Fermilab/BD/TEV
Beams-doc-1067-v14
November 18, 2004
Version 14.0

# Tevatron Beam Position Monitor Upgrade Front-End Software Design

Luciano Piccoli, Margaret Votava, Dehong Zhang, Dinker Charak
Fermilab, Computing Division, CEPA

## *Abstract*

This document contains the design for the BPM/BLM upgrade data acquisition software. The proposed design defines a general BPM framework that can be used on other similar BPM projects across the laboratory. A specialization of the framework provides the functionality necessary to meet the requirements of the Tevatron BPM upgrade project.

**1 INTRODUCTION ..... 4**

**2 PROPOSED TEVATRON BPM SOFTWARE ..... 5**

2.1 CONTROL ..... 5
2.2 BUFFERING ..... 5
2.3 DATA ACQUISITION ..... 6
2.4 ACNET COMMUNICATION ..... 6
2.5 BUFFER READOUT ..... 6
2.6 DEBUG AND DIAGNOSTICS ..... 6
2.7 SOFTWARE DIAGRAM ..... 6

**3 SOFTWARE DESIGN ..... 9**

3.1 USE CASES ..... 9
*3.1.1 Initialization ..... 11*
*3.1.2 Mode Change ..... 13*
*3.1.3 Buffer Readout ..... 15*
*3.1.4 Diagnostic ..... 16*
*3.1.5 Alarm ..... 17*
*3.1.6 Data Acquisition ..... 18*
*3.1.7 State Device Change ..... 19*
*3.1.8 Configuration Change ..... 20*
3.2 CLASS DIAGRAMS ..... 21
*3.2.1 Tasks ..... 21*
*3.2.2 Controls ..... 22*
*3.2.3 Events ..... 23*
*3.2.4 Event Listeners and Generators ..... 23*
*3.2.5 Data ..... 24*
*3.2.6 Alarms ..... 25*
3.3 ACTIVITY DIAGRAMS ..... 26
3.4 SEQUENCE DIAGRAMS ..... 29
*3.4.1 Initialization ..... 29*
*3.4.2 Mode Change ..... 31*
*3.4.3 Buffer Readout ..... 32*
*3.4.4 Alarms ..... 33*
*3.4.5 Events ..... 34*
*3.4.6 Data Acquisition ..... 35*
3.5 PACKAGES ..... 36
*3.5.1 Generic BPM classes (GBPM) ..... 36*
*3.5.2 Tevatron BPM classes (TBPM) ..... 37*
3.6 IMPLEMENTATION ..... 38
*3.6.1 Building The Generic Framework ..... 38*
*3.6.2 Building Tevatron BPM Software ..... 41*

**4 APPENDIX ..... 44**

4.1 THE RECYCLER SOFTWARE ..... 44

*4.1.1 Control* ..... *44*
*4.1.2 Buffering* ..... *44*
*4.1.3 Readout* ..... *45*
*4.1.4 ACNET Communication* ..... *45*
*4.1.5 Debug and Diagnostics* ..... *45*
*4.1.6 Alarms* ..... *45*
4.2 CLASS DIAGRAM ..... 46

**5 BIBLIOGRAPHY ..... 47**

## Table of Figures

Figure 1 - Proposed tasks, queues, command and data flow ..... 7
Figure 2 - Tasks for the TBPM system [2] ..... 8
Figure 3 – Tevatron BPM front-end software use cases ..... 9
Figure 4 - Class diagram for tasks in the system ..... 21
Figure 5 - Main control classes ..... 22
Figure 6 - Events in the system ..... 23
Figure 7 - Event listeners and generators ..... 24
Figure 8 - Reading and saving data ..... 25
Figure 9 - Alarm classes ..... 26
Figure 10 - *ControlTask* flow ..... 26
Figure 11 - *DataAcquisitionTasks* flow ..... 28
Figure 12 - *BufferReadoutTask* flow ..... 29
Figure 13 - Objects creation sequence ..... 30
Figure 14 - Tasks initialization ..... 31
Figure 15 - Changing modes ..... 31
Figure 16 - Fast abort buffer readout ..... 32
Figure 17 - Alternative fast time plot readout ..... 33
Figure 18 - Alarm generation ..... 34
Figure 19 - Clearing an alarm ..... 34
Figure 20 - Event generation ..... 35
Figure 21 - State device change ..... 35
Figure 22 - Fast abort trigger generation ..... 36
Figure 23 – Complete TBPM front-end software class diagram (Tevatron specific classes are in gray) ..... 46

# 1 Introduction

This document describes the design chosen for the front-end data acquisition software for the Tevatron BPM upgrade. The goal is to provide clear guidelines for implementing and delivering a system that fulfills the requirements as specified in document #860.

Besides the requirements, other factors have to be considered for the design in order to achieve high quality software. These are:

- Maintainability: the software should be easy to maintain and can be easily adapted to new requirements with only minor changes;
- Extensibility: software should be easily extensible. The addition of new modes of operation should be a simple task involving minimal changes that do not affect existing components;
- Flexibility: configuration of the software should be easy to modify, adapting it to new and unexpected situations.
- Portability: software can be reused on another machines (e.g. Main Injector)

With these principles in mind the expected output is:

- A generic software framework for Beam Position Monitor systems;
- A working Tevatron BPM system that is maintainable and extensible.

The next describes the proposed design for the Tevatron BPM front-end software.

# 2 Proposed Tevatron BPM Software

The proposed Tevatron front-end data acquisition software is based on the software developed for the Recycler (see section 4.1). Many of its components can be reused on the Tevatron systems, such as timing control modules and readout procedures.

Additionally, the Tevatron system would benefit from the use of the backdoor services, making it possible to control and read out data bypassing the ACNET/MOOC infrastructure.

## *2.1 Control*

Similar to the recycler software, the Tevatron BPM software will have a control task that is responsible for receiving ACNET and backdoor command for switching between modes of acquisition. The control task will have all data acquisition tasks started at initialization[1], so no additional time is needed to create tasks while the system is running. The control task needs to resume or suspend tasks according to the mode selected. VxWorks takes about five times longer to start a task than to suspend or restart it (in microseconds on the PPC603 processor).

Before letting the readout task run, the control task must configure the EchoTek boards and the timing hardware. On the recycler software, the configuration is done by the readout task when it is started.

The control task will receive commands through an input/command queue. MOOC and the backdoor send events to the queue. Status from the control task is passed back through a response mechanism. Certain types of trigger also generate events that are passed along to the control task via its command queue.

## *2.2 Buffering*

Each data acquisition task has a data source and an output data buffer. Its data source can be either hardware entities (EchoTek or BLM boards), or the output buffer of another task, while its output data buffer can be another task's data source. For controlling access and avoiding race conditions, semaphores must be used to protect all data buffers.

Buffers can be used as a data destination or a data source. On a trigger, a data acquisition task may request data from the hardware, or it may request data from an internal buffer. This will be handled transparently. In both cases, the destination of the read out data will be another buffer. The ability of having a buffer as a data source helps to implement slow read out buffers, which would get input data from fast read out buffers (e.g. Fast Abort Buffer vs. Slow Abort Buffer (a more detailed list of buffers is given at section 3.5.2.1)).

[1] The recycler software has only one data acquisition task running at a time. When modes are switched, the control task starts the task for that new mode.

## 2.3 Data Acquisition

The system will have several readout tasks. Each one will be responsible for filling at least one data buffer (Fast Abort, Slow Abort, BLM). Every task runs within a closed loop and remains waiting for a trigger or command, which is received through its input trigger queue.

A TCLK, BSYNC or State Device Change can generate the trigger. These triggers arrive at the crate controller via interrupts or software function calls, which in turn create a trigger entity and send it to the input queues of the readout tasks. The trigger awakes the readout task, which performs its job. It consists of reading the latest data from its data source and writing it to its data destination(s).

## 2.4 ACNET Communication

ACNET requests will be handled via queues. When MOOC receives an external ACNET request (for reading or setting) it invokes a callback. The callback, which is part of the Tevatron BPM software, will create and send a request to the control queue or buffer readout queue (see Figure 1) and then wait for a reply. After completing the request, the BPM tasks (control task or buffer readout task) send back a status and/or data to the callback.

## 2.5 Buffer Readout

The user requests for reading data buffers are received via MOOC/ACNET according to the above section. The system has a pool of buffer readout tasks that are waiting for requests coming into the buffer readout queue. The number of tasks can be configured, and one task means that all requests will be handled serially and more than one task means that requests can be processed in parallel if the buffers to be read are different. If data from the same buffer is requested, it will be handled serially because the buffers are protected by semaphores and only one task can access its contents at a time.

## 2.6 Debug and Diagnostics

The backdoor scheme may be used in the Tevatron BPM data acquisition software. The communication with the data acquisition software will follow the same method used by ACNET/MOOC calls. Whenever a request comes from the labview interface that is mapped to a callback that sends the request through a queue and gets the reply from the system. The DA software will look the same from the viewpoint of the backdoor system and the ACNET/MOOC system.

## 2.7 Software Diagram

The following picture (Figure 1) shows the proposed tasks, queues, data and command flow for a generic BPM system. The structure shown is valid for *one* crate within the

system. The blue circles represent the tasks; the green boxes are the input queues for the tasks; the yellow boxes are the data sources and data destinations; and the blue boxes are external entities (triggers and MOOC/Backdoor).

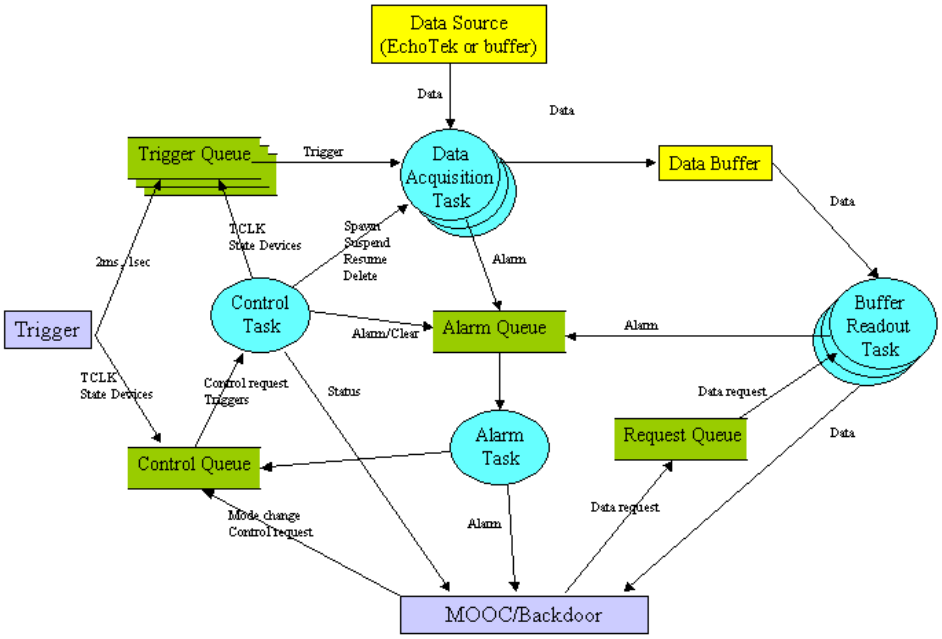


**Figure 1 - Proposed tasks, queues, command and data flow**

Object oriented design is used to realize the entities depicted in figure 1. The Unified Modeling Language is used to describe general use cases, classes and its relationships, control and data flows.

Figure 2 shows a specialized version for the Tevatron based on the generic BPM system (for a single crate). In the picture there are several data acquisition tasks (named *BPM Fast Abort Task*, *BPM Slow Abort Task*, *Turn by Turn Task*, etc), some buffers are defined (*BPM Fast Abort Buffer*, *BPM Slow Abort Buffer*, *Turn by Turn Buffer*, etc) and there is only one *Buffer Readout Task*[2].

[2] There is only one Buffer Readout Task shown, but the design allows it to have multiple tasks.

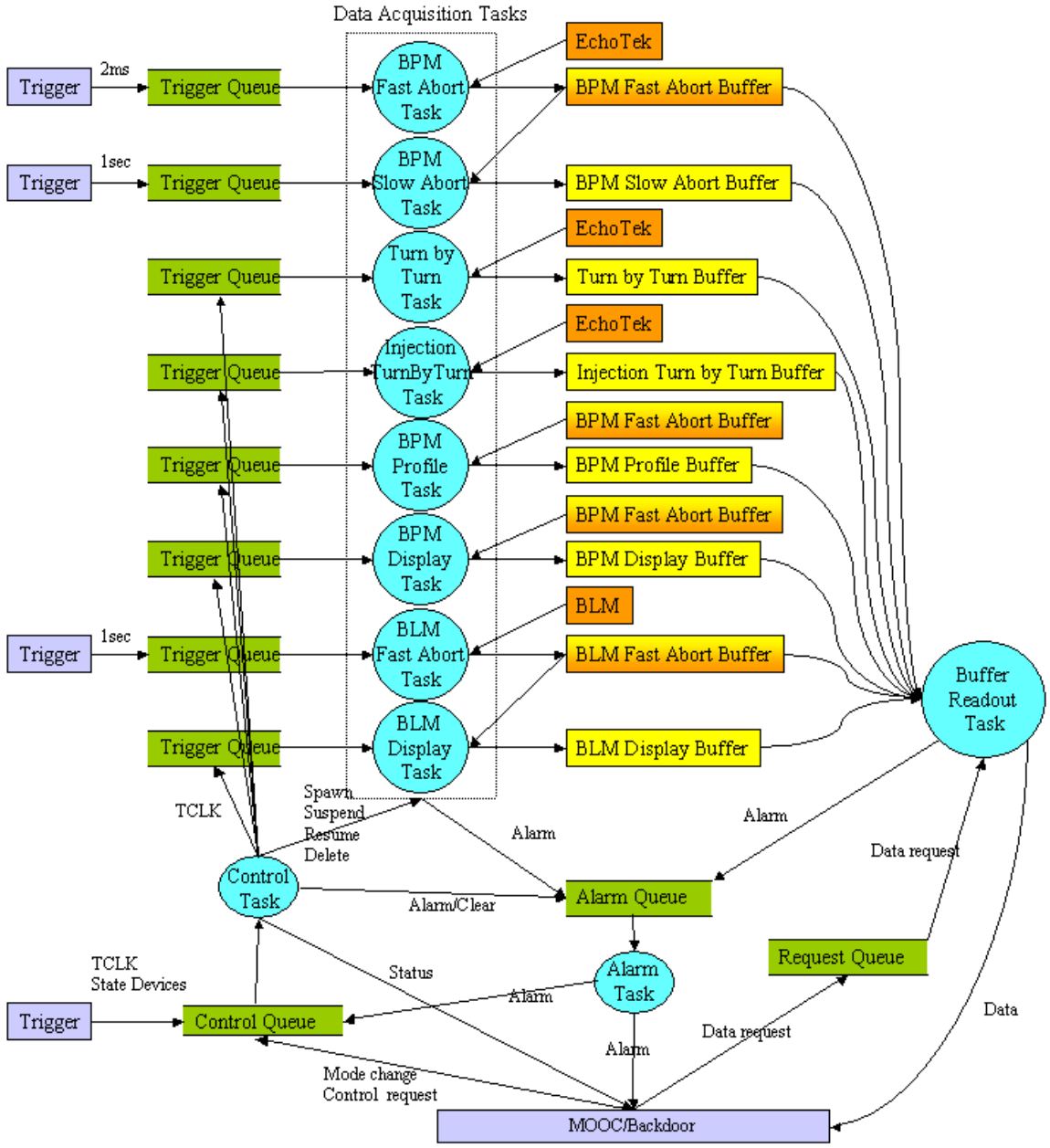


**Figure 2 - Tasks for the TBPM system** [2]

# 3 Software Design

The remaining sections of this document describe the design of the Tevatron BPM upgrade front-end software. It takes into consideration general software quality aspects as well as aims to provide an extensible framework for future similar projects within the laboratory.

The following sections describe the use cases identified for the project, static structures and dynamic diagrams. Use cases follow the format adopted by Alistair Cockburn [Cockburn] and the notation of static and dynamic diagrams follow the UML standard [Fowler].

## *3.1 Use Cases*

One crate in the TeV BPM DAQ system interacts with the extenal world through actions initiated by actors. The main actors interacting with the system are: *User* and *Trigger*. Actors being used by the system are: *EchoTek*, *BLM* and *TimingSystem*.

The User can be a control room operator, a beam physicist or another software. The User interacts with the system by *initializing* it; requesting *mode changes*; *reading out* its buffers; activating *diagnostics*. On any of these interactions there can be *alarms*, which is handled by a separate use case.

The other actor in the system, the Trigger, is any external event that is capable of changing the internal state of the system. A trigger activates the *data acquisition* from BPM and BLM boards; and input to *state device changes*. The user may request *configuration changes* of the system at any time.

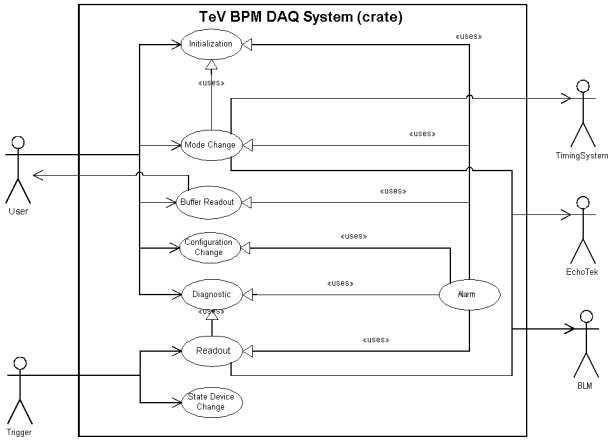


**Figure 3 – Tevatron BPM front-end software use cases**

Figure 3 shows the use cases identified for the Tevatron BPM front-end system. Each ellipse represents one use case. The use cases are described in more detail in the following sections.

### 3.1.1 Initialization

#### 3.1.1.1 Description

This use case allows the user to initialize one front-end DAQ system crate.

#### 3.1.1.2 Basic Flow of Events

1. User ask the system (one crate) to be initialized
2. Control task is created
    a. Control task creates control queue
3. Configuration for the crate is downloaded
4. Control task initializes EchoTek hardware
    a. EchoTek hardware is tested (optional)
5. Control task initializes BLM hardware
    a. BLM hardware is tested (optional)
6. Control task initializes timing system
    a. Timing hardware is tested (optional)
7. Control task creates data acquisition tasks
    a. Queues for receiving triggers are created by the tasks
8. Control task allocates internal buffers
9. Control task creates buffer read out tasks
    a. Buffer request queue is created
10. Control task creates alarm task
    a. Alarm queue is created by the alarm task
11. Alarm task announces itself to the tasks in the system
12. Trigger generators are created
13. Trigger listeners are registered
14. System is enabled
15. All tasks are started
16. System is ready for use (READY state)

#### 3.1.1.3 Alternative Flows

1. Control task fails to start (2) – other basic OS failures follow same steps
    a. Report error to user
    b. Generate alarm (if alarm task is running)

2. Could not download configuration (3)
    a. Use default configuration
    b. Limit usage of the system (e.g. don’t support turn-by-turn requests)
    c. Report error to user

3. EchoTek card(s) did not pass test (4.a)
    a. Generate internal alarm
    b. Set ALARM state
    c. Report error to user

4. BLM board(s) did not pass test (5.a)
    a. Generate internal alarm
    b. Set ALARM state
    c. Report error to user

5. Timing system did not pass test (6.a)
    a. Generate internal alarm
    b. Set ALARM state
    c. Report error to user

#### 3.1.1.4 Preconditions

Crate is turned off or rebooted.

#### 3.1.1.5 Postconditions

System is taking data in normal operation mode (READY state) or in a limited operational mode.

### 3.1.2 Mode Change

#### 3.1.2.1 Description

This use case allows the user to request a mode change of the front-end DAQ software. There are basically two modes of operation: closed orbit and turn-by-turn. The default mode is closed orbit, and the turn-by-turn mode is enabled at user requests or at a certain TCLK event. When changing modes, the system has to reload and reprogram the EchoTek boards and timing hardware according to the mode specification.

#### 3.1.2.2 Basic Flow of Events

1. User requests a mode change (e.g. from closed orbit to turn by turn)
2. MOOC call back creates an internal request for mode change
3. Request is posted to the control task queue
4. Request is retrieved by the control task
5. Control task checks the request
6. EchoTek boards are configured
7. Timing system is configured
8. Triggers are enabled/disabled (e.g. 2 ms closed orbit trigger)
9. Read out tasks are suspended/resumed
10. Mode has changed (CLOSED_ORBIT or TURN_BY_TURN state)

#### 3.1.2.3 Alternative Flows

1. An event triggers a mode change (1,2)

2. Mode cannot be changed (4)
    a. Generate internal alarm
    b. Return error to user

3. Requested mode change to the current mode (4)
    a. Restart mode (e.g. second turn-by-turn request); or
    b. Ignore request
    c. Return error.

4. Data acquisition task for current mode is in the middle of a readout (4)
    a. Data partially read must be thrown away
    b. Pointers and counters are not updated
    c. Data acquisition task has to go back to a safe place when it is restarted, i.e. it cannot go back to where it was when the mode was changed (unless there is no data loss or data read is consistent).

5. Failure to change mode (6 to 9)
    a. There are conditions preventing the system to change mode
    b. Return error

#### 3.1.2.4 Preconditions

System is in a known operational state.

#### 3.1.2.5 Postconditions

System has been reconfigured to run in a new mode and is acquiring or ready to acquire data.

### 3.1.3 Buffer Readout

#### 3.1.3.1 Description

This use case allows users to request data from the front-end software. Data is read out from the data acquisition boards and stored in internal buffers. Data from these internal buffers are requested in this use case, and portions of it or all its contents are returned.

#### 3.1.3.2 Basic Flow of Events

1. User requests data buffer from the system
2. An internal request is created
3. Request is posted to the buffer readout queue
4. Request is retrieved by one buffer readout task
5. The request is verified and the buffer is selected
6. Buffer is read and converted to online format (see document #860 for structures)
7. A reply with the resulting data structure returns from the buffer readout task
8. Data is sent back to the user

#### 3.1.3.3 Alternative Flows

1. Request is not valid (5)
    a. The data requested does not exist or is out of boundaries

2. No data in the buffer (6)
    a. Error is returned

#### 3.1.3.4 Preconditions

Internal data buffers have data.

#### 3.1.3.5 Postconditions

None.

### 3.1.4 Diagnostic

#### 3.1.4.1 Description

Use case used when user wants to get more information about the current system situation. Level of debug can be increased; buffers, queues and tasks are monitored more closely.

#### 3.1.4.2 Basic Flow of Events

1. User requests system to enable diagnostics through an online application
2. An internal request is created
    a. A request can be:
        i. Increase debug/diagnostic level
        ii. Return statistics information
        iii. Start test sequences (for EchoTek, timing board, calibration subsystem)
3. Request is posted to the control task queue
4. Request is retrieved by control task
5. Control task perform the diagnostic request

#### 3.1.4.3 Alternative Flows

1. System cannot enter diagnostic mode (5) (e.g. system is currently in turn-by-turn mode – high priority)
    a. Return error

#### 3.1.4.4 Preconditions

System has been initialized and may not be performing well.

#### 3.1.4.5 Postconditions

If item 2.a.i – system is running at a higher debug/diagnostics level. Performance of the system may be affected.

If item 2.a.iii – test are finished and system is back to normal operation.

### 3.1.5 Alarm

#### 3.1.5.1 Description

This is a use case used by other use cases in the system. It is triggered by alarm situations within the system. It is generated internally and there is no input from external actors. The alarm is handled by an alarm task, which may announce the alarm to the external world, depending on how critical is the situation. The system enters an alarm state that is cleared when the alarm conditions have been removed.

#### 3.1.5.2 Basic Flow of Events

1. An internal failure is detected
2. An alarm is created
3. Alarm is posted to the alarm queue
4. Alarm task retrieves alarm from queue
5. Task evaluates the priority of the alarm
6. Task generate an external alarm, if necessary
7. Control task is informed of the alarm state (if control task is not the generator of the alarm)
8. Control task decides the alarm is cleared
9. Alarm clear event is created
10. Alarm clear is posted to the alarm queue
11. Alarm task retrieves alarm clear from queue
12. Alarm task clear the alarm state

#### 3.1.5.3 Alternative Flows

1. User clears the alarm through the online software (8)

#### 3.1.5.4 Preconditions

A failure or a potential future failure is detected.

#### 3.1.5.5 Postconditions

System is set to an alarm state; the state can be cleared after the alarm condition is removed.

### 3.1.6 Data Acquisition

#### 3.1.6.1 Description

This use case describes the actual data acquisition part of the system. The external actors involved with this use case are the triggers. A trigger is any entity that starts the action of data acquisition. Following a trigger, the system has to perform the read out of a data source (hardware or internal buffers) and save the data to internal buffers.

#### 3.1.6.2 Basic Flow of Events

1. A trigger is generated and received by the system
2. A trigger event is created and posted to an event queue
3. The data acquisition task retrieves the trigger from the queue
4. Data acquisition task performs the data acquisition
5. Data is saved in an internal buffer
6. Data acquisition task is ready for next trigger

#### 3.1.6.3 Alternative Flows

1. Specific triggers may be posted to the control tasks' queue (2)
    a. Control task retrieves the trigger from the queue
    b. Control task redirects trigger to a data acquisition input trigger queue

2. Data source is not ready to send data (4)
    a. Data acquisition task has to wait for a defined amount of time
    b. If there is a time out an alarm is generated

#### 3.1.6.4 Preconditions

Data acquisition hardware and timing system are configured and ready to provide data.

#### 3.1.6.5 Postconditions

New data is saved in internal buffer and can latter be retrieved by the user

### 3.1.7 State Device Change

#### 3.1.7.1 Description

This use case illustrates the reaction of the system after a state device is changed. A state device can be considered an actor, more specifically a *trigger*, even though it does not trigger any data acquisition. The system has to monitor several state devices, which contain information about the accelerator status, beam type, etc. Those are important information that is part of the metadata sent back to the user (Buffer Readout use case).

#### 3.1.7.2 Basic Flow of Events

1. A state change is received by the system
2. A state change event is created
3. The event is posted to the control queue
4. The control task receives the event
5. Control task updates the metadata

#### 3.1.7.3 Alternative Flows

None

#### 3.1.7.4 Preconditions

None

#### 3.1.7.5 Postconditions

Metadata is updated with latest state device status.

### 3.1.8 Configuration Change

#### 3.1.8.1 Description

The configuration use case describes the actions taken by the user in order to change the behavior of the system. The user can specify new values for calibration, timing, filter settings, etc. During the initialization, the system receives a default configuration, and this use case represents system changes after the initialization phase.

#### 3.1.8.2 Basic Flow of Events

1. User request a configuration change (through some mechanism not defined yet)
2. A control request is created
3. Control task receives the request
4. Request is validated
5. Check if configuration can be changed
6. Change configuration

#### 3.1.8.3 Alternative Flows

1. Request is not valid (4)
    a. Generate error
    b. Do not change configuration
    c. Generate internal alarm

2. Configuration cannot be changed (e.g. system is in turn-by-turn mode) (5)
    a. Wait until configuration can be changed

#### 3.1.8.4 Preconditions

System is initialized.

#### 3.1.8.5 Postconditions

New configuration has been applied to the system.

## 3.2 Class Diagrams

This section describes the static structure of the system. The complete class diagram is available in the appendix. We broke down the main diagram into pieces that handle specific parts of the system. Every piece is described below, each one contains a part of the full class diagram. Classes painted in gray contain code specific for the Tevatron BPM system, remaining classes are part of the generic framework. Class names throughout the text are written in *italic*.

### 3.2.1 Tasks

The system has a certain number of independent processes; each one has a specific job. The tasks in the system are all subclasses of a VxWorks task wrapper (Class *Task*). The wrapper contains basic methods and attributes that represent a task. Figure 4 contains the task classes in the system. The upper class represents the wrapper.

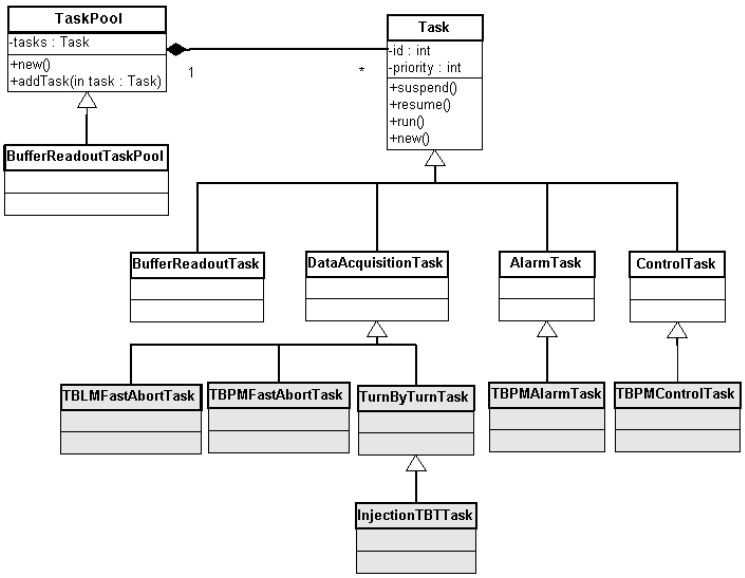


**Figure 4 - Class diagram for tasks in the system**

The system is overseen by a *ControlTask*, which is responsible for initializing most of the system, configure hardware (EchoTek boards, BLMs and timing module), switching acquisition modes, control other tasks in the system and keep track of the overall state.

The *DataAcquisitionTask* represents the tasks that are responsible for acquiring data and storing them in internal buffers. There can be several *DataAcquisitionTask* subclasses, each one has a different acquisition method, can read data from different sources and

store them in different destinations. Examples are *TBLMFastAbortTask*, *TBPMFastAbortTask* and *TurnByTurnTask*.

*BufferReadoutTask* provides a standard interface to retrieve data from internal buffers. That interface can be used by MOOC and backdoor for requesting any type of data (closed orbit, turn-by-turn, display frame, etc.). The system can provide parallel access to the internal data buffers. That feature is available through the *TaskPool* class. The *BufferReadoutTaskPool* provides any number of *BufferReadoutTask*, which can handle requests in parallel.

The *AlarmTask* handles any alarms generated in the system. It is its responsibility to check the system alarm queue and decide whether to put the system in an alarm state and send an alarm to the outside world.

### 3.2.2 Controls

The main class in the system is the *ControlTask*. It is however controlled by the *BPM* class. The class *BPM* makes a few assumptions about the system, and has common code for BPM systems in general. A more specialized class (*TBPM*) has specific implementation for the Tevatron BPM system. It contains objects of the classes *TimingSystem*, *EchoTekPool* (*EchoTek*) and *TBLM*, which are the hardware present in the system. Additional hardware classes may not be shown in the diagram on Figure 5.

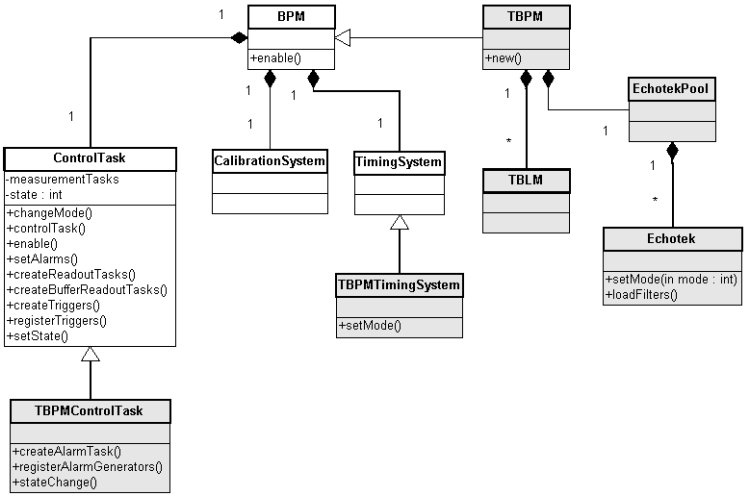


**Figure 5 - Main control classes**

The *BPM* class contains the entry point of the system. It is responsible for starting the *ControlTask*, which will in turn start the rest of the system.

### 3.2.3 Events

The system is composed of tasks and queues. The information flowing through the queues into the tasks are *events*. *Event* is the super class which has the most basic information about a generic event.

There can be several types of events. Those are described below as subclasses of *Event*. Types of event are:

- *Alarm*: event generated by a task signaling an alarm situation;
- *Request*: generic request;
  - *ReadoutRequest*: request to read an internal buffer;
  - *ControlRequest*: request of some control action (e.g. change mode)
- *Reply*: generic reply for a request;
  - *ReadoutReply*: reply for a buffer readout operation;
- *Trigger*: event generated on a trigger.

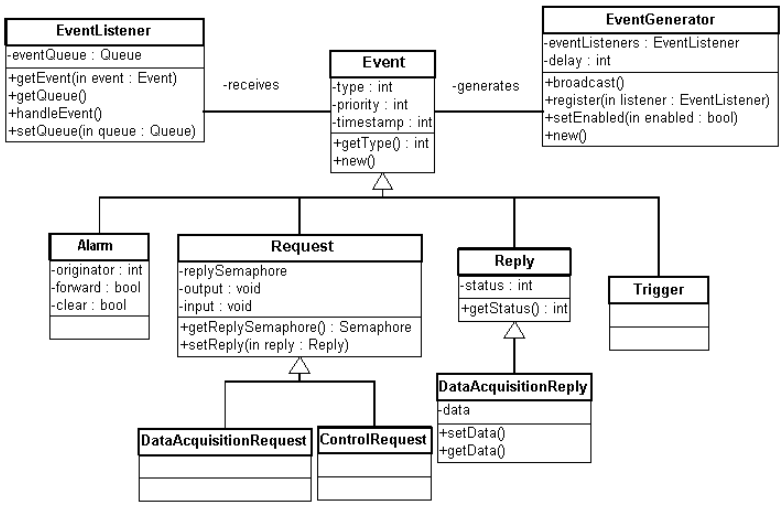


**Figure 6 - Events in the system**

An *Event* is generated by an *EventGenerator*. The *EventGenerator* has a list of *EventListeners*, to which an event is broadcasted after being generated. *EventListeners* can be dinamically added or removed from the list. The *EventListener* receives an event in its *eventQueue*. The *Event* is removed from the queue by *handleEvent ()*.

### 3.2.4 Event Listeners and Generators

*Events* can be generated and received by any entity in the system. Figure 7 shows the classes that currently deal with events. *EventListeners* are:

- *DataAcquisitionTask*: receive *Trigger* events signaling the data acquisition process;
- *BufferReadoutTask*: receive *ReadoutRequest* events when buffered data is requested.

*EventGenerators* are:

- *StateChangeEventGenerator*: generate a *Trigger* signaling a state device change;
- *InterruptEventGenerator*: generic event generator based on interrupts;
    - *TCLKGenerator*: generate TCLK *Triggers* on interrupts;
    - *TimeEventGenerator*: generate an *Event* on every tick of a timer.

*EventListeners* **and** *EventGenerators*:

- *AlarmTask*: receives *Alarms* from other tasks in the system; and generates *Events* sent to the *ControlTask* to inform about the current alarm situation;
- *ControlTask*: receives *ControlRequests* and *Triggers*; and generates *Alarms* and *Triggers*.

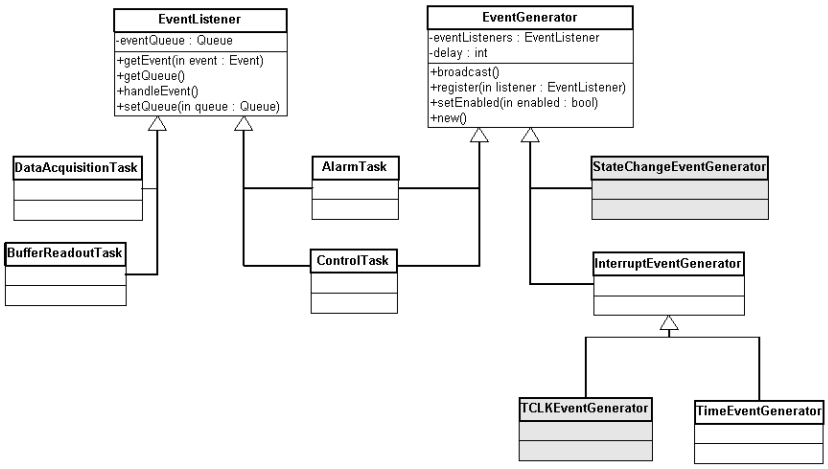


**Figure 7 - Event listeners and generators**

### 3.2.5 Data

During the data acquisition process the *DataAcquisitionTasks* perform reads from a *DataSource* (EchoTek or BLM boards) and save the result to an internal *DataBuffer*. A *DataSource* defines a generic class for reading out *DataEntries*. There can be several types of *DataSource*. For the Tevatron BPM system three of them are defined: *EchoTek*, *BLM* and *DataBuffer* (see Figure 8). This means that data can be retrieved either from the EchoTek boards, BLM boards or from an internal buffer (e.g. a task can feed the slow abort buffer with data from the fast abort buffer).

The destination of data read by the *DataAcquisitionTask* is a *DataBuffer*. It has knowledge of the *Metadata* used to tag the data, such as beam type, accelerator state and system status. All data entries are organized as *DataEntries*. The *DataEntry* can vary depending on the type of measurement.

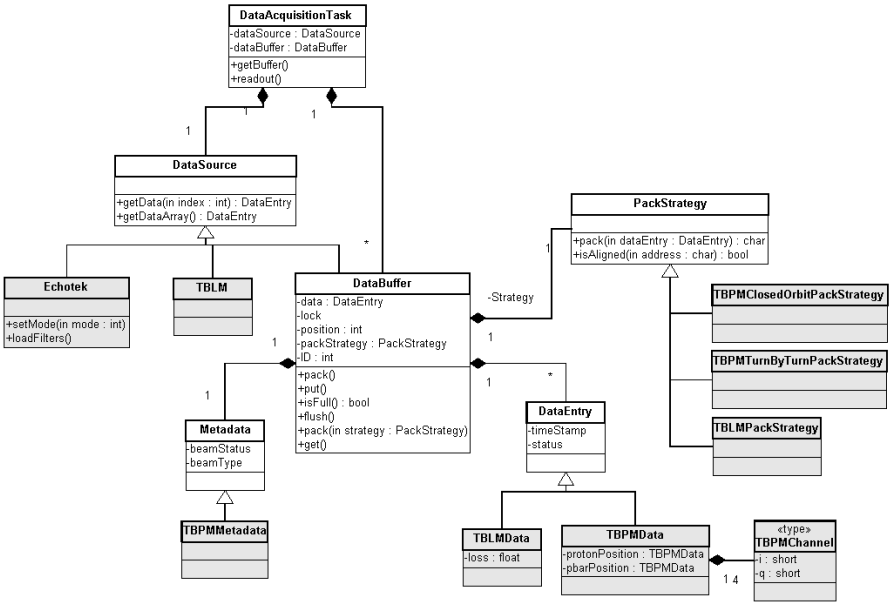


**Figure 8 - Reading and saving data**

The *DataBuffers* have data stored in a format that may be different from the format sent to the end user through the *BufferReadoutTasks*. Data is formated according to a *PackStrategy*. Depending on the data type and on the *ReadoutRequest* a specific *PackStrategy* is used (e.g. *TBPMClosedOrbitPackStrategy* and *TBPMTurnByTurnPackStrategy*).

### 3.2.6 Alarms

The classes responsible for handling and generating alarms are shown on Figure 9. An *Alarm* is generated by an *AlarmGenerator*. The generators in the system are the following tasks: *BufferReadoutTaks*, *DataAcquisitionTask* and *ControlTask*.

The *AlarmTask* is responsible for receiving *Alarms* generated by the *AlarmGenerators*. It declares an alarm state depending on the *Alarm* received.

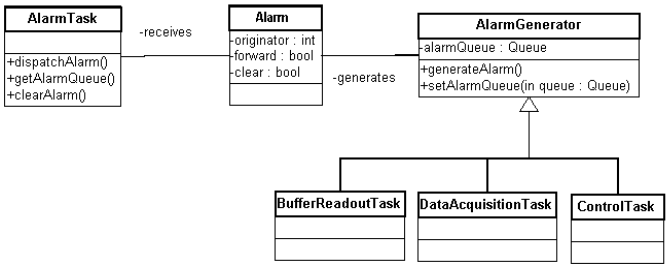


**Figure 9 - Alarm classes**

## 3.3 Activity Diagrams

This section contains diagrams showing the work flow of different tasks in the system. Figure 10 contains the basic flow for the *ControlTask*. It basically has to take care of the initialization of the system and enter a closed loop waiting for commands from its input queue. These commands are requests from MOOC, backdoor messages, alarms or triggers.

After receiving a request from its input queue, the *ControlTask* starts to process the it. This is represented by the ProcessRequest state, in which all types of input requests are handled. Requests can be data acquisition, calibration or diagnostics commands.

**Comment [LP1]:** What happens if the control task gets stuck and don't read the input queue? What if a very high priority request comes into the queue? The control task has to deal with it ASAP. Perhaps there should be a pool of control tasks for reading the input queue?

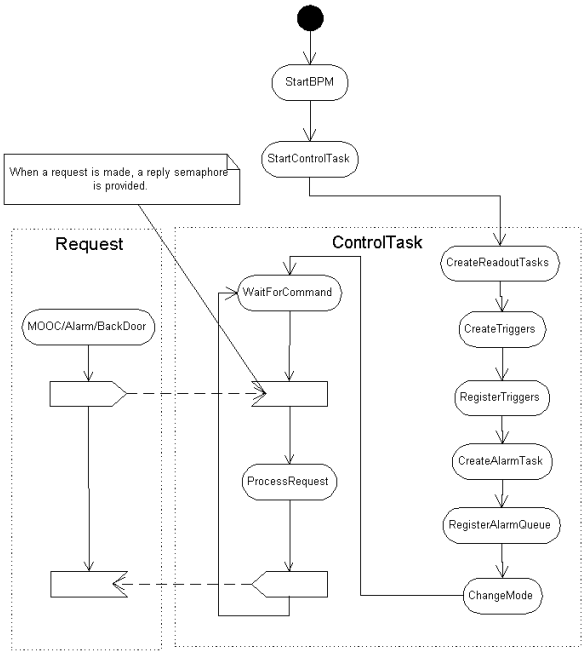


**Figure 10 - *ControlTask* flow**

Similarly, the *DataAcquisitionTasks* run in a closed loop waiting for *Triggers*. Upon the reception of a *Trigger*, the *DataAcquisitionTask* begins the data acquisition process from its *DataSource*, which can be hardware or a software entity. The *DataAcquisitionTasks* are independent of each other but may share some source code (e.g. *TurnByTurn* and *InjectionTbT* in the picture).

Figure 11 depicts several *DataAcquisitionTasks*, but the functionality of some can be combined into only one task in the final implementation. For example, the *FastAbort* may also be responsible for the tasks performed by the *SlowAbort*. It is an implementation choice, and the final decision may be driven by the performance of the options.

The framework also allows a *DataAcquisitionTask* to generate *Triggers* to another *DataAcquisitionTask*. Suppose that there is an *InjectionTbTClosedOrbit* task. It would receive a trigger from the *InjectionTbT* task informing it that new data is in the internal buffer, and a closed orbit can be calculated.

The process of retrieving data from internal buffers is shown in Figure 12. A request comming from Mooc or backdoor is posted on the queue and one of the *BufferReadoutTasks* picks up the request, processes it and return the reply already in the format expected by the online applications (doc #860).

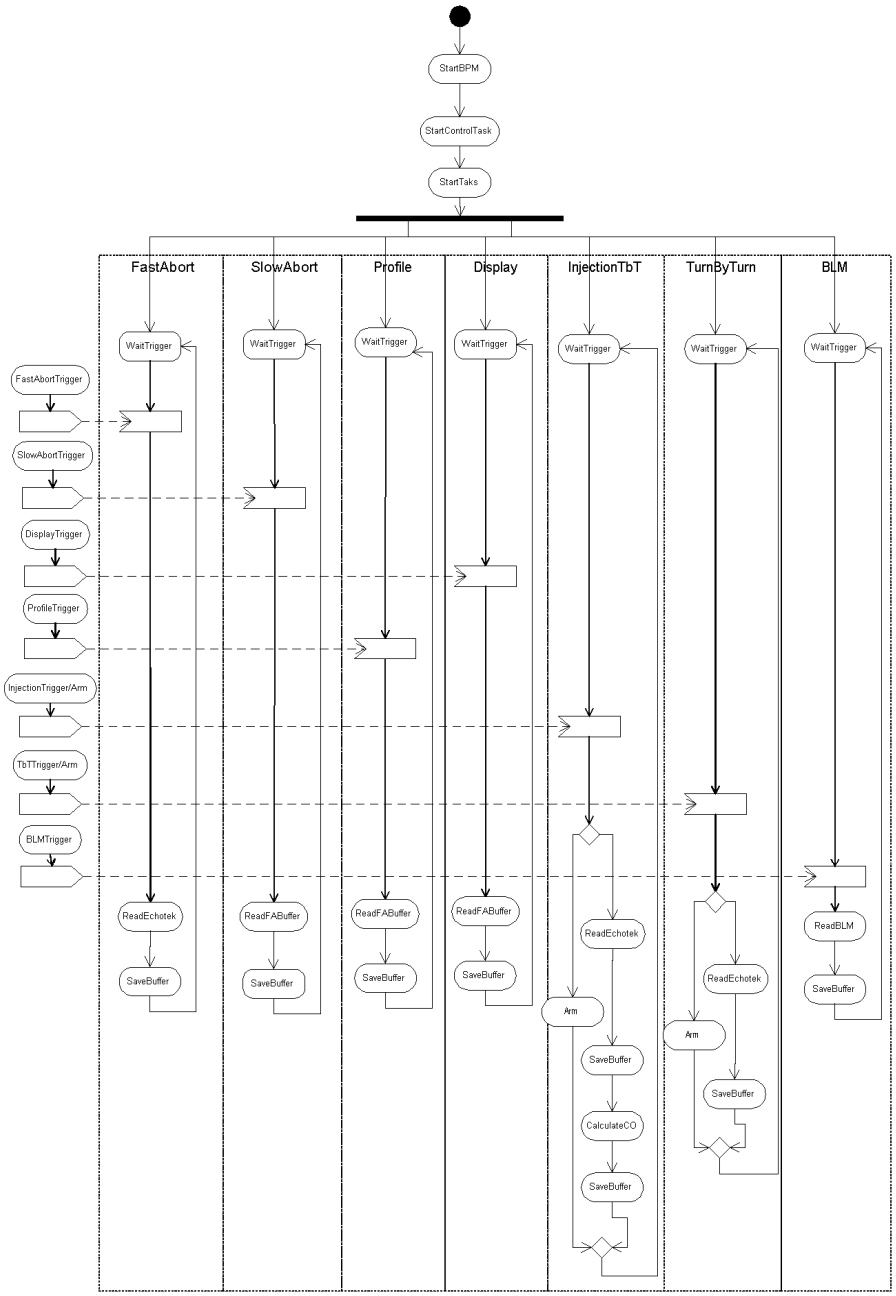


**Figure 11 - *DataAcquisitionTasks* flow**[3]

[3] The tasks depicted in the picture (swim lanes) do not necessarily represent how the system will be implemented. Functionality of tasks can be combined (e.g. the InjectionTbT and TurnByTurn could be one task).

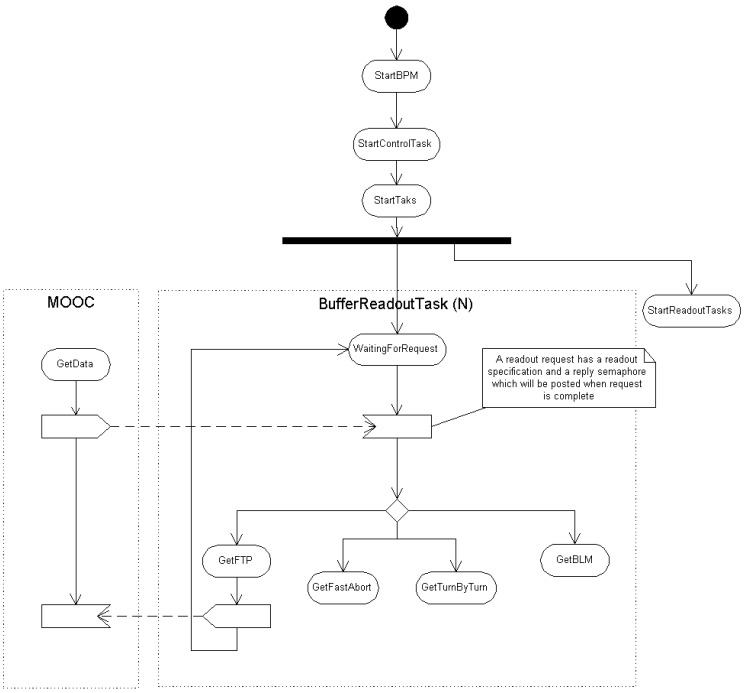


**Figure 12 - *BufferReadoutTask* flow**

## *3.4 Sequence Diagrams*

This section describes common software scenarios for the front-end Tevatron software. The diagrams contain objects of the classes previously discussed and show interactions between them throughout the course of a given scenario. The sequences shown do not correspond exactly to the implementation, but they serve as a guide to understand how objects and classes are related to each other in a dynamic environment. The flow of events starts at the top of the diagram and go downwards, following the string of method calls.

### 3.4.1 Initialization

Figure 13 shows how the objects in the system are first created and what are the expected operations. The entry point is the *BPM* object, which will create the *ControlTask*. The *ControlTask* is responsible for creating most of the objects within the system, it must instantiate the *DataAcquisitionTasks*, the *BufferReadoutTasks*, create the *AlarmTask* and create the *EventGenerators*.

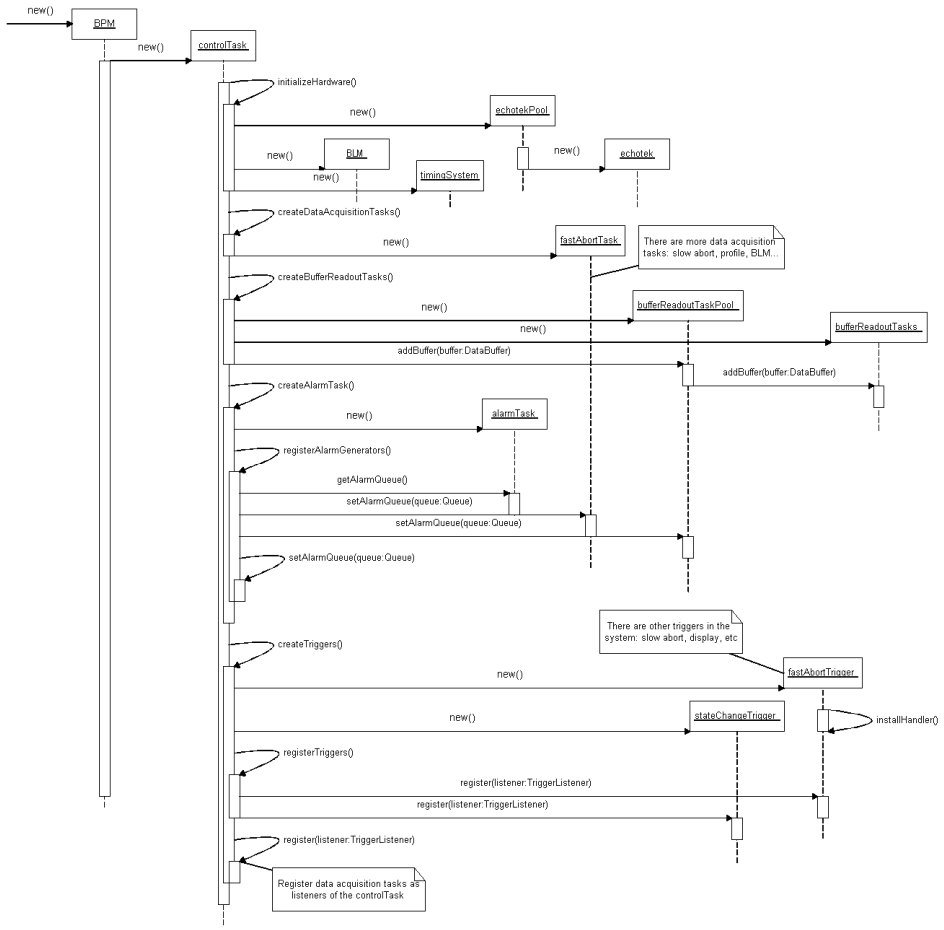


**Figure 13 - Objects creation sequence**

Following the instantiation of the objects in the system, the tasks need to be started in order to do the actual work of data acquisition. The *ControlTask* is responsible for getting them to work, as shown in Figure 14. After starting the tasks, the normal mode of operation is enabled through *changeMode ()*.

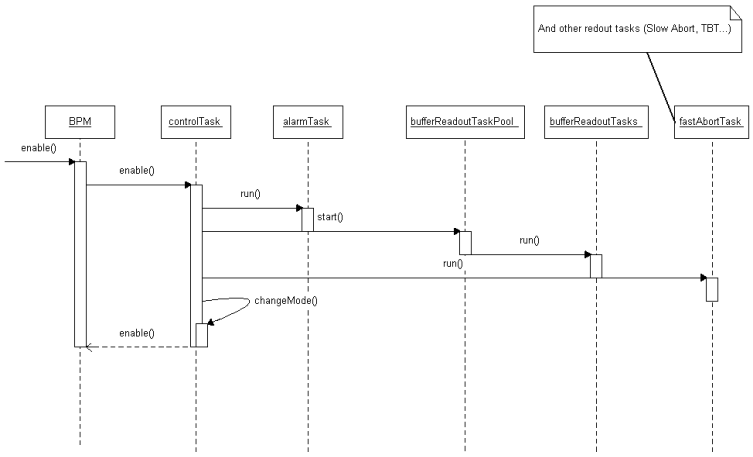


**Figure 14 - Tasks initialization**

### 3.4.2 Mode Change

The system has the ability to change modes of operation when running. The most common modes are closed orbit and turn-by-turn[4]. The closed orbit mode is the default mode of operation. Turn-by-turn mode is enabled on user request or on a programmed *Trigger*. Figure 15 shows the sequence of operations when changing from the default mode to the turn-by-turn mode.

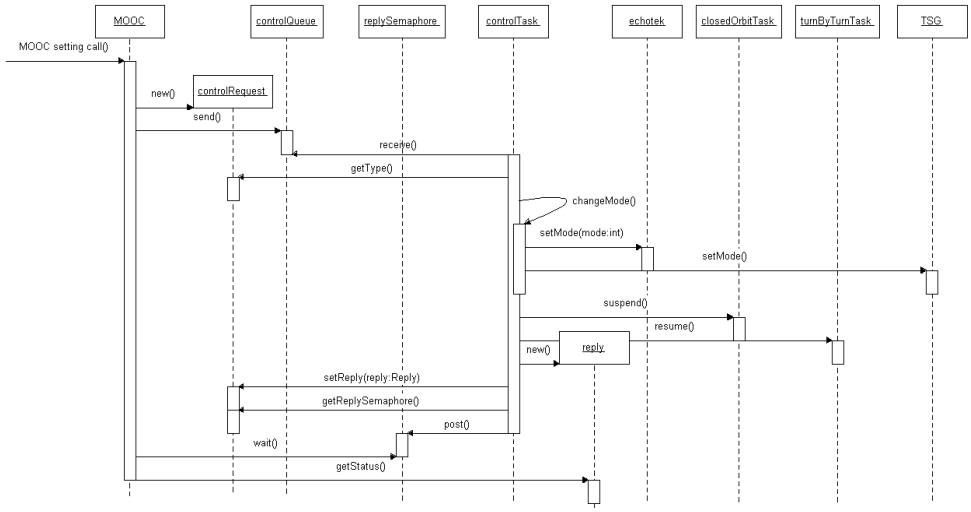


**Figure 15 - Changing modes**

[4] There can be a turn-by-turn request when the system is already in turn-by-turn mode. In this case, the system must halt the current measurement and restart it according to the new specification

In the particular case depicted on Figure 15, the mode change is triggered by a user command from MOOC. The command will be passed down to the *controlTask* as a *ControlRequest* through the *controlQueue*.

The *controlTask* is responsible for changing the mode of operation of the EchoTek boards (by loading a different configuration) and setting the timing system (*TSG*) to the turn-by-turn mode. It is also its job to suspend and resume *DataAcquisitionTasks* according to the mode of operation.

The action of suspending and resuming the *DataAcquisitionTasks* is accomplished by sending control requests via their input trigger queue. When a *DataAcquisitionTask* receives a suspend command it will ignore any triggers from that moment on. Upon the reception of a resume command, the *DataAcquisitionTask* starts to process triggers.

After the new mode is enabled, the *ControlTask* returns a *Reply* to the MOOC framework containing the result of the operation. In the figure the MOOC framework remains blocked in a semaphore after submitting the request and is released upon receipt of the reply, which involves a semaphore post operation.

### 3.4.3 Buffer Readout

Buffer readout operations follow the structure defined in Figure 16. Similarly to the situation depicted in Figure 15, a request comes through the MOOC framework, a *BufferReadoutRequest* is created and passed to the *BufferReadoutTasks*. Those have to check the type of the request, get the data from the internal buffer and pack it in the format described by the document 860.

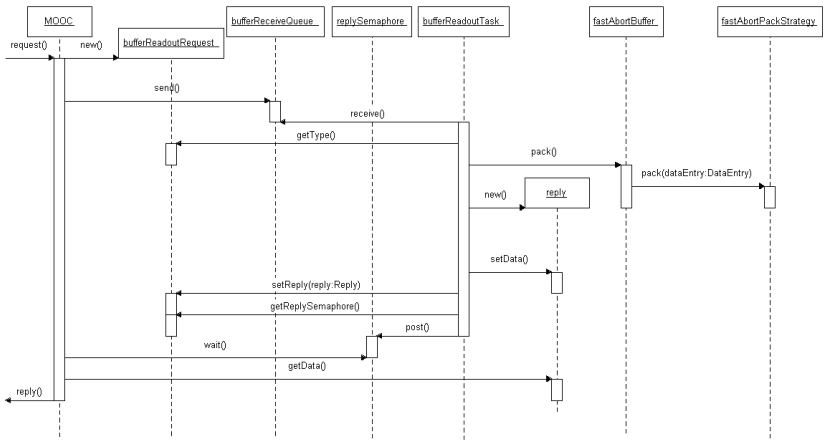


**Figure 16 - Fast abort buffer readout**

Data packed from the internal buffer is passed to a *Reply*, which will be received by the MOOC framework upon the completion of the operation. Similarly to the sequence described in the above section, the MOOC framework remains blocked on a semaphore while the system is processing its request. The semaphore is released after the *Reply* is ready.

This buffer readout structure isolates MOOC from software internals. This allows standard communication with other frameworks, such as the backdoor. The price to be paid for the isolation of the code is the amount of processing involved.

For operations that require high rate of data readout, the sequence depicted in Figure 16 may not be efficient. For that purpose, it should be possible for the MOOC framework (or backdoor) to have direct access to internal buffers. Since fast time plots are nothing more than a single value, the access should be basically read the first element of the *DataBuffer*. Figure 17 shows this alternative scenario.

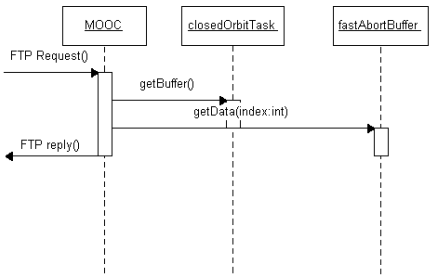


**Figure 17 - Alternative fast time plot readout**

### 3.4.4 Alarms

All tasks in the system are capable of generating *Alarms*. Figure 18 shows the sequence of an alarm generation. A task in the system creates an *Alarm* and it is sent to the *alarmQueue*, which is monitored by the *AlarmTask.* This task decides the criticality of the alarm and send out a MOOC alarm and informs the *ControlTask* that the system is in an alarm state.

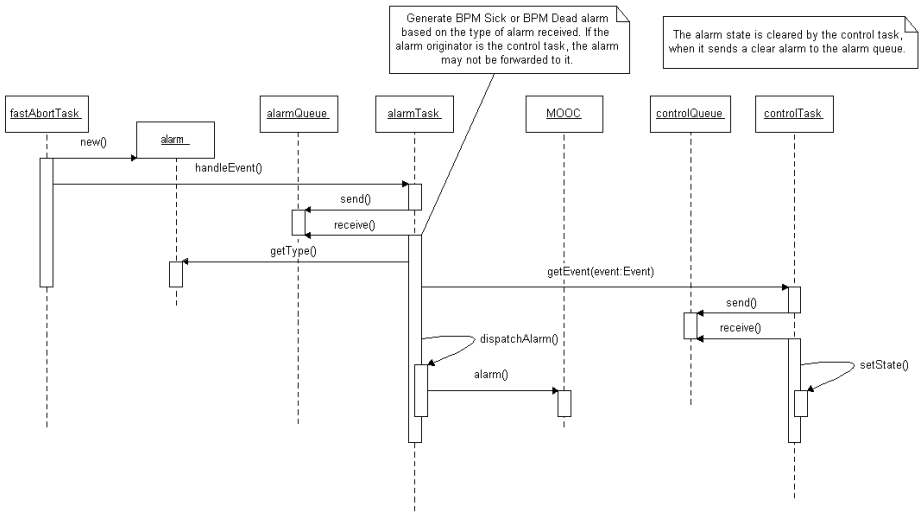


**Figure 18 - Alarm generation**

The alarm state can be cleared by the *ControlTask* through sending a clear message to the *AlarmTask* (see Figure 19).

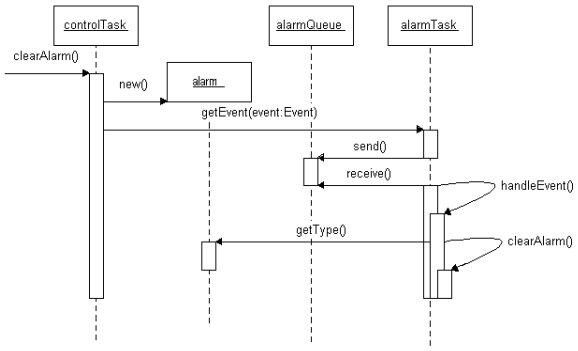


**Figure 19 - Clearing an alarm**

### 3.4.5 Events

Figure 20 shows a generic view of how an *Event* is handled in the system. The *EventGenerators* create *Events*, which are sent to *EventQueues* owned by *EventListeners*. An *EventListener* is usually a task and will receive events from its queue and process them within the *handleEvent ()* method.

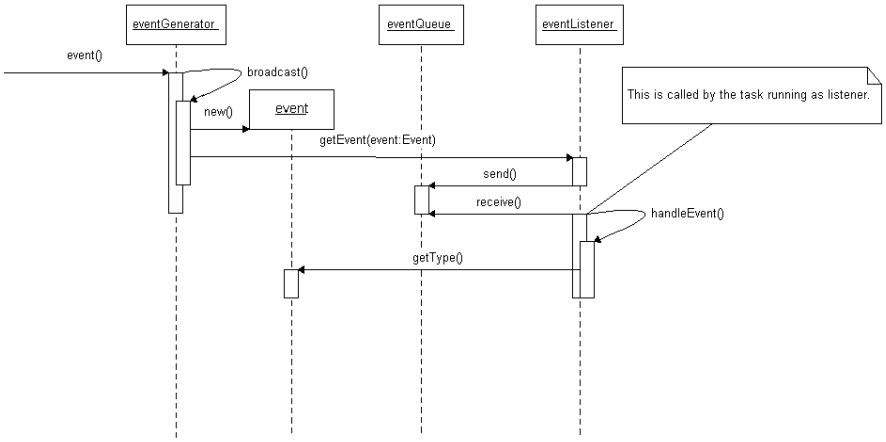


**Figure 20 - Event generation**

A particular case of event handling is shown in Figure 21, where the generator is of the type *StateChangeEventGenerator*, and the receiving side is the *ControlTask.*

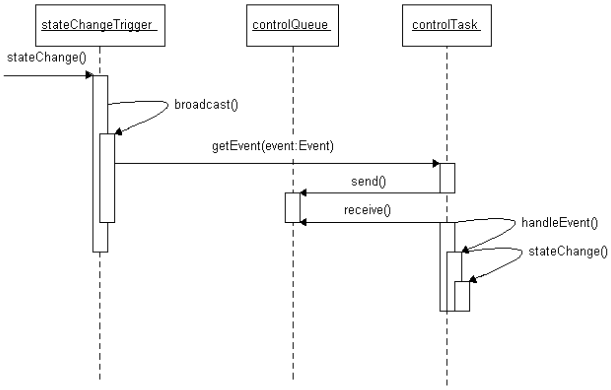


**Figure 21 - State device change**

### 3.4.6 Data Acquisition

The data acquisition process is similar to the event handling scheme on Figure 20. The event in question is a *Trigger*, generated by a *TriggerGenerator*. The event is sent to a *DataAcquisitionTask*, which in turn will acquire data from a *DataSource* and save it to a *DataBuffer*. This process is illustrated in Figure 22.

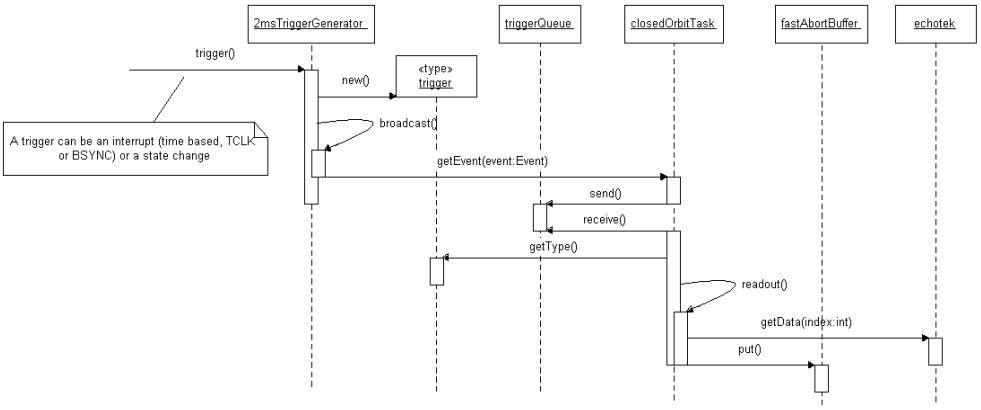


**Figure 22 - Fast abort trigger generation**

## *3.5 Packages*

The system is designed to provide a generic and flexible framework for BPM projects including the Tevatron. For this purpose, the classes are divided in a generic BPM class package and Tevatron BPM package.

### 3.5.1 Generic BPM classes (GBPM)

These are generic classes that form the BPM framework. These should make up a running system with hooks for machine specific code. It contains classes that provide data acquisition, control, configuration management, alarms and buffering. The following classes are part of the GBPM package:

- BPM
- Task
  - ControlTask
  - BufferReadoutTask
  - DataAcquisitionTask
  - AlarmTask
- Queue
- Semaphore
- EventGenerator
  - StateChangeEventGenerator
  - InterruptTriggerGenerator
  - TCLKGenerator
  - TimeTriggerGenerator

- AlarmGenerator
- Event
  - Alarm
  - Request
    - ReadoutRequest
    - ControlRequest
  - Reply
    - ReadoutReply
  - Trigger
- DataSource
  - DataBuffer
- Metadata
- DataEntry
- PackStrategy
- TaskPool
  - BufferReadoutTaskPool
- CalibrationSystem
- TimingSystem

### 3.5.2 Tevatron BPM classes (TBPM)

These are the classes that implement the specific BPM behavior for the Tevatron machine. Code for specific hardware and Tevatron alarms must be implemented in these classes. Table 1 lists the classes that belong to the TBPM package.

| Superclass | Subclasses |
|---|---|
| *BPM* | *TBPM* |
| *DataAcquisitionTask* | *TBPMClosedOrbitTask*<br>*TBPMTurnByTurnTask*<br>*TBPMInjectionTurnByTurnTask*<br>*TBLMFastAbortTask* |
| *AlarmTask* | *TBPMAlarmTask* |
| *DataSource* | *EchoTek* |
| *TimingSystem* | *TBPMTimingSystem (TSG)* |
| *Metadata* | *TBPMMetadata* |
| *PackStrategy* | *TBPMClosedOrbitPackStrategy*<br>*TBPMTurnByTurnPackStrategy*<br>*TBPMTimeSlicedPackStrategy*<br>*TBLMPackStrategy*<br>*TBLMTimeSlicedPackStrategy* |
| *DataEntry* | *TBPMData*<br>*TBLMData*<br>*TBPMChannel* |
| *EventGenerator* | *TCLKEventGenerator*<br>*StateChangeEventGenerator* |

| | |
|---|---|
| *BufferReadoutTask* | *TBPMBufferReadoutTask* |

**Table 1 - TBPM classes**

#### 3.5.2.1 TBPM Buffers

Even though the data source of position, intension and loss data are only the EchoTek boards and the BLM chassis, the system must keep several types of data in different buffers. Some of these data may be the same, what makes them different is the event that triggered its acquisition. According to AD document #903, these are the buffers that must be implemented in the system:

BPM buffers:

- Fast Abort Buffer (array)
- Slow Abort Buffer (array)
- Fast Time Plot Buffer
- Profile Frame Buffer (array)
- Display Frame Buffer
- Snapshot Buffer

BLM buffers:

- Fast Abort Buffer (array)
- Display Buffer
- Fast Time Plot Buffer

These buffers are realized in the system by the class *DataBuffer*. The Fast Time Plot buffers are the only ones that do not have its own memory location. When handling a FTP request the system will return the latest value from the Fast Abort Buffer.

Data from the buffers are read by the *BufferReadoutTask* and organized by a *PackStrategy* according to the structures defined in AD document #860.

## *3.6 Implementation*

This section is a guideline for the implementation of the system. It is divided into two parts: the first one contains the elements related to the generic BPM framework and the second lists the components of the Tevatron system.

It is highly recommended that every class have a unit test associated to it. The tests should call all methods from the classes and check the returning data and status.

### 3.6.1 Building The Generic Framework

Implementation of classes is independent, otherwise noted that there are requirements. First level of elements can be implemented in parallel, while elements within (a – z) usually require sequential implementation. Here is the list of implementation tasks:

1. Buffers
    a. Implement *DataSource*
        This is a generic class to provide means to get data. It can return data points or data arrays. The returning data are of the generic type *DataEntry*.
    b. Implement subclass that generates a known pattern (e.g. *TestDataSource*)
        We need a data source class capable of generating predefined patterns for testing, debugging and to provide diagnostics.
    c. Implement *DataEntry*
        It is a generic data point, it does not define the type of data it will carry, this should be defined in its subclasses. It contains minimum information such as a time stamp and the status of the data.
    d. Implement Subclass of *DataEntry* (e.g. *TestDataEntry*)
        This would be a class for testing and debugging the code. Can contain a simple integer as the data.
    e. Implement *DataBuffer*
        Generic class that allocates memory for containing *DataEntries*. Has methods for controlling its contents and is able to return pointer to elements within the array.
    f. Implement *PackStrategy*
        Generic class that provides the method interface for packing data. This will be used by the system when the user requests data. Data has to be read from the internal buffer and repackaged into some format. The subclasses of *PackStrategy* will provide the appropriate algorithm for packing the data according to the users request.
    g. Implement subclass of *PackStrategy* (e.g. *TestPackStrategy*)
        For completing the *Test* environment there is the need of a *PackStrategy* for our *TestDataEntry*. Should be a simple class that implements an algorithm for packing *TestDataEntry* type of data. It will follow the interface defined in the *PackStrategy* class.
2. Wrappers
    a. Implement *Task*
        This is a VxWorks task wrapper. It will allow a class to be a task by providing a *run ()* method, which is called by *start ()*. *start ()* will encapsulate the system call *taskSpawn*. The class also should take care of operating system errors that may ocurr and should keep information such as priority and task id as attributes.
    b. Implement *Queue*
        Wrapper for the VxWorks queues. Should take care of operating system errors and keep information about its status. Should provide methods for retrieving current status and statistics.
3. Events (requires 2b)

a. Implement *Event*
   An *Event* is a generic container for any kind of event in the system. An event can be a 2ms trigger generated by a timer; a TCLK just received, an interrupt coming from the timing system.
b. Implement *EventListener* (requires 2b)
   The *EventListener* is a class that has an input queue, through which it receives *Events*. Subclasses of *EventListener* will be able to receive *Events*. It also provides interfaces for handling the received events.
c. Implement *EventGenerator*
   This class provides means to broadcast *Events* to *EventListeners*. It contains a list of listeners, and when an event is generated it is passed to the members of the list. The class provides calls for adding and removing listeners.
d. Implement subclasses of *Event* (*Trigger*, *Request*, *ReadoutRequest*, *ControlRequest*, *Reply*, *ReadoutReply*)
e. Implement *InterruptEventGenerator*
   Contains interface for install, enable and disable an interrupt handler. The interrupt handler is a method within the class.
f. Implement *TimeEventGenerator*
   Subclass of *InterruptEventGenerator*. Configures software timer to call the interrupt handler.

4. Alarms
   a. Implement *Alarm* (requires 3a)
      An *Alarm* is an event that is generated if certain conditions are met, such as an EchoTek time out.
   b. Implement *AlarmGenerator*
      Provides the ability to send *Alarms* to the *AlarmTask*.
5. Tasks (requires on 2a)
   a. Implement *ControlTask*
      This class contains code for managing a generic data acquisition environment. Provides calls for adding *DataAcquisitionTasks* and buffers.
      i. Implement *Metadata*
         This class contains any generic metadata associated with the data acquisition system. For example the current status of the system.
   b. Implement *DataAcquisitionTask*
      The actual data acquisition work is performed by the *DataAcquisitionTask*. Generically this task repeats the following operation upon the reception of a trigger: read the *DataSource*, save *DataBuffer* and wait for another *Trigger*. Specialized subclasses can implement code for dealing with specific hardware. This class should also be available to use in an actual system without adding any code, if the *DataSource* and *DataBuffers* don't require any special handling (e.g. the *BPMDisplayTask* on Figure 2 may be only a *DataAcquisitionTask*

whose *DataSource* is the *BPMFastAbortBuffer* and whose *DataBuffer* is a *BPMDisplayBuffer*).

c. Implement *BufferReadoutTask*
   This class handles user data requests. Data is read from internal buffers and formatted according to a *PackStrategy* before being sent to the user.

d. Implement *AlarmTask*
   This task receives internal *Alarms*, and decide if external alarms should be generated.

6. Task Pool
   a. Implement *TaskPool* (requires 2a)
      Provides a generic pool of tasks, providing interface for managing a group of identical tasks.
   b. Implement *BufferReadoutTaskPool* (requires 5p)
      Implements a pool of *BufferReadoutTasks* by adding specific functionality for managing and controlling that type of task. For example, this specialized pool need to implement calls to add/remove buffers to the tasks.
7. Hardware Support
   a. Implement *TimingSystem*
      Defines basic interface for a BPM timing system, does not have implementation of specific timing system. Subclasses must define the behavior of the timing system.
   b. Implement *CalibrationSystem*
      Defines basic interface for a BPM calibration system, does not have actual implementation. Subclasses must define the behavior of the calibration system.
8. External Communication
   Implementation of calls (set of classes and wrappers) that can be made from ACNET/MOOC for data request, data acquisition specifications and control requests.

At the end of the implementation of the generic framework it is expected to have a test version of the system running, generating fake data, receiving comands and requests from users.

### 3.6.2 Building Tevatron BPM Software

The implementation of all classes for the Tevatron specific system are independent. Any requirement on input/output data can be fulfilled by using Test classes from the generic framework. For example, it is not necessary to have the EchoTek class in place to generate data, one can use the *TestDataSource* class for that purpose or implement a *TestEchoTek* class which generates simulated data. The list of implementation tasks follows:

1. BPM hardware
   a. Implement *EchoTek*

Contains all EchoTek related code. Provides interface for configuring the board, set diagnostics mode, enable debugging, etc.

b. Implement *EchoTekPool*

Represents a set of *EchoTek* objects. Has the ability to probe the VME bus for boards and add them to the pool automatically. Provides access to a single board and is able to send commands to all boards.

2. BLM hardware
   a. Implement *TBLM*

      Software representation of the BLM hardware, providing the interface for reading and writing to BLM registers.
3. Timing system
   a. Implement *TBPMTimingSystem*

      Contains the implementation of the interfaces defined by the *TimingSystem* class. This class is able to configure, diagnostic, enable and disable the Tevatron BPM timing system.
4. Control
   a. Implement *TBPMControlTask*

      Extends the functionality of the *ControlTask* class by adding code for dealing with specifics of the Tevatron BPM system. The *TBPMControlTask* is responsible for creating the *DataAcquisitionTasks* in the system, has the knowledge on how to proceed on mode changes (when to configure the EchoTek cards and the timing system). Controls the overall system status; is able to clear internal alarms; receive TCLK triggers and forward them to the *DataAcquisitionTasks*; controls what *DataAcquisitionTasks* are currently active (receiving triggers or ignoring them).
5. Data
   a. Implement *TBLMData*

      Class or structure containing a BLM data entry.

   c. Implement *TBPMData*

      Class or structure containing a BPM data entry.

   d. Implement *TBPMChannel*

      Class or structure containing information about a single BPM channel.

   e. Implement *TBPMClosedOrbitPackStrategy*

      Implementation of the pack strategy for closed orbit measurements. This class receives *DataEntries* from a *DataBuffer* and generates a structure that is sent to the user in response to a request.

   f. Implement *TBPMTurnByTurnPackStrategy*

      Pack strategy for turn-by-turn measurements.

   g. Implement *TBPMTimeSlicedPackStrategy*

      Pack strategy for time sliced closed orbit requests.

   h. Implement *TBLMPackStrategy*

      Pack strategy for BLM requests.

   i. Implement *TBLMTimeSlicedPackStrategy*

      Pack strategy for BLM time sliced requests.
6. Data acquisition

a. Implement *TBPMClosedOrbitTask*
   This class has to deal directly with the EchoTek boards and the timing system. It does not configure the EchoTek and timing boards (that is done by the *TBPMControlTask*), only reads data or get interrupts from them. Cannot use the generic *DataAcquisition* class for readout.
b. Implement *TBPMTurnByTurnTask*
   This is a specialized class and like the *TBPMClosedOrbitTask* it has to communicate with the EchoTek boards and the timing system.
c. Implement *TBPMInjectionTurnByTurnTask*
   This is a specialization of the *TBPMTurnByTurnTask*. There are not many additions to the code. In the implementation it is possible that a subclass is not even necessary to implement this feature.
d. Implement other data acquisition tasks: The tasks shown in Figure 2 are able to use the generic algorithm implemented in the *DataAcquitisionTask* class. They basically will read data from an input buffer and save it to an output buffer. With the exception of the *TBLMFastAbortTask*, which will get input data from the BLM chassis.

7. Buffer Readout
   a. Implement *TBPMBufferReadoutTask*
      Add functionality specific to the Tevatron BPM system (the generic *BufferReadoutTask* may be enough).
8. Events
   a. Implement *StateChangeEventGenerator*
      This class provides the ability to receive state device changes and generate internal events.
   b. Implement *TCLKEventGenerator*
      This class is able to receive predefined TCLKs and generate internal events when they are received.
9. Alarms
   a. Implement *TBPMAlarmTask*
      Contains code for handling internal alarms and decide when to generate external alarms. Can change the system state to ALARM, and is able to receive clear alarm commands from the *TBPMControlTask*.

# 4 Appendix

## *4.1 The Recycler Software*

The current recycler software provides configuration, read out and diagnostics capability for a hardware setup specific to get proton position in the recycler ring. The hardware involved consists of:

- Motorola MVME processor board with two IP320 ADC and one IPUCD TCLK/MDAT
- IP carrier containing four IPTSG timing signal generator modules developed at Fermilab.
- EchoTek boards

The software configures all timing devices and the EchoTek boards according to read out specifications passed through ACNET. The software allows the recycler BPM to run in the following modes:

- Background Flash;
- Flash;
- Closed Orbit;
- Turn-by-Turn;
- Turn-by-Turn Scan.

The following subsections describe the current software from the viewpoint of management, control and buffering techniques.

### 4.1.1 Control

The system has a main control task (ArmEventTask), which is responsible for spawning tasks to handle specific modes of operation. For example: RepetitiveFlashTask and TurnByTurnTask.

When switching modes, the ArmEventTask has to start a new task that has the ability to configure and conduct the read out for the requested mode. When starting the new mode the new task is responsible for configuring the hardware (EchoTek board and timing system).

The communication between tasks is handled through message queues, for example to signal events such as measurement complete. Message queues are also used to signal triggers received by interrupt handlers.

### 4.1.2 Buffering

The data read out by the tasks is kept in buffers visible inside the BPM class (which is the main class in the system). The buffers have predefined size and are managed by the task that currently use it. Since there is only one readout task running at a time, there are no writing conflicts. Data can be read from the buffers via ACNET or backdoor requests.

### 4.1.3 Readout

Readout tasks wait for signals from the message queue. The interrupt handlers receive the external trigger and send data to the queue, which is immediately received by the readout task.

### 4.1.4 ACNET Communication

In parallel to the readout, the software can receive ACNET/MOOC calls to retrieve data from the buffers. The software is accessed through callbacks, which extract raw data from the BPM buffers and calculates the positions and intensities. The data is packed and returned in the response to the online software.

### 4.1.5 Debug and Diagnostics

The page R25 provides ways to configure the recycler BPMs, such as selection mode of operation, enabling/disabling diagnostics and setting timing constants. It is possible to control any recycler crate from the software. The communication between R25 and the recycler software is via ACNET.

The recycler software also uses the backdoor scheme for getting data using and alternative way to ACNET. It is a client/server facility, where the server is the front-end and the client is a remote application, currently labview.

The server receives requests to change configuration, return data, and turn diagnostics on among others. The server is programmed by adding *Accessors*, which will serve as bridge to the internals of the system.

A new version of the backdoor software allowing peer-to-peer communication is planned.

### 4.1.6 Alarms

Not implemented in the current version of the recycler software.

## 4.2 Class Diagram

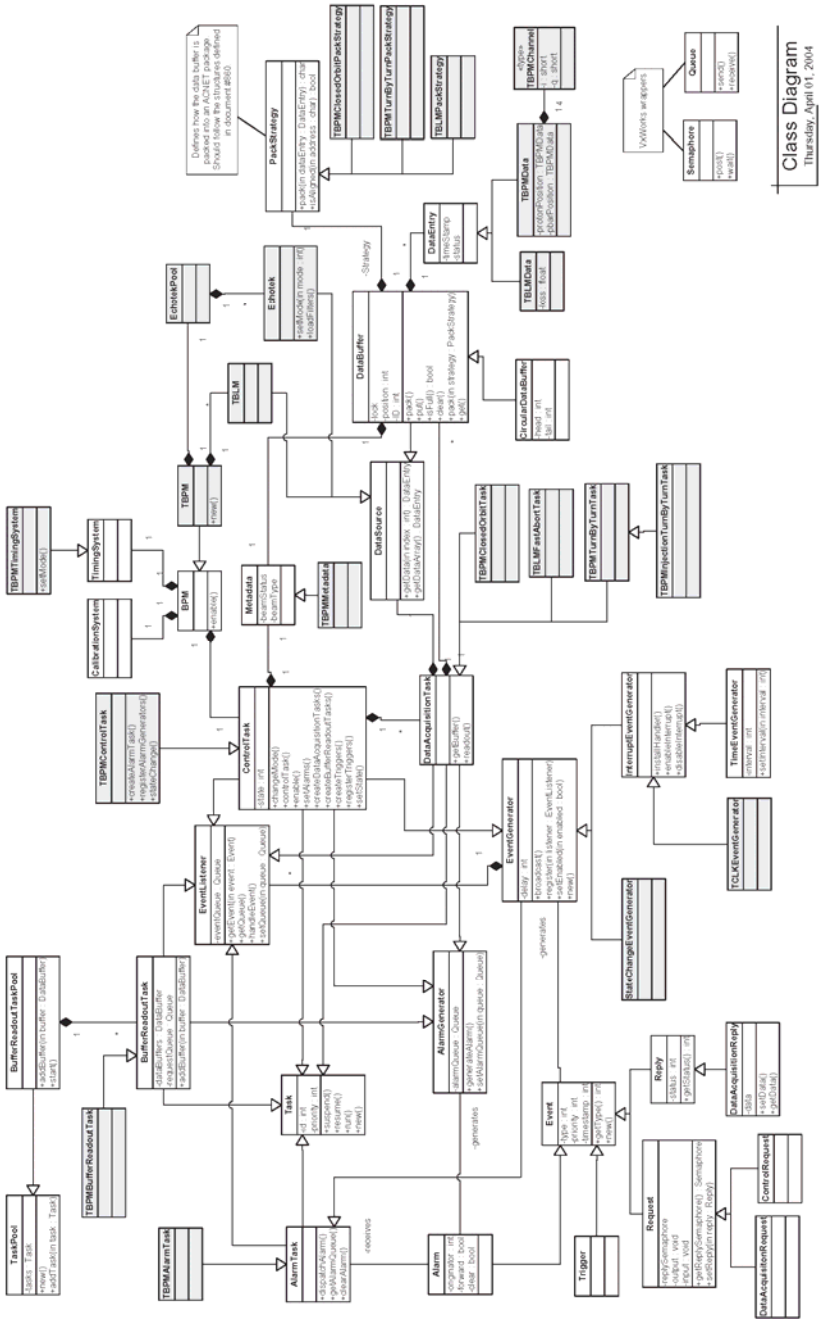


**Figure 23 – Complete TBPM front-end software class diagram (Tevatron specific classes are in gray)**

# 5 Bibliography

[Cockburn] Alistair Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2001.

[Fowler] Martin Fowler, *UML Distilled: A brief guide to the standard object modeling language*, Addison-Wesley, 2003.

Fermilab/AD/TEV
Beams-doc-1067-v13
Date April 13, 2004

# Tevatron BPM Software Design for Data Acquisition

**L.Piccoli**
**M.Votava**
**D.Zhang**
**D.Charak**

## Introduction

## References

## Change Log

| Version | Issue Date | Description of Change |
|---|---|---|
| 1.0 | March 10, 2004 | Original |
| 2.0 | March 11, 2004 | |
| 3.0 | March 12, 2004 | |
| 4.0 | March 15, 2004 | |
| 5.0 | March 16, 2004 | |
| 6.0 | March 24, 2004 | |
| 7.0 | March 26, 2004 | |
| 8.0 | March 26, 2004 | |
| 9.0 | March 31, 2004 | |
| 10.0 | March 31, 2004 | |
| 11.0 | April 1st, 2004 | |
| 12.0 | April 2nd, 2004 | |
| 13.0 | April 13, 2004 | |

TBPMrevdoc1067.doc

**Concurrence**

Following persons reviewed and concurred with the content of this document.

Steve Wolbers 5/19/04
Steve Wolbers, Project Manager (date )

Robert C. Webber 6/23/04
Bob Webber, Deputy Project Manager (date )

Jim Steimel 5-12-04
Jim Steimel, Technical Coordinator (date )

Brian Hendricks 5/19/04
Brian Hendricks, Subsystem manager (date )

Following persons reviewed this document:

Jim Kowalkowski
Duane Voy
Charles Briegel

Bakul Banerjee Page 2 5/12/2004