `BoardDecoder::DecodeEvent()`, then to `AlpideDecoder::DecodeEvent()`.

Pending: For integration of the MOSAIC probably two methods will be added that need to be called before and after data taking (`Start/StopDataTaking()`).

3 Reference Manual

3.1 Readout Board

All readout board types used for the readout of ALPIDE chips or modules are derived from the class TReadoutBoard. The following sections describe the common functionality of all readout boards. (For the special functionality of the different board types for the time being refer to the header files `TReadoutBoardDAQ.h` and `TReadoutBoardMOSAIC.h`)

3.1.1 Board Communication

- `int ReadRegister(uint16_t Address, uint32_t &Value)`:
  read value of readout board register at address `Address`.
- `int WriteRegister(uint16_t Address, uint32_t Value)`:
  write value `Value` into readout board register at address `Address`.

3.1.2 Chip Control Communication

The methods for the chip control communication are supposed to be called only through the chip object and are therefore protected. The only exception is the method to read the chip registers, which can, if needed, also be called directly through the readout board (e.g. in a module where it is not clear, which chip addresses exist / function).

- `int ReadChipRegister(uint16_t Address, uint16_t &Value, uint8_t ChipID = 0)`:
  read value of chip register at address `Address`.

3.1.3 Triggering, Pulsing and Readout

Triggering and pulsing consists in sending a combination of first pulse, then strobe to the chip. Either one can be omitted (i.e. sending only pulse or only trigger) and the delays between them are programmable. Also the source for the trigger can be selected:

- `void SetTriggerConfig (bool enablePulse, bool enableTrigger, int triggerDelay, int pulseDelay)`: Chose which signal to send upon a trigger (software or external) as well as the delays from pulse to strobe (`triggerDelay`) and from strobe to the next pulse (`pulseDelay`, only relevant for MOSAIC).
- `void SetTriggerSource (TTriggerSource triggerSource)`: chose between internal or external trigger, trigger-source type defined in `TReadoutBoard.h`.

For data taking itself the readout board provides the following two methods:

- `int Trigger (int NTriggers)`: Sends `NTriggers` triggers to the chip(s). Here trigger means the strobe opcodes, pulse opcodes or a combination of pulse then strobe with the programmed distance inbetween. (The latter kept for redundancy, aim would be to send only pulses and generate the strobe on-chip.)

- `int ReadEventData(int &NBytes, char *Buffer)`: Returns the event data received by the readout card. Event data is in raw (undecoded) format, including readout card headers and trailers. The method returns one event at a time. Decoding is done in two separate decoder classes.

3.1.4 General

Construction: TReadoutBoard is the abstract base class for all readout boards. A readout board is therefore typically constructed the following way:

```
TReadoutBoard *myReadoutBoard = new TReadoutBoardDAQ   (config);
TReadoutBoard *mySecondBoard  = new TReadoutBoardMosaic(config);
```

Setup: In order for the control communication and the data readout to work in all cases (single chips, IB staves and OB modules) the readout card needs to have information on which chip, defined by its ID, is connected to which control port and to which data receiver. This information has to be added once from a configuration and then stored internally, such that the chip ID is the only parameter for all methods interacting with the chip for control or readout. The necessary information is added by the method

```
int AddChip (uint8_t ChipID, int ControlInterface, int Receiver)
```

3.2 Alpide Chip

This class implements the interface of the alpide chip (control interface and data readout) as well as the list of accessible register addresses and will be used for all test setups. All further information on the internal functionality of the chip are contained in the helper classes AlpideConfig (for configuration information) and AlpideDecoder (for decoding of event data). The class TAlpide contains the following functions:

3.2.1 Constructing etc.

- `TAlpide (TChipConfig *config)`:
  Constructs TAlpide chip according to chip configuration.

- `TAlpide (TConfig *config, TReadoutBoard *myROB)`:
  Constructor including pointer to readout board

- `void SetReadoutBoard(TReadoutBoard *myROB)`:
  Setter function for readout board

- `TReadoutBoard *GetReadoutBoard ()`):
  (Private) Getter function for readout board

3.2.2 Low Level Functions

- `int ReadRegister(Alpide::TRegister Address, uint16_t &Value`:
  read value of chip register at address `Address`. (A second version with an integer address exists).
- `int WriteRegister(Alpide::TRegister Address, uint16_t Value`:
  write value `Value` into chip register at address (A second version with an integer address exists). `Address`.
- `int ModifyRegisterBits(Alpide::TRegister Address, int lowBit, int nBits, int Value)`: write bits [`lowBit, lowBit + nBits - 1`] of register `Address`.
- `int SendOpCode(uint8_t OpCode)`:
  Send an Opcode (NB: in case of modules this sends an opcode to all chips connected to the same command interface as the current chip.)

3.2.3 Register Definitions

Chip registers are published in an enum type `TAlpideRegister`

3.2.4 High Level Functions

Higher level functionality of the alpide chip is implemented in two helper classes: AlpideDecoder for the event decoding and AlpideConfig for all configuration commands that go beyond mere communication with the chip and act upon the internal functionality of the chip.

3.3 Event Decoding

Event decoding is done in two static helper-classes (i.e. no object needs to be instantiated). Both provide a method `DecodeEvent`:

- `bool AlpideDecoder::DecodeEvent(unsigned char *data, int nBytes, std::vector <TPixHit> *hits)`: Tries to decode the event contained in `data` and stores the found hits in `hits`. Returns `true` in case of successful decoding, `false` otherwise. The pixel hit type is defined in `AlpideDecoder.h`.
- `bool BoardDecoder::DecodeEvent(TBoardType boardType, unsigned char *data, int &nBytes, TBoardHeader &boardInfo)`: Tries to decode board header and trailer of the event contained in `data`. The complete information of header and trailer (flags, time stamps...) is stored in `boardInfo`; `data` and `nBytes` are reduced to the chip event only. The header type is defined in `BoardDecoder.h`, the board type in `TReadoutBoard.h`.

3.4 Config class

The class TConfig contains all configuration information on the setup (module, single chip, stave, type of readout board, number of chips) as well as for the chips and the readout boards. It is based on / similar to the TConfig class used by the software for the Cagliari readout board and MATE. In addition to those versions it will allow modification / creation on the fly by the software (Use case, e.g.: create a config object after an automated check, which chips of the module work; modify settings according to parameters entered in the GUI by the user).

The TConfig object, which is needed for the initialisation of the setup can be constructed in three different ways:

- `TConfig(const char *fName)`: reads the config from the given file.
- `TConfig(int nBoards, std::vector <int> chipIds)`: creates a config for a given number of (DAQ) boards, and chips with the given chip IDs. In this case standard settings are used as defined in the header files `TBoardConfig.h` and `TChipConfig.h`.
- `TConfig(int chipId)`: creates a standard config for one DAQ board and one chip with the given chip Id. This is currently the only implemented constructor and can be used for initial testing of Alpide single chips.