PixLib Specifications – V 0.1

Introduction

PixLib is a software interface between the Pixel ROD and the software applications making use of the ROD to access Pixel modules. In this sense, PixLib is not intended to perform any specific task but to provide access to all the fundamental functions needed to control a Pixel module hiding to the end user the details of the ROD programming. PixLib will also provide an interface to the TIM and to the Pixel DCS system in order to give the end user the ability to control the trigger setup, the voltages and the temperatures. Last but not least, PixLib will provide access to a Configuration and Calibration Data Base where all the relevant module parameters will be stored.

Framework

PixLib will run in the ATLAS DAQ framework, so its primary target system is an Intel based Single Board Computer running Linux. It's however desirable to preserve the investment in the National Instruments VME interfaces. This can be done in two ways. The first is to use those interfaces under Linux using the existing NI driver. The cost of this operation is minimal; in particular all the existing DAQ and DCS infrastructure can be taken as-is. The second approach is to run PixLib under Windows. This approach gives the possibility to save existing LabWindows code at the application level, but requires some extra coding work. Moreover, being impossible to port the entire DAQ/DCS system to Windows, this implementation would not be full-functional.

Coding Issues

PixLib will be a set of C++ classes. The code must follow as close as possible the "ATLAS C++ Coding Standard Specification " (see this link).

Design considerations

Most of the operations we usually perform on a module are simple, in the sense that they can be easily described in terms of sequences of well known basic operations. So it must be simple for a detector expert to write an application performing some custom operation on a module.
We can distinguish 5 basic types of operations:

- Type 0: operations requiring no interactions with the module (like DB, DCS or memory operations)
- Type 1: module configuration commands generating no response
- Type 2: module configuration commands generating a response to be sent back to the application.
- Type 3: run commands generating histograms to be read out via VME (typically calibration runs).
- Type 4: run commands generating hits to be read out via S-link (typically data-taking runs).
- Type 5: run commands generating hits to be read out via VME (typically data-taking runs in a ROD-crate DAQ).

Operations of type 0 and 1 are performed synchronously.
Operations of type 2 can be performed asynchronously, even if, in most cases, the application simply waits for the response.
Operations of type 3, 4 and 5 (run modes) are performed asynchronously, but in mode 3 and 4 the application has nothing to do while waiting (if not collecting monitoring histograms or performing module reconfiguration in case of failure).
If a thread controls a single ROD, there is in general very little room for parallelism: there is not much the host can do while waiting for the ROD to complete the operation. The ROD itself will operate in parallel over different modules.
Different RODs are operated independently: the same set of commands can be dispatched to many ROD, but the individual executions are actually independent.

General Structure

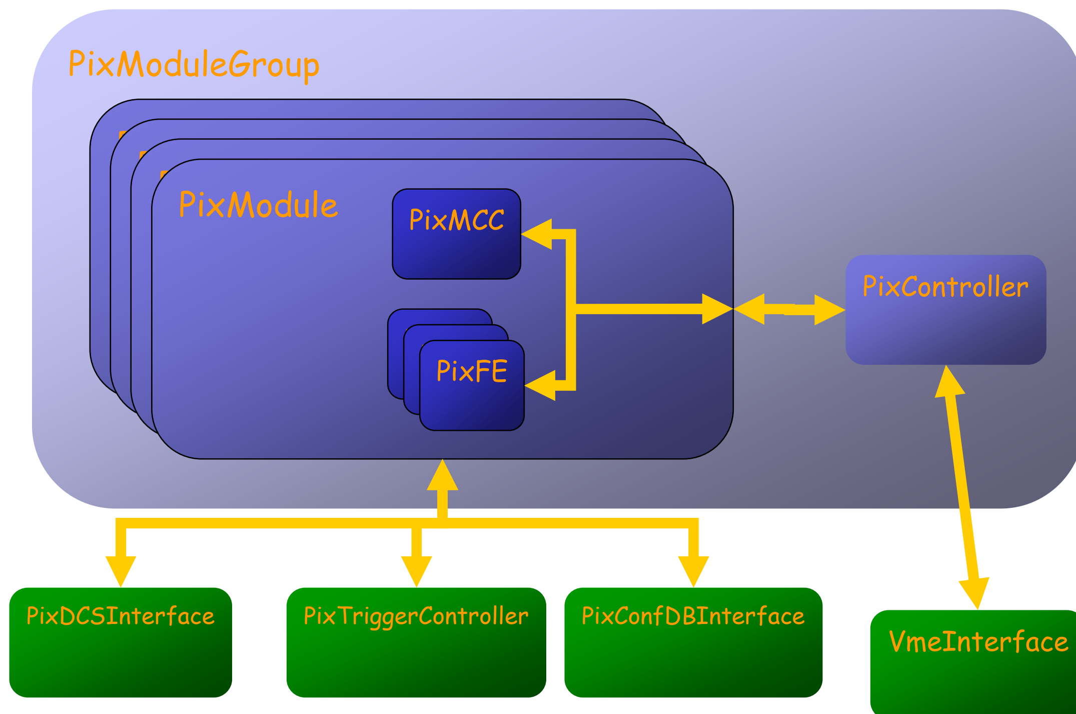

Figure 1 - General structure of PixLib

The class structure will closely follow the hardware structure (see Figure 1). The most relevant classes are:

- ***PixModuleGroup***: this is the top level class, and corresponds to a set of modules controlled by the same ROD. This means that the coordination of several RODs in a multi-ROD set-up is not part of PixLib and is left to the application layer (typically a run controller). This choice assume that the coordination needed is limited to dispatching command and checking for termination, and that there is no need for intelligent coordination between RODs. Should such a coordination

be necessary, it should be included in a PixLib class containing several PixModuleGroup objects. A PixModuleGroup object creates one or more instances of the following objects:
  - PixController (1)
  - PixModule (one per module connected to the controller)

  PixModuleGroup object receives pointers to:
  - VmeInterface
  - PixTriggerController
  - PixDCSInterface
  - PixConfDBInterface
- ***PixController***: it's an abstract class giving access to a specific ROD implementation. Typically this class will be a simple interface to RODModule; however it seems safer to keep an extra layer in the structure to give the possibility to interface PixModuleGroup with different controllers (e.g. a TPLL in the test-beam environment).
- ***PixTriggerController***: it's an interface to the TIMModule class. Since a single TIMModule will be used by several PixModuleGroup this class could be a good place to perform the necessary coordination. If needed, PixTriggerController could be used to interface PixLib with the BAT telescope trigger controller (TLU).
- ***PixDCSInterface***: it's an interfaces to the pixel DCS system, It must provide access to the voltage settings of the individual modules, and to the monitored parameters (voltages, currents, temperatures). It must provide notification of alarm conditions (via asynchronous signal handlers ?).
- ***PixConfDBInterface***: it's an interface to the configuration database, allowing to save and retrieve the configuration parameters. Note that this imply the definition of at least one module naming schema (probably at the beginning a production name will be used, then we will move to a geographic address).
- ***PixModule***: this is the class performing most of the work, since it contains the code to perform actions on a module. This class will create instances of the following objects:
  - PixMCC (1)
  - PixFE (16)

  PixMCC and PixFE will be responsible of the generation of the specific MCC and FE commands and of the handling of the configuration.
  PixModule will provide a set of methods performing complex task, like, for example, full module configuration, calibration loops, threshold scans, etc.).
  The actual implementation of these methods will depend on the capabilities of the PixController in use.
- ***PixMCC***: this class contains an image of all the MCC registers; this image is first loaded from the configuration DB and copied to the ROD memory to be used for configuration. Specific methods will allow to modify the values in memory and to update the ROD data structure. The class provide methods for generating and executing any MCC command. In case a specific ROD primitive exists, PixMCC will directly execute that primitive; otherwise PixMCC will generate the bit stream corresponding to the command and pass it to PixModule for execution.
- ***PixFE***: this class contains an image of all the FE DACs and pixel bits; this image is loaded from the configuration DB during the construction and copied to the ROD memory to be used for configuration. Specific methods will allow to modify the values in memory and to update the ROD data structure. The class provide methods for generating and executing any FE command. In case a specific ROD primitive exists, PixFE will directly execute that primitive; otherwise it

will generate the 5 MHz bit stream corresponding to the command and pass it to PixMCC which will encapsulate it in a RD/WR_FRONTEND command.

Elementary commands, like writing into a register, will be executed on a specific FE by calling the appropriate method of the corresponding PixFE class.
Higher level commands however will appear not only as PixFE methods, but also as PixModule and PixModuleGroup methods, so that you can execute them on many modules at the same time. In this case there will be an enable/disable mechanism to select only a subset of the full group.
The implementation details (i.e. if the operation will be a pure loop over low-level commands or a true parallel implementation) will not be visible by the end user.

Multi-thread safeness

PixLib will run in a multi-threaded environment, so, generically speaking, it has to be thread-safe. However, since the thread-safeness has a sizeable performance cost, some care must be taken to identify the specific sections being used concurrently by different threads. This means we have to put some constraints on the level of multi-threading the applications can implement.
It's difficult at this stage to be very precise on this subject. Here are few remarks to start the discussion:

- There is essentially no interplay between different RODs, so we have a class (PixModuleGroup) mapping an entire ROD; this class will provide the necessary isolation between different RODs provided that all the common classes (PixTriggerController, VmeInterface, PixDCSInterface, PixConfDBInterface) are fully thread safe.
- Most of the operations we need to perform at ROD level do not require parallelism, in the sense that the host has to ask the ROD to perform an operation and wait for the completion. The parallelism is provided by the ROD itself.
- The experience at the test beam shows that asynchronous readout driven by VME interrupts works very well, it's stable, reliable and easy to implement. So, it's natural to assume that the ROD will signal problems, errors and operation completions using VME interrupts.
- It's probably sufficient to run PixModuleGroup in two threads, one sending commands and receiving output in a synchronous way, the other receiving asynchronous outputs or polling the ROD. Most of the operations will be however synchronous

Use of the classes

The user of PixLib is expected to interact only with PixModuleGroup, PixModule, PixMCC and PixFE. PixDCSInterface, PixConfDBInterface and PixTriggerController are expected to be created by the application but not used directly (the same holds for VmeInterface, which is not part of PixLib). The other classes are internally referenced and should not be used directly. The list of the public methods of the classes is at the end of this document.
There are a couple of ancillary classes referenced in the list of methods. One is "bits", a serial bit stream handler; this class already exists in SimPix. The other is "histo" a general purpose histogram class; the idea is to have a lightweight class able to import/export histograms from/to (at least) IS and ROOT. We have to investigate if we can recycle something from DAQ-1 or ATHENA.

Revision History

V 0.1 – 15/2/03 Initial proposal to start the discussion. Public methods are outlined only for PixModuleGroup, PixModule, PixMCC and PixFE.

PixModuleGroup public methods

```
PixModuleGroup(string groupName,              // Constructor
               PixConfDBInterface *ifDB,
               PixDCSInterface *ifDCS,
               PixTriggerController *trigCtrl,
               VmeInterface *ifVME  );
~PixModuleGroup();                            // Destructor

// Accessors
string groupName();                         // Name of the ROD
int nModules();                             // Number of modules
PixModule *pixModule(int nMod);             // Pointers to modules
PixController *pixController();             // Pointer to controller

// Module selection
void enableModule(int modNum);              // Enable a module
void disableModule(int modNum);             // Disable a module
void enableAllModules();                    // Enable all modules
void disableAllModules();                   // Disable all modules
bool moduleEnabled(int modNum);             // Check if a module is enabled

// Configuration
void configure();                           // Configure enabled modules
void loadConfig(string configName);         // Load a configuration from DB
void saveConfig(string configName);         // Save current config to DB

// Trigger control
void internalTrigger(bool intTrig);         // Enable or disable internal
                                            // trigger
bool internalTrigger();                     // Test trigger mode
void triggerFrequency(int nClk);            // Set/Read internal trigger
frequency
bool triggerFrequency();
void triggerDelay(int nClk);                // Set/Read signal-command delay
int  triggerDelay();
void enableStrobe(bool enaStr);             // Enable or disable strobe command
bool enableStrobe();                        // Test strobe generation
void strobeDelay(int nClk);                 // Set/Read strobe-trigger delay
int  strobeDelay();
void consecutiveLVL1(int nLVL1);            // Set/Read number of consecutive
                                            // LVL1
int  consecutiveLVL1();

// Run control
void startHistoRun(int nEv, string histoName,  // Start a run with histogram
                   int histoBin2);             // output
void startVmeRun(int nEv);                     // Start a run with VME output
void startSlinkRun(int nEv);                   // Start a run with S-Link output
void stopRun();                                // Stop a run
int  nEvTaken();                               // Return the number of events
                                               // taken
```

```
// Histogram handling
histo getHisto(string histoName);             // Read an histogram

// Event handling
bool hitReady();                              // Test for pending hits
int  getHit();                                // Read an hit via VME

// Scans
void thresholdScan(int nEv, int nStep,        // Perform a threshold scan
                   int vcalMin, int vcalMax,
                   string histoName);
void tuneTDACScan(int nEv, string histoName); // Start a TDAC tuning procedure
void tuneFDACScan(int nEv, string histoName); // Start a FDAC tuning procedure
void monleakScan(int nEv);                    // Start a leakage current
measurement
```

PixModule public methods

```
PixModule(string moduleName,                   // Constructor
          PixModuleGroup *pixModGrp);
~PixModule();                                  // Destructor

// Accessors
string moduleName();                           // Name of the module
PixMCC *pixMCC();                              // Pointers to MCC
PixFE  *pixFE(int nFE);                        // Pointer to FEs

// Configuration
void configure();                                // Configure enabled modules
void loadConfig(string configName);              // Load a configuration from DB
void saveConfig(string configName);              // Save current config to DB

// Run control
void startHistoRun(int nEv, string histoName,  // Start a run with histogram
                   int histoBin2);              // output
void startVmeRun(int nEv);                      // Start a run with VME output
void startSlinkRun(int nEv);                    // Start a run with S-Link output
void stopRun();                                 // Stop a run
int  nEvTaken();                                // Return the number of events
                                                // taken

// Histogram handling
histo getHisto(string histoName);               // Read an histogram

// Event handling
bool hitReady();                                // Test for pending hits
int  getHit();                                  // Read an hit via VME

// Scans
void thresholdScan(int nEv, int nStep,          // Perform a threshold scan
                   int vcalMin, int vcalMax,
                   string histoName);
void tuneTDACScan(int nEv, string histoName);   // Start a TDAC tuning procedure
void tuneFDACScan(int nEv, string histoName);   // Start a FDAC tuning procedure
void monleakScan(int nEv);                      // Start a leakage current
                                                // measurement
```

PixMCC public methods

```
PixMCC(PixConfDBInterface *ifDB,               // Constructor
       PixController *pixCtrl  );
~PixMCC();                                     // Destructor

void writeRegister(string regName, int value);     // Write into a register
int  readRegister(string regName);                 // Read a register
void writeFifo(int value);                         // Write into a FIFO
int  readFifo(int fifoNum);                        // Read from a FIFO
void globalResetMCC();                             // Issue a full MCC reset
void globalResetFE(int nCK);                       // Issue a FE reset
void ecr();                                        // Send a ECR command
void bcr();                                        // Send a BCR command
void syncFE();                                     // Send a SyncFE command
void enableDataTaking();                           // Start MCC event builder
void writeFE(const bits &cmd, int dataLen);             // Send FE config
void readFE(const bits &cmd, int dataLen, bits &out);  // Send FE config

void configure();                                  // Configure the MCC
void loadConfig(string configName);                // Read the config from DB
void saveConfig(string configName);                // Save the config to DB
void enableFE(int feNum);                          // Enable a FE
void disableFE(int feNum);                         // Disable a FE
void enableAllFE(int feNum);                       // Enable all FEs
void disableALlFE(int feNum);                      // Disable all FEs
bool feEnabled(int feNum);                         // Check if a FE is enabled
void setOutSpeed(MccOutSpeed speed);               // Set MCC output speed
void consecutiveLVL1(int nLVL1);                   // Set the consecutive LVL1

void testEventBuilder();                           // Test the event builder
void testFifo();                                   // Test the FIFOs
```

PixFE public methods

```
PixFE(PixConfDBInterface *ifDB,                // Constructor
      PixMCC *PixMCC,
      PixController *pixCtrl  );
~PixFE();                                      // Destructor

// Configuration
void writeGlobReg(string regName,            // Write a value in a ROD mem copy
                  int value);                // of a Global Register
int  readGlobReg(string regName);            // Read a value from ROD mem
void writePixelReg(string regName,           // Write a pattern in the ROD mem
                   pixPattern pixPatt);      // copy of a Pixel Register
void writePixelReg(string regName,           // Write into a ROD mem copy of a
                   vector<int> pixVal);      // Pixel Register
vector<int> readPixelReg(string regName);    // Read a ROD mem copy of a reg
void configure();                            // Configure the FE
void setGlobalReg();                         // set Global Registers from mem
void getGlobalReg();                         // read Global Registers to ROD mem
void getGlobalLatch();                       // read Global Latches to ROD mem
void setPixelReg(string regName);            // set a Pixel Register from ROD mem
void setPixelReg(string regName,             // Write a pattern in a Pixel Reg
                 pixPattern pixPatt);        // (ROD copy not modified)
void setPixelReg(string regName,             // Write into a Pixel Register
                 vector<int> pixVal);        // (ROD copy not modified)
void getPixelReg();                          // read the Pixel Register to ROD mem
void getPixelLatch(string regName);          // read a Pixel Latch to ROD mem
void stepPixelReg(string regName,            // step a mask pattern
                  int nStep);
void loadConfig(string configName);          // Read the config from DB
void saveConfig(string configName);          // Save the config to DB

// Test
testGlobalReg();                             // Test Global Registers
testPixelReg(string regName);                // Test a Pixel Register
measureLeakage();                            // Read the MonLeak ADC
```