ALPIDE software - Architecture description

rev. 1, July 20, 2016

1 Introduction

The software described in the following is intended to offer a lean software environment, which facilitates the development of test routines for the ALPIDE chip and ALPIDE modules.

The basic guidelines are the following:

- Simplicity of application development: the software should offer a package of building blocks that allow the user to easily implement applications for tests.
- Independence from the readout system: as far as the generic functions of the readout boards are concerned, the implementation should be transparent to the application, i.e. the high level implementation should 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 are 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.

2 Readout Board

Each board used for the readout of ALPIDE chips or modules has to be derived from the class TReadoutBoard and implement the functions described in the following:

2.1 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`.

2.2 Chip Control Communication

- `int ReadChipRegister(uint16_t Address, uint16_t &Value, uint8_t ChipID = 0)`:
  read value of chip register at address `Address`.
- `int WriteChipRegister(uint16_t Address, uint16_t Value, uint8_t ChipID = 0)`:
  write value `Value` into chip register at address `Address`.
- `int SendOpCode(uint8_t OpCode)`:
  Send an Opcode to all connected chips

These methods should be called only by a TAlpide object, not by the user directly. They are therefore to be declared `protected` and the classes TReadoutBoard and TAlpide friend classes (exception:`ReadChipRegister` may be declared public).

2.3 Triggering, Pulsing and Readout

Triggering and pulsing work differently in the MOSAIC readout board and in the Cagliari DAQ board. However, a common subset of the functionality can be found by dropping the "pulse-after-trigger" mode of the Cagliari DAQ board. In that case the parameters of pulse and trigger would be:

- Trigger enabled (yes / no)
- Pulse enabled (yes / no)
- Delay pulse - trigger
- Number of triggers
- Internal / external trigger
- (Delay trigger - following pulse: only for MOSAIC)

The ReadoutBoard class therefore has to implement the following configuration / setter methods:

- `void SetTriggerConfig (bool enablePulse, bool enableTrigger, int triggerDelay, int pulseDelay)`
- `void SetTriggerSource (TTriggerSource triggerSource)`

For data taking 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 redundance, 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. Decoding is done in a separate decoder class **To be decided:** Single events, vectors of events, blocks of data..

Note: This could be combined in one method. Keeping two leaves more flexibility, e.g. to have pulsing and reading in separate threads, however might require (in particular for the Cagliari DAQ board) to read the data already in the trigger function and buffer it in the software object until retrieved by ReadEventData.

2.4 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 TDAQBoard (config);
TReadoutBoard *mySecondBoard   = new TMosaic   (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)
```

Accessing readout board methods: Generic methods for chip interactions should not be accessed directly but only through the chip object (i.e. call `TAlpide::WriteRegister()` instead of `TReadoutBoard::WriteChipRegister()`).

Generic methods for readout board interaction can be accessed directly. Special methods for certain types of readout boards can be accessed e.g. after casting the readout board on the corresponding board type:

```
TReadoutBoard *myReadoutBoard = new TDAQBoard (config);

...
...

TDAQBoard *myDAQBoard = dynamic_cast<TDAQBoard*> myReadoutBoard;
if (myDAQBoard) {
  myDAQBoard->GetADCValue();
  ...
}
```

3 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 TAlpideConfig (for configuration information) and TAlpideDecoder (for decoding of event data). The class TAlpide contains the following functions:

3.1 Constructing etc.

- `TAlpide (TConfig *config, int ichip)`:
  Constructs TAlpide chip according to chip #`ichip` in the configuration.
- `TAlpide (TConfig *config, int ichip, 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 Low Level Functions

- `int ReadRegister(TAlpideRegister Address, uint16_t &Value`:
  read value of chip register at address `Address`.
- `int WriteRegister(TAlpideRegister Address, uint16_t Value`:
  write value `Value` into chip register at address `Address`.
- `int ModifyRegisterBits(TAlpideRegister Address, int lowBit, int nBits, int Value)`: write bits [`lowBit, lowBit + nBits - 1`] of register `Address`. This can be implemented either by reading from the chip, then writing a new value and / or caching the values in the software.
- `int SendOpCode(uint8_t OpCode)`:
  Send an Opcode (Q: define opcodes?)

3.3 Register Definitions

Chip registers are published in an enum type `TAlpideRegister`

3.4 High Level Functions

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

4 Overall structure

4.1 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 precise structure of the config class will be defined in the coming days. For the time being assume the following structure:

```
class TConfig {
 private:
  // List of general settings
  int  Setting1;
  int  Setting2;
  bool Setting3;
  ...

  // Board and Chip configs
  std::vector <TBoardConfig *> fBoardConfigs;
  std::vector <TChipConfig *>  fChipConfigs;

 public:
  // Constructor from file
  TConfig (const char *fName);
  // Constructor for on-the-fly construction
  TConfig (int numberOfBoards, int numberOfChips);

  // List of getter functions for settings
  int  GetSetting1 ();
  int  GetSetting2 ();
  bool GetSetting3 ();
  ...
  // List of setter functions for general settings
  void SetSetting1 ();
  void SetSetting2 ();
  ...

  // Getter functions for board and chip configs
  TChipConfig  *GetChipConfig  (int chipId);
```

```
  TBoardConfig *GetBoardConfig (int iBoard);

  // Write to file for future reference / bookkeeping
  void WriteToFile (const char *fName);
};
```

4.2 Application

An example for the skeleton of an application is given below:

```
main () {

  // initialise setup
  // 1) Create config object (here: from config file)
  // 2) Create readout board and chips
  // 3) Pass pointer to readout board to chip objects
  // 4) Pass information on ChipId / ControlInterface / Receiver to readout board

  TConfig       *myConfig       = new TConfig ("Config.cfg");
  TReadoutBoard *myReadoutBoard = new TMOSAIC (myConfig);

  std::vector<TAlpide*> Chips;


  for (int ichip = 0; ichip < myConfig->GetNChips(); ichip ++) {
    Chips          -> push_back       (new TAlpide (myConfig, ichip));        // create ichipth chip out of the config
    Chips [ichip]  -> SetReadoutBoard (myReadoutBoard);                       // set pointer to readout board in the chip object
    myReadoutBoard -> AddChip         (myConfig->GetChipId           (ichip),  // add ChipId / ControlInterface / Receiver settings
                                       myConfig->GetControlInterface (ichip),  // to readout board
                                       myConfig->GetReceiver         (ichip));
  }


  // configure chips
  // a) write registers directly
  for (std::vector<TAlpide*>::iterator ichip = Chips.begin(); ichip != Chips.end(); ichip ++) {
    ichip->WriteRegister (VCASN,   57);
    ichip->WriteRegister (ITHR,    51);
    ichip->WriteRegister (VPULSEH, 170);
    //...
  }

  // b) use predefined methods in TAlpideConfig class
  for (std::vector<TAlpide*>::iterator ichip = Chips.begin(); ichip != Chips.end(); ichip ++) {
    TAlpideConfig::SettingsForBackBias (ichip, 3);        // apply DAC settings for 3 V back bias
    //...
  }


  // do the scan

  int  NBytes;
  char Buffer[];

  myReadoutBoard->Trigger(myConfig->GetNTriggers(), OPCODE_PULSE);

  // Format of data still to be defined (single events, full buffer ...
  myReadoutBoard->ReadEventData (NBytes, Buffer);

  TAlpideDecoder::DecodeEvent (Buffer, std::vector<TPixHit> Hits);
  // ...


  // delete chips + readout board

}
```

5 Open Points:

- To be investigated: command sequencer implementation: this is needed for the probe station, is existing in the MOSAIC and planned for the DAQ board. Find best software implementation
- Common subclass for applications / scans?
- possibility to read command sequences from file?
- ...