Fermilab/BD/TEV
Beams-doc-1067-v17
July 23, 2004
Version 17.0

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 5

2 PROPOSED TEVATRON BPM SOFTWARE 6

2.1 CONTROL 6
2.2 BUFFERING 6
2.3 DATA ACQUISITION 7
2.4 ACNET COMMUNICATION 7
2.5 BUFFER READOUT 7
2.6 DEBUG AND DIAGNOSTICS 8
2.7 CALIBRATION 8
2.8 SOFTWARE DIAGRAM 8

3 SOFTWARE DESIGN 11

3.1 USE CASES 11
3.1.1 Initialization 12
3.1.2 Mode Change 13
3.1.3 Buffer Readout 14
3.1.4 Diagnostic 15
3.1.5 Alarm 15
3.1.6 Data Acquisition 16
3.1.7 State Device Change 17
3.1.8 Configuration Change 17
3.1.9 Calibration 18
3.2 FRONT-END EVENTS 19
3.3 ARMING AND TRIGGERING 19
3.4 TEVATRON BPM DATA BUFFERS 20
3.5 TEVATRON METADATA 20
3.6 TEVATRON BPM STATE DIAGRAM 23
3.7 CLASS DIAGRAMS 24
3.7.1 Tasks 24
3.7.2 Controls 25
3.7.3 Events 26
3.7.4 Event Listeners and Generators 27
3.7.5 Data 28
3.7.6 Alarms 30
3.8 TIMING DIAGRAMS 30
3.9 ACTIVITY DIAGRAMS 33
3.10 SEQUENCE DIAGRAMS 36
3.10.1 Initialization 36
3.10.2 Mode Change 41
3.10.3 Buffer Readout 42
3.10.4 Alarms 43
3.10.5 Events 44
3.10.6 Data Acquisition 45
3.11 PACKAGES 46

3.11.1 Generic BPM classes (GBPM) ... 46
3.11.2 Tevatron BPM classes (TBPM) ... 47
3.12 IMPLEMENTATION ... 48
3.12.1 Building The Generic Framework ... 48
3.12.2 Building Tevatron BPM Software ... 51
4 APPENDIX ... 54
4.1 CLASS DIAGRAM ... 54
5 BIBLIOGRAPHY ... 55

Table of Figures

Figure 1 - Proposed tasks, queues, command and data flow ... 9
Figure 2 - Tasks for the TBPM system ... 10
Figure 3 – Tevatron BPM front-end software use cases ... 11
Figure 4 - Tevatron BPM state diagram ... 23
Figure 5 - Data acquisition task state diagram ... 24
Figure 6 - Class diagram for tasks in the system ... 25
Figure 7 - Main control classes ... 26
Figure 8 – Handling events in the system ... 27
Figure 9 - Event generators ... 28
Figure 10 - Event listeners ... 28
Figure 11 - Reading and saving data ... 29
Figure 12 - Buffer readout related classes ... 29
Figure 13 - Alarm classes ... 30
Figure 14 - Timing diagram for the BPM fast abort DAQ ... 31
Figure 15 - Timing diagram for BPM fast and slow abort DAQ and BLM fast abort DAQ ... 31
Figure 16 - Timing diagram for a turn-by-turn measurement ... 32
Figure 17 - *ControlTask* flow ... 34
Figure 18 - *DataAcquisitionTasks* flow ... 35
Figure 19 - *BufferReadoutTask* flow ... 36
Figure 20 – Initialization sequence ... 37
Figure 21 - Hardware initialization sequence ... 38
Figure 22 - Metadata initialization sequence ... 38
Figure 23 - Buffer initialization sequence ... 39
Figure 24 - Data acquisition tasks initialization sequence ... 39
Figure 25 - Alarm initialization sequence ... 39
Figure 26 - Event generators initialization sequence ... 40
Figure 27 - Tasks initialization ... 40
Figure 28 - Changing modes ... 41
Figure 29 - Return to close orbit mode ... 42
Figure 30 - Fast abort buffer readout ... 42
Figure 31 - Alarm generation ... 43

Figure 32 - Clearing an alarm 43
Figure 33 - Event generation..... 44
Figure 34 - State device change 44
Figure 35 - Fast abort trigger generation 45
Figure 36 - Turn by turn data acquisition 46
Figure 38 – Complete TBPM front-end software class diagram 54

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

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. Many of its components can be reused on the Tevatron systems, such as timing control modules and data acquisition 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 commands 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 data acquisition tasks run, the control task must configure the EchoTek boards and the timing hardware. On the recycler software, the configuration is done by the data acquisition 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. It is also possible to run simple control commands within the context of MOOC, avoiding the queueing overhead. The control task also is able to receive certains event, such as specific TCLKs (e.g. $47 for Tevatron abort).

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, the implementation should consider the use of semaphores or other mechanism to protect 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 should be handled as transparently as possible. 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

[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.

(e.g. Fast Abort Buffer vs. Slow Abort Buffer (a more detailed list of buffers is given in sections 3.4 and 3.11.2.1)).

2.3 Data Acquisition

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

The input event can be generated by other tasks in the system or by interrupt handlers. The control task can issue an event for arming a turn-by-turn measurement, which is sent to the input queue of the turn-by-turn task. The data acquisition task uses it to prepare itself for the acquisition, which happens when the timing board generates an interrupt. The interrupt handler creates an event that goes into the task's queue, informing that the EchoTek boards can be readout.

Similarly, for TCLKs, when an interrupt is generated by the PMCUCD card, the interrupt handler creates an event and puts it into the event queue of the task that is expecting that particular TCLK.

In general terms the data acquisition cycle is: data acquisition is armed; task receives an event task performs the acquisition; data is saved into a data buffer.

2.4 ACNET Communication

All communication via ACNET will be handled by callbacks, which in turn will invoke the BPM system. There are basically two types of commands coming from ACNET: control commands and data request commands. The bottom part of Figure 1 shows the interaction between MOOC and the front-end system.

Simple commands can be handled directly at the callback level. An example of simple command is the change of a single EchoTek channel configuration through the SETDAT protocol. For commands that require more complicated actions, such as changing the mode of operation, the callback posts a request (or event) into the control task's input queue.

Data request commands on the other hand are handle directly by the callbacks invoked by MOOC. The callbacks will select the buffer that was requested, pack the data into the format expected by the online side and send it.

2.5 Buffer Readout

The user requests for reading data buffers are received via MOOC/ACNET according to the above section. The callback provided to MOOC for handling data requests contains access to all buffers in the system and also knows how to pack the data according to the online specifications. It is important to notice that this code will run within the MOOC context and not in other task's context (as when handling mode changes).

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 the same callbacks used by MOOC.

2.7 Calibration

The calibration of the system is done on the offline processing. However, the front-end software is required to know when a data acquisition is generating data for offline calibration. Any data generated has to be tagged as calibration data. Additionally, the front-end is able to change configuration of the timing system and EchoTek boards for calibration operations. The data also will have metadata describing the configuration used for data acquisition.

2.8 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. The boxes on top of MOOC represent the callbacks used to direct control commands and to retrieve data from the buffers.

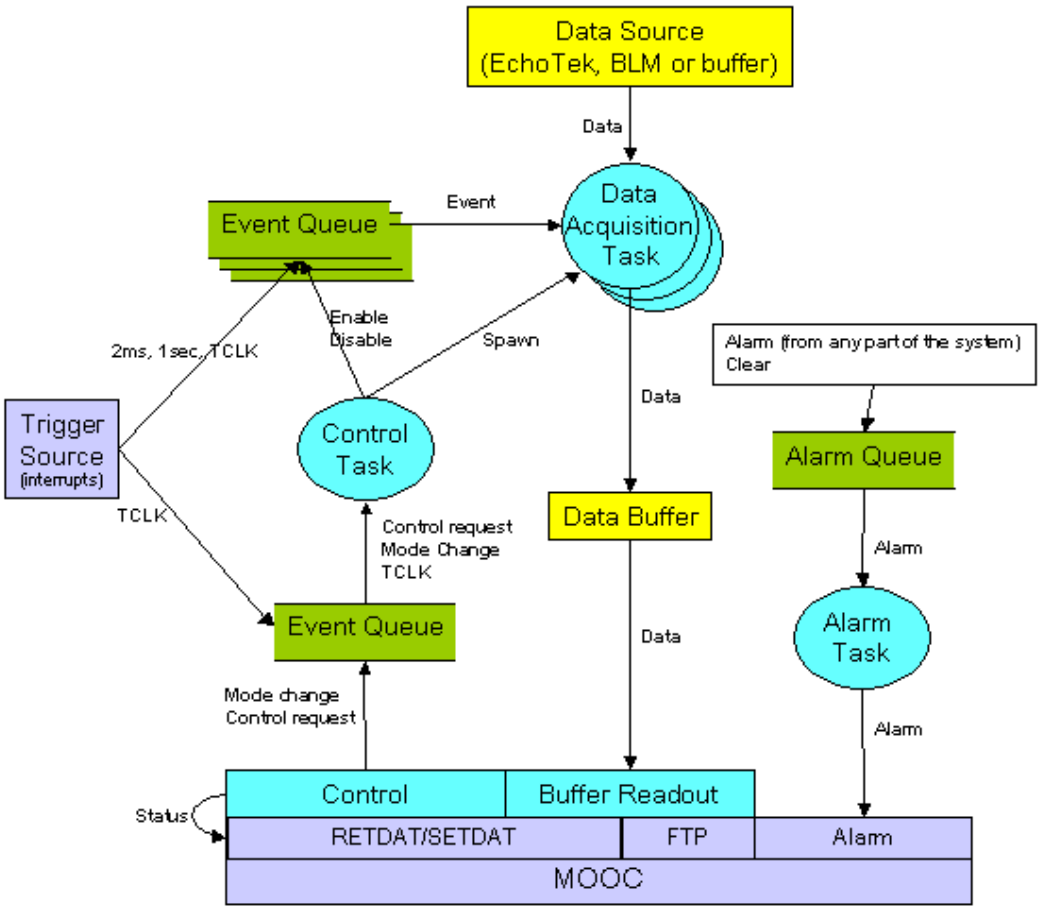

Figure 1 - Proposed tasks, queues, command and data flow

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. It also shows the control task handling directly the timing, diagnostic and calibration hardware, besides the EchoTek cards.

The figure bellow also defines the TCLKs received by the system. The control task receives TCLK $71, $77, $4D and $47. TCLK $71 signals prepare for beam; TCLK $77 signals an arm turn-by-turn measurement; $47: beam has been aborted; and $4D: arm injection turn-by-turn measurement. Other TCLKs are directed to data acquisition tasks, such as TCLK $75 for a BPM profile measurement, TCLK $78 for BPM display measurement and TCLK $76 for BLM profile measurement.

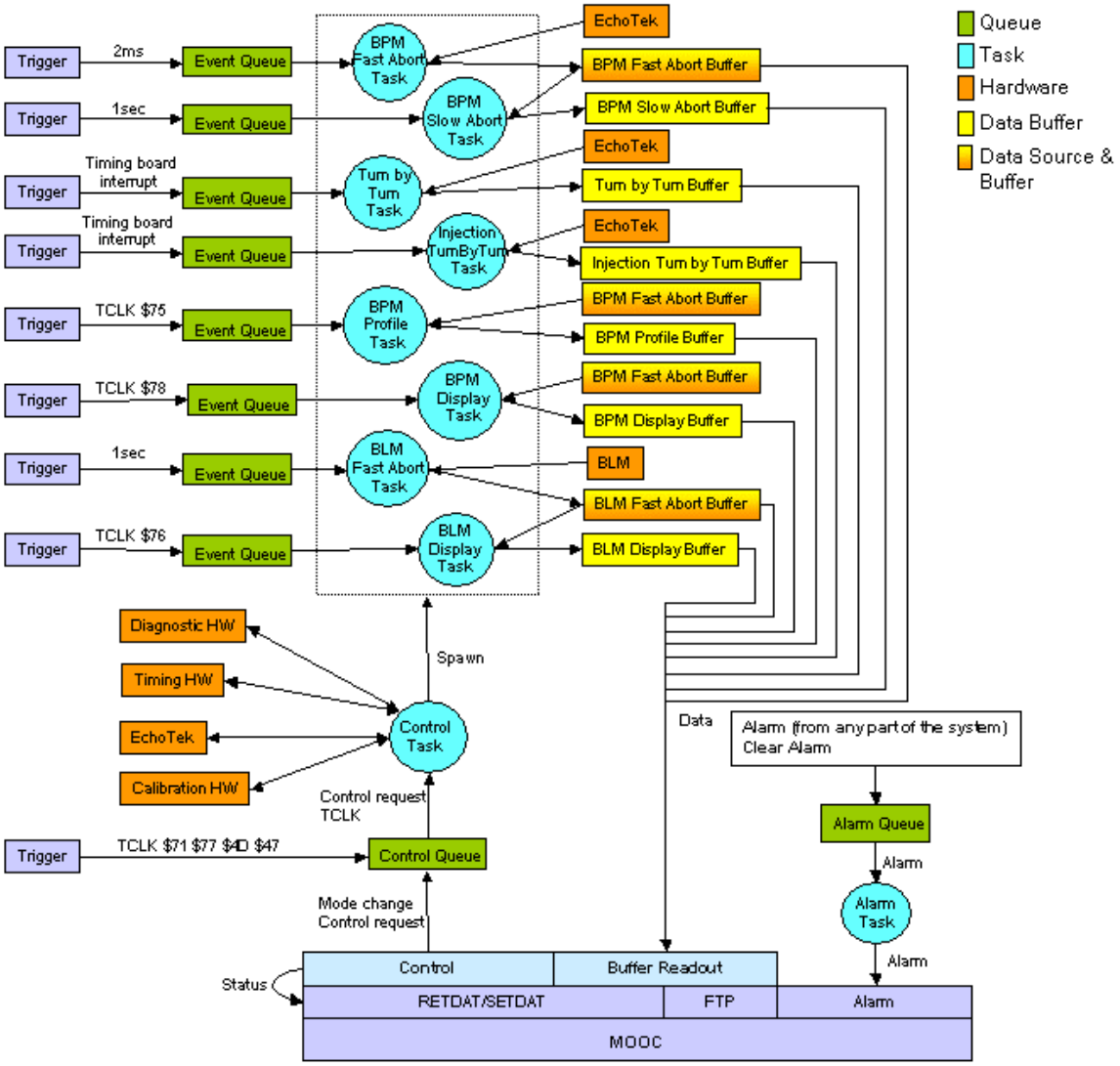

Figure 2 - Tasks for the TBPM system

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 *Event*. 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* or *calibration*. On any of these interactions there can be *alarms*, which is handled by a separate use case.

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

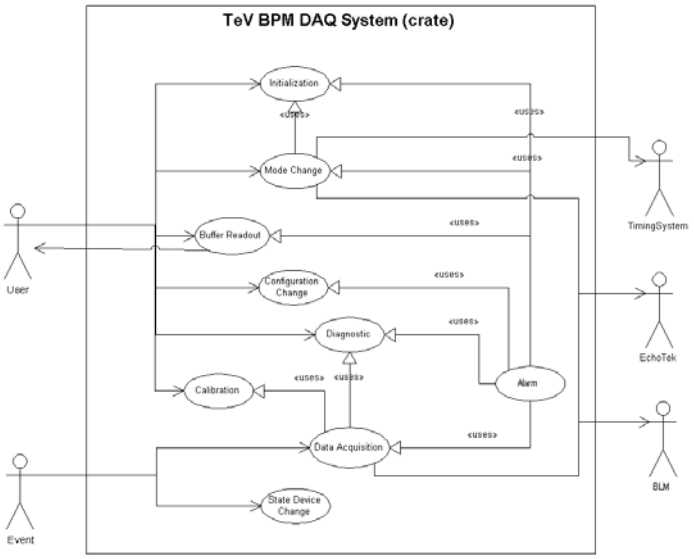

Figure 3 – Tevatron BPM front-end software use cases

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
3. Configuration for the crate is downloaded
4. Configuration task initializes status of state devices
5. Control task initializes EchoTek hardware
    a. EchoTek hardware is tested (optional)
6. Control task initializes BLM hardware
    a. BLM hardware is tested (optional)
7. Control task initializes timing system
    a. Timing hardware is tested (optional)
8. Control task creates data acquisition tasks
9. Control task allocates internal buffers
10. Control task creates 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 through ACNET variable
    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 through ACNET variable

3. EchoTek card(s) did not pass test (5.a)
    a. Generate internal alarm
    b. Set ALARM state
    c. Report error to user through ACNET variable

4. BLM board(s) did not pass test (6.a)
    a. Generate internal alarm
    b. Set ALARM state
    c. Report error to user through ACNET variable

5. Timing system did not pass test (7.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 through ACNET variable

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 to user through ACNET variable

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 to user through ACNET variable

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. Callback for buffer data readout is invoked by MOOC
3. The request is verified and the buffer is selected
4. Buffer is read and converted to online format (see document #860 for structures)
5. Data is sent back to the user

3.1.3.3 Alternative Flows

1. Request is not valid (3)
    a. The data requested does not exist or is out of boundaries
    b. Return error stating the problem found

2. No data in the buffer (4)
    a. Return error specifying that there is no data to be read

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 performs 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 to the user through ACNET variable

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. Alarm task evaluates the priority of the alarm
6. Alarm task generates 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, echotek and BLM. 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 (EchoTeks, BLMs 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 (TCLK or time trigger)
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. 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. The configuration is changed by the control task.

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 handled at the callback level – skip to steps (2, 3)

2. Request is not valid (4)
    a. Generate error to user through ACNET variable
    b. Do not change configuration
    c. Generate internal alarm

3. Configuration cannot be changed (e.g. system is in turn-by-turn mode) (5)
    a. Generate error to user through ACNET variable
    b. Generate internal alarm

3.1.8.4 Preconditions

System is initialized.

3.1.8.5 Postconditions

New configuration has been applied to the system.

3.1.9 Calibration

3.1.9.1 Description

This use case shows the steps that allow the user to take a calibration run with the system. This use case is identical to the Data Acquisition use case (section 3.1.6). The difference is that data returned to the online user is tagged as ‘calibration data’.

3.1.9.2 Basic Flow of Events

1. Tag data as being ‘calibration data’
2. Use Data Acquisition use case

3.1.9.3 Alternative Flows

Same for the Data Acquisition use case.

3.1.9.4 Preconditions

Same for the Data Acquisition use case.

3.1.9.5 Postconditions

Same for the Data Acquisition use case.

3.2 Front-End Events

The front-end software is composed by several tasks running concurrently. The communication between the tasks happens via message queues. Tasks are able to send and retrieve information from the queues.

The information sent and received from the queues is called event. An event is a simple data structure that signals that something has happened and some action has to be taken by the receiver.

One example of usage of an event is when the front-end receives a TCLK. The TCLK is first serviced by an interrupt handler, which fills an event with the TCLK information and sent it to the queue of the task that is waiting for that specific TCLK.

Similarly, when EchoTek boards are new data from from a turn-by-turn measurement, the timing board generates an interrupt that is caught by an interrup handler, which in turn creates the event with information about the new measurement and send it to the task that is waiting for the turn-by-turn measurement to complete.

3.3 Arming and Triggering

The Tevatron BPM front-end software must switch modes when arming itself for turn-by-turn and injection turn-by-turn measurements. The following tables (Table 1 and Table 2) define the events for arming and triggering BPM and BLM readouts.

DAQ Type	Arm	Readout Trigger
Fast Abort Closed Orbit	TCLK $71; Return from TBT	2 ms timer
Slow Abort Closed Orbit	TCLK $71; Return from TBT	1 second timer
Fast Time Plot (FTP)	TCLK $71; Return from TBT	User request
Profile	TCLK $71; Return from TBT	TCLK $75
Display	TCLK $71; Return from TBT	TCLK $78
Snapshot	TCLK $71; Return from TBT	User request
Turn By Turn	User request	Timing board interrupt
Injection Turn By Turn	V:BPJINE and TCLK $4D	Timing board interrupt
Injection Closed Orbit	TCLK $77; User request	Injection Turn By Turn complete

Table 1 - BPM arming and triggering

DAQ Type	Arm	Readout Trigger
Fast Abort	Initialization	1 second timer
Display	Initialization	TCLK $76
Fast Time Plot (FTP)	Initialization	User request

Table 2 - BLM arming and triggering

Table 1 does not show interactions between the timing board and the EchoTek boards. Additional arming and triggering using TVBS occur between the boards timing board and the EchoTek boards (see timing board document #???). The front-end software does not receive TVBS signal and does not trigger the EchoTek boards for closed orbit or turn-by-turn measurements (e.g. the timing board received the TVBS $77 signal which triggers a turn-by-turn measurement). The only interaction between front-end software and the EchoTek boards is during configuration and data readout from the EchoTek random access memory.

3.4 Tevatron BPM Data Buffers

The data input for the system come from the EchoTek boards and the BLM chassis. Basically they provide information about the beam position, intensity and loss. All those values, however, need to be taken at different times and hardware configurations. All data acquired in different modes and times must be kept in distinct buffers, making it accessible at any time by the online user.

These buffers are defined in the specifications document (section 2.3) and illustrated in the AD document #903. The following tables (Table 3 and Table 4) describe the buffers identified for the Tevatron BPM system.

Buffer	Type	Size	Readout Trig.	Source	Stops	Cleared
Fast Abort	Circular	1024	2ms timer	EchoTek	TCLK $47; TBT	never
Slow Abort	Circular	1024	1 second timer	Fast Abort	TCLK $47; TBT	never
Fast Time Plot	Circular		User request	Fast Abort	never	never
Profile Frame	FIFO	128	TCLK $75	Fast Abort	when full	TCLK $71
Display Frame	FIFO	1	TCLK $78	Fast Abort	never	never
Snapshot	FIFO	1	User request	Fast Abort	never	never
Turn By Turn	FIFO	8192	Timing board intr	EchoTek	end TBT	TBT Arm
Injection TBT	FIFO	8192	Timing board intr	EchoTek	end TBT	TBT Arm
Injection C.O.	FIFO	1	Inj. TBT complete	TBT Buffer	end TBT	TBT Arm

Table 3 - BPM Buffers

Buffer	Type	Size	Readout Trig.	Source	Stops	Cleared
Fast Abort	Circular	1024	1 second timer	BLM	never	never
Display Frame	FIFO	1	TCLK $76	Fast Abort	never	never
Fast Time Plot	Circular		User request	Fast Abort	never	never

Table 4 - BLM Buffers

3.5 Tevatron Metadata

Table 5 shows all metadata kept by the system. Following the table there is a description of the columns and the values they might contain.

Data	Valid Values	Source	Update at	Where	Output
Starting turn number	0 through N	Timing Board	Readout	Data Buffer	5 and 7
Total time within the cycle	>= 0	Timer	Readout	Calculated on Output	5, 6 and 7
Number of detectors	1 through 12 12 or 24	Internal Config	Readout	Main Metadata	5 and 6
Number of turns	1 through 8196	EchoTek, Timing Board User Request	Readout	Data Buffer	10
Endianess	0 for little endian else for non little endian	Internal Config	Static	Main Metadata	7
Header version	any	Internal Config	Static	Main Metadata	7
Overall status	0 is ok	Internal	Readout	Main Metadata	7
Detector status	0 is ok	EchoTek BLM	Turn	Data Buffer	3 and 4
Time stamp	>= 0	Timer	Readout, Channel Read	Data Buffer	7, 3 and 4
Data type	Flash/Fast Abort Slow Abort Profile Snapshot Display Turn By Turn Injection TBT Inection Closed Orbit	Data Buffer	Readout	Data Buffer	7
Trigger type	periodic TCLK		Turn	Main Metadata	7
Data source	Beam Calibration SW Diagnostics HW Diagnostics	Internal Config	Turn	Data Buffer	7
Particle type	Proton Pbar		Turn	Data Buffer	7
Bunch type	Coalesced Uncoalesced	V:COALP V:COALA	Turn	Data Buffer	7
Scaled data	Scaled Raw	User Request	Readout	Calculated on Output	7
Machine state	1 - 24	V:CLDRST	Turn	Main Metadata	13
Helix state		V:HELIX	Turn	Data Buffer	13
Proton bunches		V:PBKTC	Turn	Data Buffer	13
Pbar bunches		V:ABKTC	Turn	Data Buffer	13
EchoTek config	Turn By Turn Closed Orbit Calibration Diagnostic	Internal Config	Mode Change	Data Buffer	
EchoTek status	0 is ok	EchoTek	Board Read	Data Buffer	5 and 10
BPM state	To be defined	Internal state	To be defined	Main Metadata	13

Table 5 - System metadata

Source: defines where data comes from

EchoTek: data is retrieved from the EchoTek board(s)
BLM: data is retrieved from the BLM chassis
Timing Board: data retrieved from the Timing Board
Timer: internal timer that is reset on a TCLK to be defined
User Request: data comes with the ACNET request
State Device: data is updated from a state device change (e.g. V:CLDRST)
Internal Config: data is kept internally by the syste
Data Buffer: data is within a data buffer (e.g. BPM Fast Abort Buffer)

Update at: defines when data is updated

Turn: data is updated every turn or every time EchoTeks and BLMs are read
Readout: data is update/calculated when processing online request
Board Read: data is updated when an EchoTek board is read
Mode Change: data is updated when there is a mode change
Channel Read: data is updated when reading channel (equivalent to Board Read)
Static: data does not change during normal operation

Where: defines where the data is kept internally

Data Buffer: data kept within the data buffer and is unique for every entry (turn)
Calculated On Output: data is calculated at online readout request
Main Metadata: data is kept by the system main metadata structure

Output: define what returning structures contain the metadata (doc #860 section 4.2 – Data Structures (Output Data)):

ID	Structure	Contains
0	BPM_TIME	-
1	TRIGGER_INFO	-
2	TEVATRON_BPM_TBT_TURN	-
3	TEVATRON_BPM_TIME_SLICE_VALUE	-
4	TEVATRON_BLM_TIME_SLICE_VALUE	-
5	TEVATRON_BPM_FRAME_DATA	0 and 13
6	TEVATRON_BLM_FRAME_DATA	0
7	TEVATRON_BPM_HEADER	0 and 1
8	TEVATRON_BLM_DATA	6 and 7
9	TEVATRON_BPM_ORBIT_DATA	5 and 7
10	TEVATRON_BPM_TBT_DATA	2, 7 and 13
11	TEVATRON_BPM_TIME_SLICE_DATA	3, 7 and 13
12	TEVATRON_BLM_TIME_SLICE_DATA	4, 7 and 13
13	TEVATRON_BPM_STATE_DATA	-

Table 6 - Output data structures

3.6 Tevatron BPM State Diagram

The possible states for the front-end system are displayed in Figure 4. The upper part of the picture depicts the states assumed by the control task, while the lower section contains state diagrams for the data acquisition tasks. Additionally, other states shown in the middle section define states and transitions traversed by the MOOC task running BPM code through its callbacks.

The system's default mode of operation is *ClosedOrbit*, and it gets to that default state during initialization, passing through the states *ChangeEchoTekMode* and *ChangeTimingMode*. Depending on events received while at *ClosedOrbit*, the system enters other modes, such as *Diagnostics*, *Calibration*, *TurnByTurn* and *Alarm*.

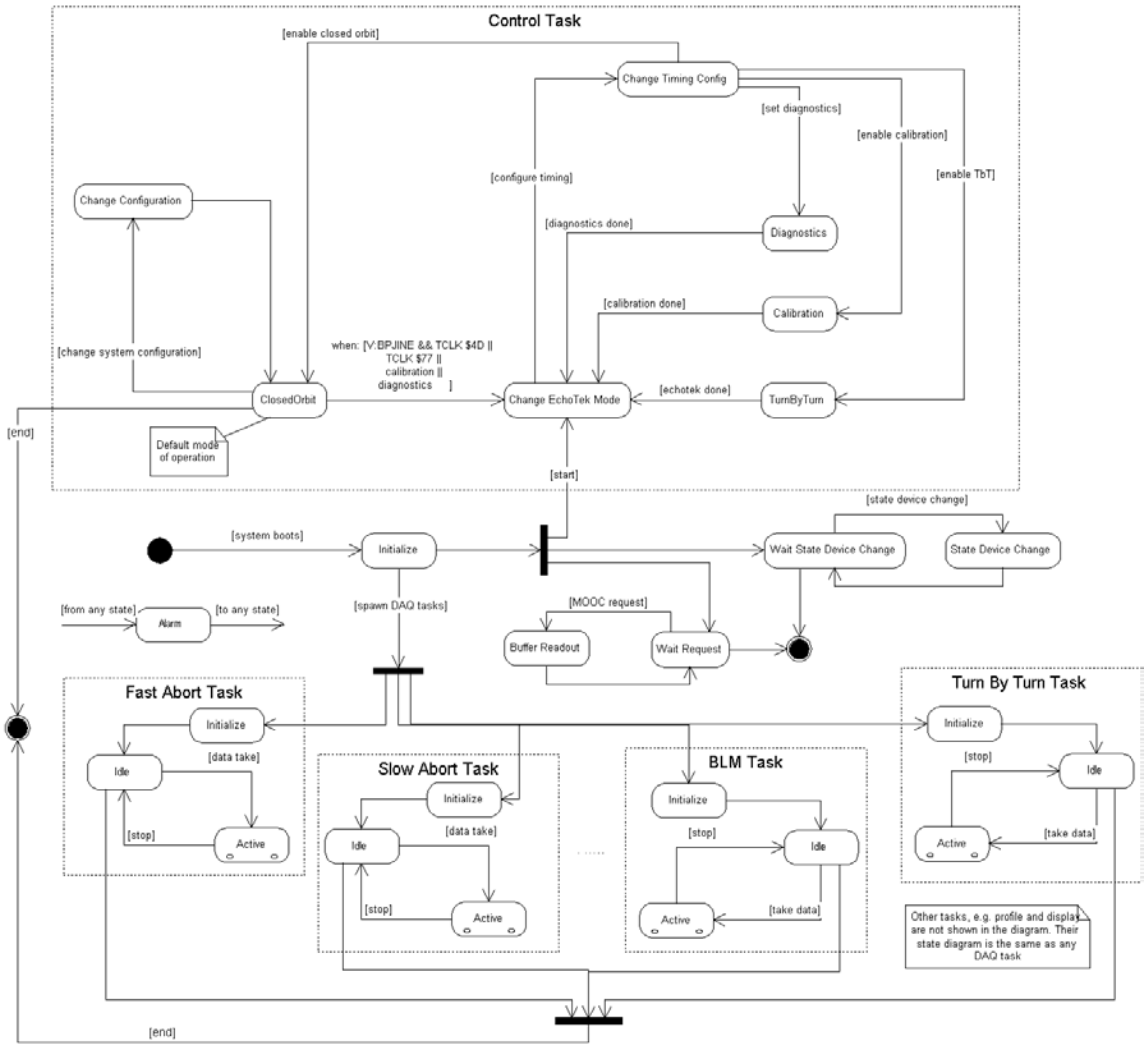

Figure 4 - Tevatron BPM state diagram

Figure 4 shows only four data acquisition tasks. However the system contains more tasks. Their state diagrams are similar, therefore not repeated in the picture. Notice that the states for the data acquisition tasks are the same. The difference between the tasks is located in the internal state diagram defined for the state *active*. Figure 5 describes the internal states for an active data acquisition task. The difference tasks is in the type of event it waits in order to perform data acquisition. For example, the turn-by-turn task waits on an interrupt from the timing board while the BPM profile task waits for the TCLK $75.

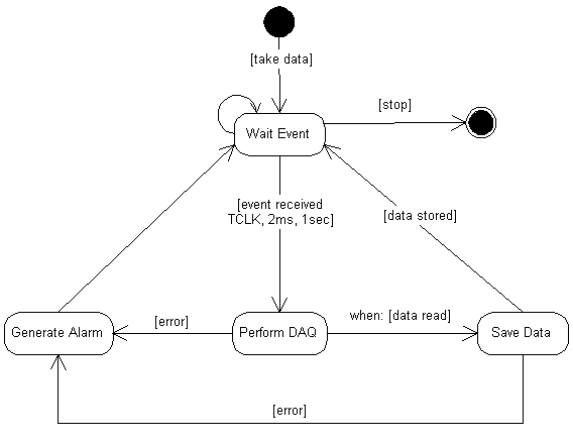

Figure 5 - Data acquisition task state diagram

3.7 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 are specific to the Tevatron BPM system. Remaining classes are part of the generic framework. Class names throughout the text are written in *italic*.

3.7.1 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 6 contains the task classes in the system. The upper class represents the wrapper.

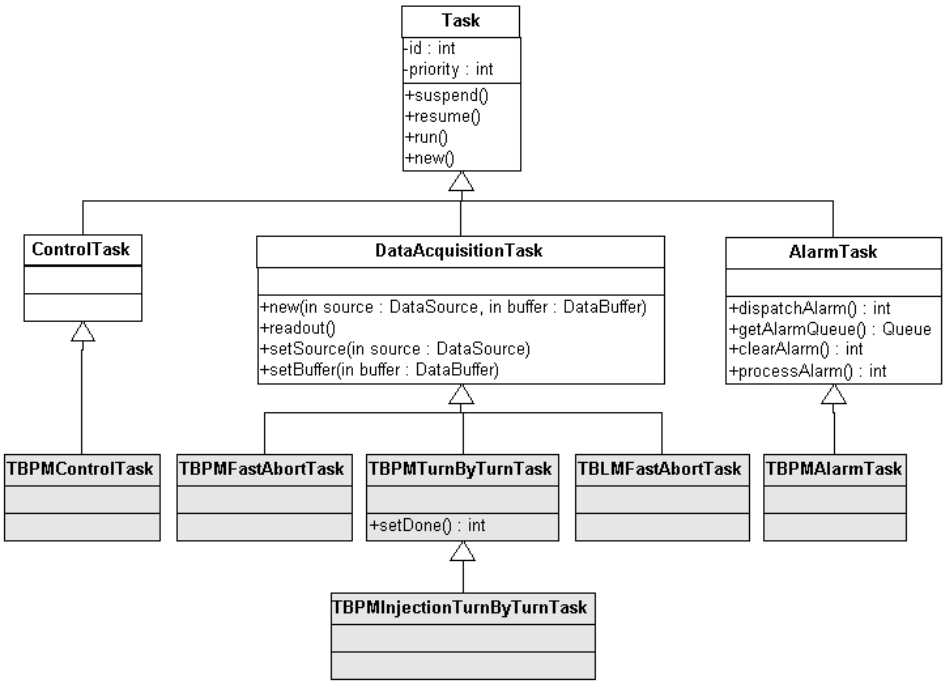

Figure 6 - Class diagram for tasks in the system

The system is overseen by a *ControlTask*, which is responsible for initializing most of the system, configure hardware (EchoTek boards, BLMs, timing module, calibration and diagnostics hardware), 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*.

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.

3.7.2 Controls

The main class in the system is *Control*. It is however contained 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 7.

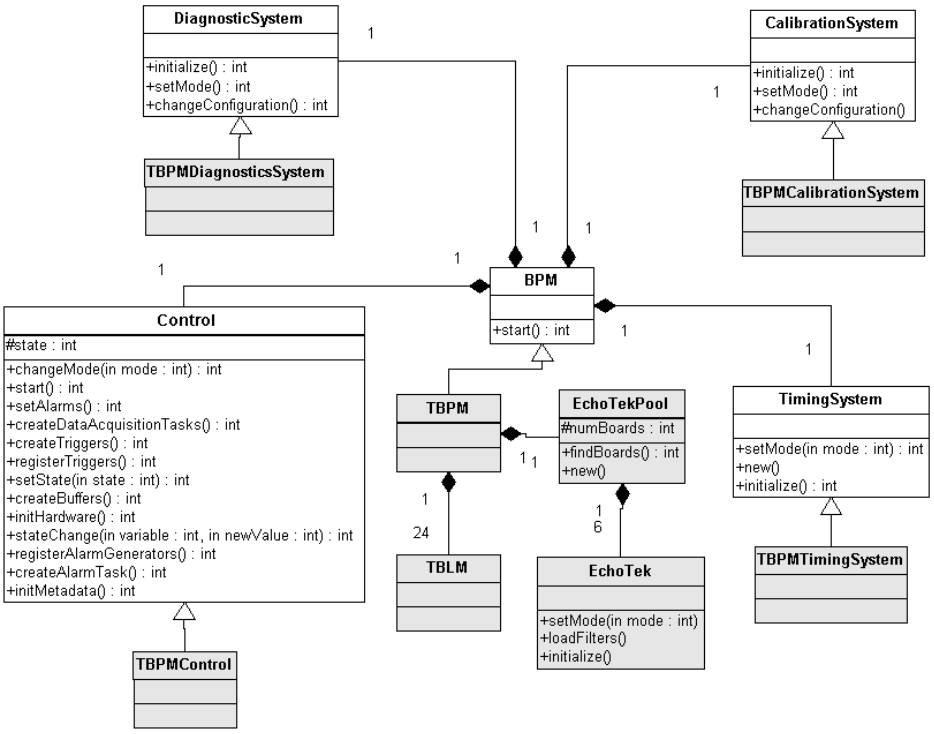

Figure 7 - Main control classes

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.

3.7.3 Events

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

Events are classified by their field *type*. Based on the type of the event a specific action is taken. For example, when a TCLK $75 is received by the interrupt handler, it puts in the queue of the BPM profile task an event structure whose type indicates that a TCLK $75 has occured.

Similarly other types of events can be defined by using distinct values for the type field. It can define distinct types of alarms for instance. Additionally, the structure contains a void pointer (called payload), which can be used for passing additional information within the event. Taking the same example, the payload can have additional information about the alarm being generated.

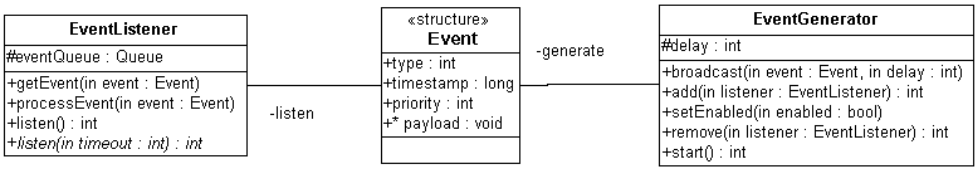

Figure 8 – Handling events in the system

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 *processEvent ()*.

3.7.4 Event Listeners and Generators

Events can be generated and received by any entity in the system. Figure 9 shows the classes that currently generate events, while Figure 10 shows classes able to receive events. *EventListeners* are:

- *ControlTask*: receives requests and events.

EventGenerators are:

- *StateChangeEventGenerator*: generate an event signaling a state device change;
- *InterruptEventGenerator*: generic event generator based on interrupts;
    - *TCLKGenerator*: generate TCLK event on interrupts;
    - *TimeEventGenerator*: generate an event on every tick of a timer.
- *Control*: generates alarms for the alarm task and events for data acquisition tasks.

EventListeners **and** *EventGenerators*:

- *AlarmTask*: receives alarm events from other tasks in the system; and generates events sent to the *ControlTask* to inform about the current alarm situation;
- *DataAcquisitionTask*: receive events signaling the data acquisition process; may generate events to other *DataAcquisitionTasks*.

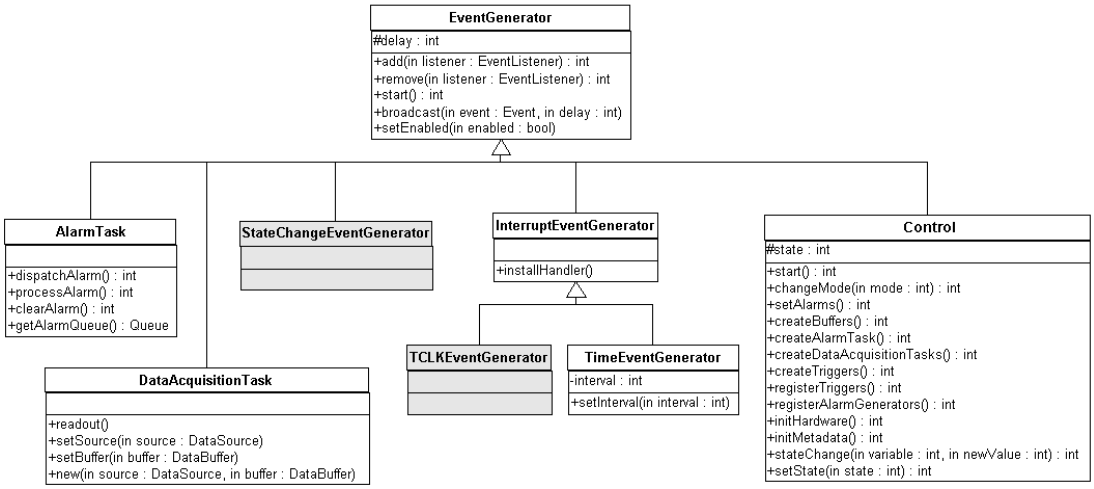

Figure 9 - Event generators

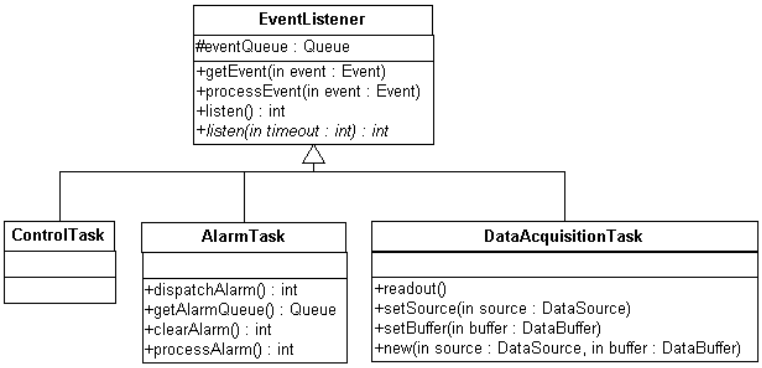

Figure 10 - Event listeners

3.7.5 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 11). 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.

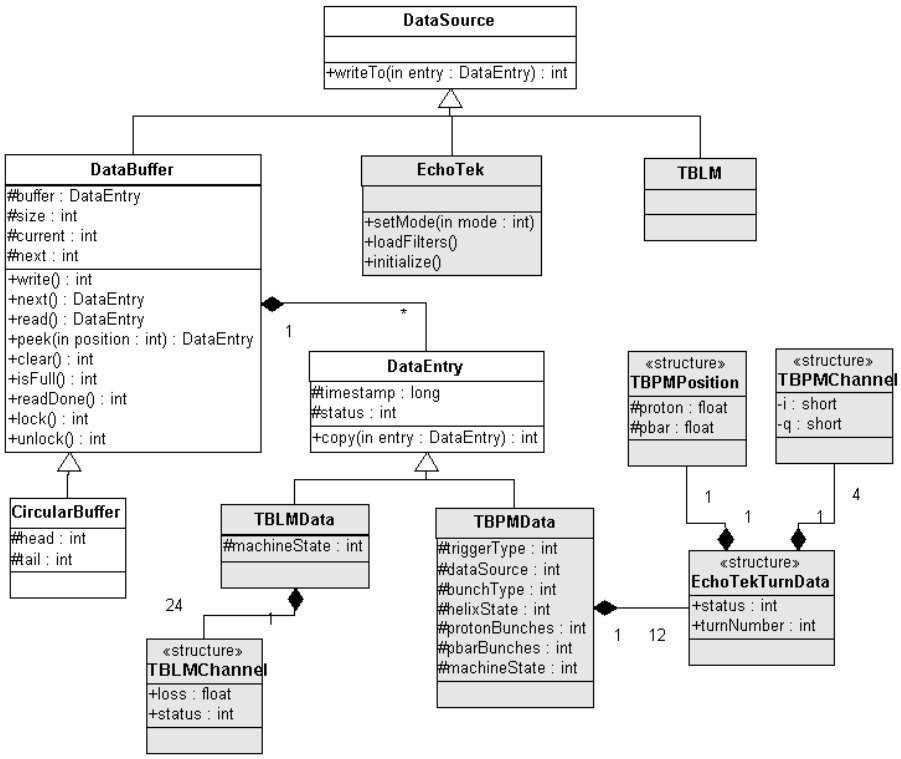

Figure 11 - Reading and saving data

The *DataBuffers* have data stored in a format that may be different from the format sent to the end user through MOOC by invoking the *BufferReadout* class (Figure 12). Data is formated according to a *Packer* (Figure 12). Depending on the data type and on the user request a specific *Packer* is used (e.g. *TBPMClosedOrbitPacker* and *TBPMTurnByTurnPacker*).

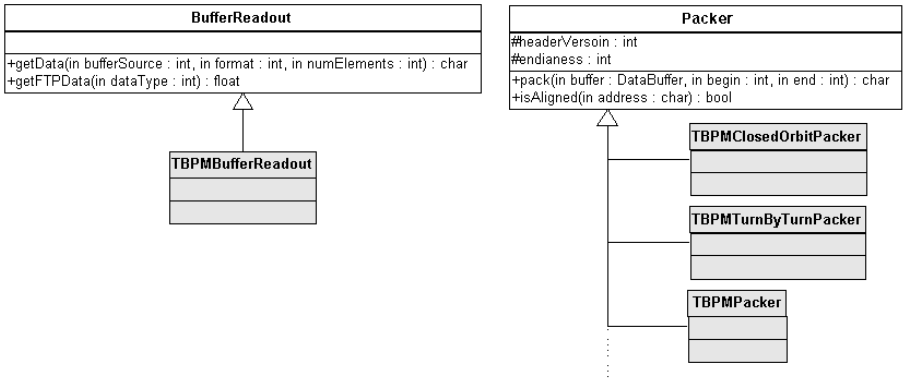

Figure 12 - Buffer readout related classes

3.7.6 Alarms

The classes responsible for handling and generating alarms are shown on Figure 13. An alarm event is generated by an *AlarmGenerator*. The generators in the system are the following: *DataAcquisitionTask* and *Control*.

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

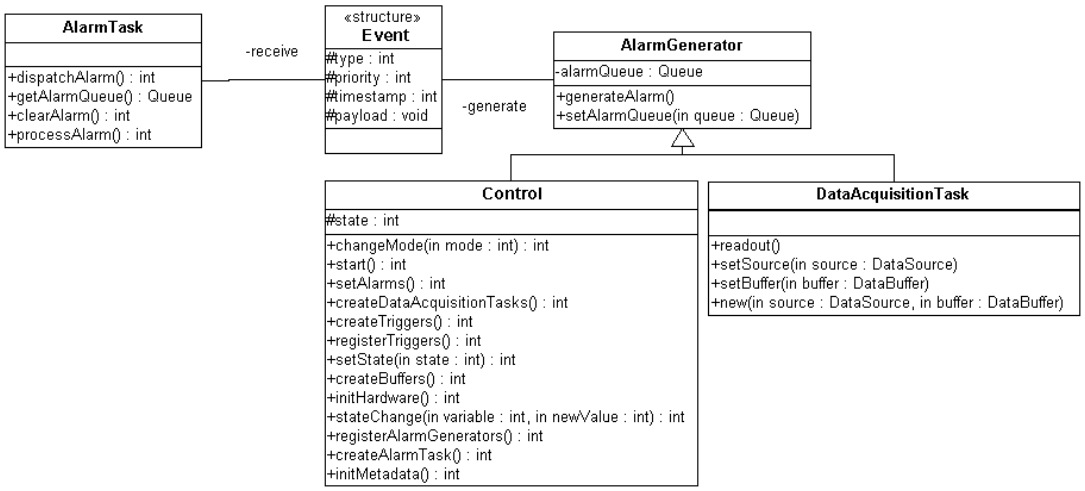

Figure 13 - Alarm classes

3.8 Timing Diagrams

This section contains timing diagrams illustrating the behavior of the several components of the system during data acquisition operation. The default mode of operation of the BPM front-end system is the closed orbit mode. In this mode, the EchoTek boards are readout on every 2 milliseconds. All EchoTek boards present in the system are expected to be read out during this 2 milliseconds interval. Figure 14 shows the components involved in the closed orbit mode, and at what time they are expected to be run or be accessed. The 2 millisecond interrupt enables the BPM fast abort task to take a measurement, which accesses the EchoTek cards by reading their random access memory at the location where closed orbit measurements are stored. While reading the boards, the task also needs to access the fast abort buffer, where the data will be stored in the front-end side. The amount of time to access data from EchoTek boards is expected to take most of the processing time of the BPM fast abort task. The task will also have to acquire the lock for the fast abort buffer, in order to avoid access to the data while it is being written. This lock must be configured to allow priority inversion, such that if a user is accessing the buffer, the processor will bump its priority in order to allow the BPM fast abort task to gain the buffer control.

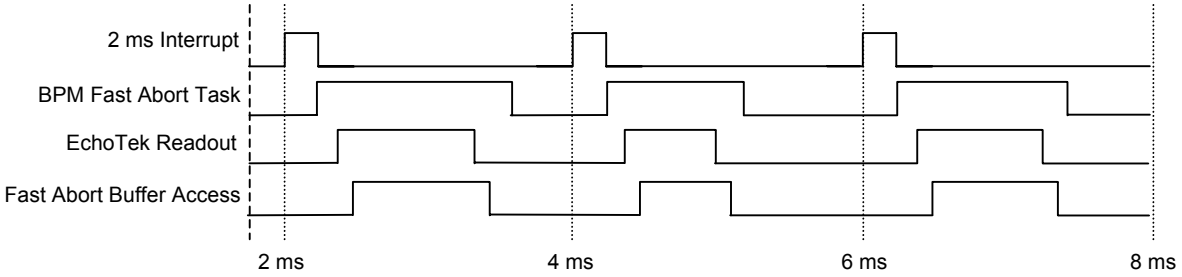

Figure 14 - Timing diagram for the BPM fast abort DAQ

The system contains other data acquisition tasks running besides the BPM fast abort task. Figure 15 shows the situation where other data acquisition tasks run concurrently to the fast abort task. In addition, it is considered that the 2 millisecond interrupt generates the event at 1 second which enables the BPM slow abort task. After the fast abort task is done reading out the EchoTek boards the BLM fast abort task is allowed to run. In its turn it reads out the BLM chassis and save data to the BLM fast abort buffer. The diagram shows the BLM fast abort task being interrupted by another cycle of the BPM fast abort task. The BLM fast abort task will resume the readout after the BPM fast abort has finished its new cycle. Still before a new 2 millisecond event the processor is able to run other data acquisition tasks. The picture shows the BPM slow abort task being scheduled, and it accesses the BPM fast abort buffer and the BPM slow abort buffers. Similarly, other tasks such as the BPM display or BLM display. Those tasks would use the remaining time between the BPM fast abort readout cycles.

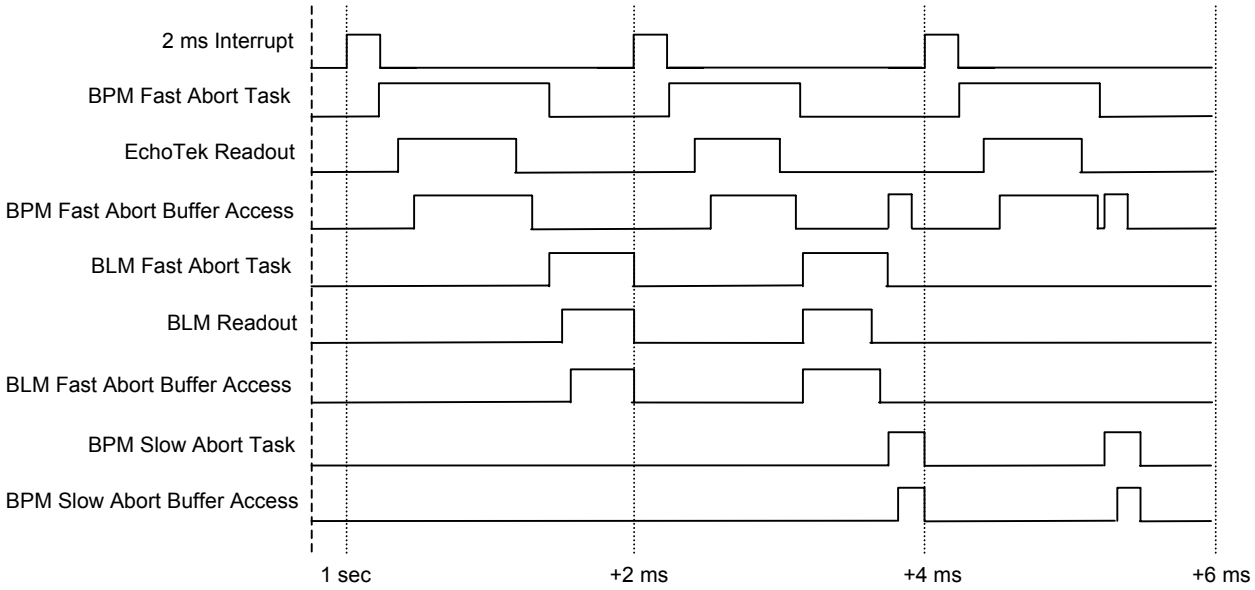

Figure 15 - Timing diagram for BPM fast and slow abort DAQ and BLM fast abort DAQ

Figure 16 describes the process of making a turn-by-turn measurement. For entering the turn-by-turn mode, the system must receive TCLK $77, which is received by the control task. The control task is responsible for disabling the closed orbit mode and for arming the turn-by-turn task. While in turn-by-turn mode, tasks that read closed orbit measurements (either from the EchoTek boards or from the BPM fast abort buffer) must be stopped. The BLM fast abort task can still be allowed to run in turn-by-turn mode because it does not depend on the EchoTek boards.

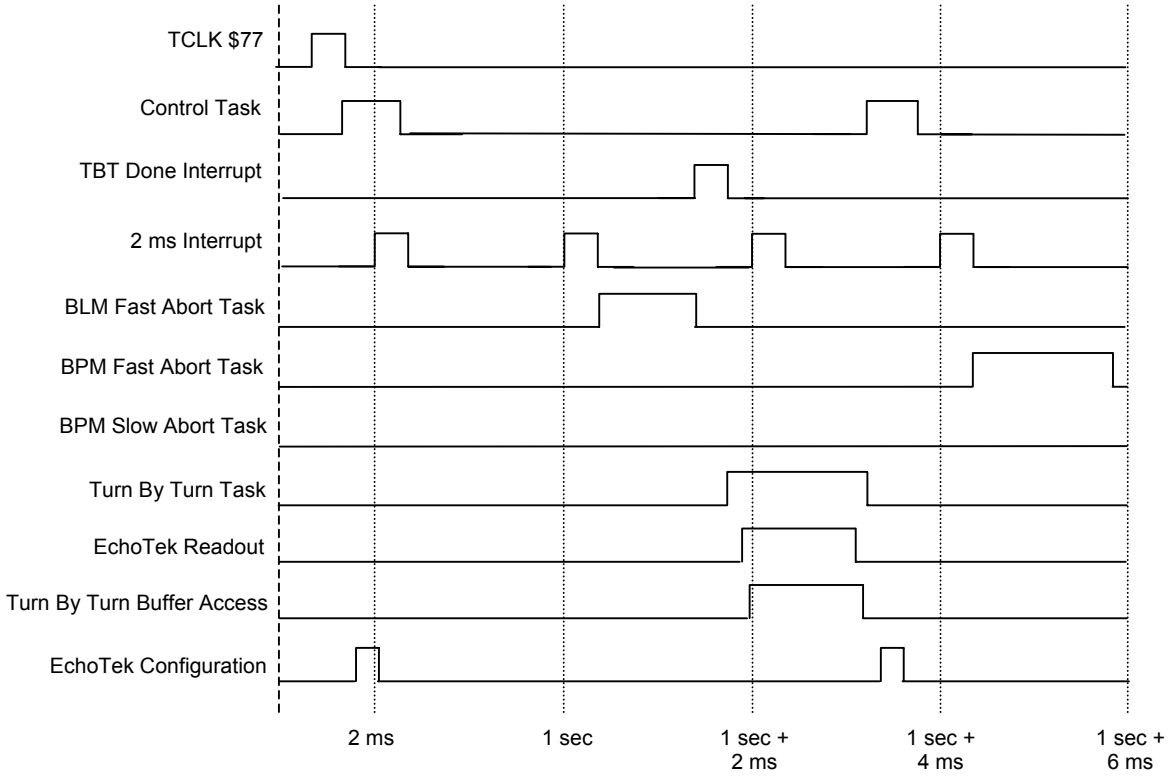

Figure 16 - Timing diagram for a turn-by-turn measurement

After receiving the turn-by-turn done interrupt from the timing board, the turn-by-turn task performs the EchoTeks readout and saves data into the turn-by-turn buffer. The control task is signaled that the measurement is complete, allowing it to reconfigure the EchoTek boards and resume closed orbit related tasks.

The order of execution of data acquisition tasks is dictated by the priority it has. Table 7 shows the expected priorities of the different tasks in the BPM system. It is considered

that smaller numbers represent higher priorities, i.e. the control task has the highest priority while the BLM display task has the lower.

Task	Priority
Control task	0
Turn by turn task	1
Injection turn by turn task	1
BPM fast abort task	2
Alarm task	3
BLM fast abort task	4
BPM slow abort task	5
BPM profile task	6
BPM display task	6
BLM display task	6

Table 7- Task priorities

3.9 Activity Diagrams

This section contains diagrams showing the work flow of different tasks in the system. Figure 17 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 and other events generated within the system.

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. As an alternative for simple requests, the MOOC callback invokes directly the code for processing the request.

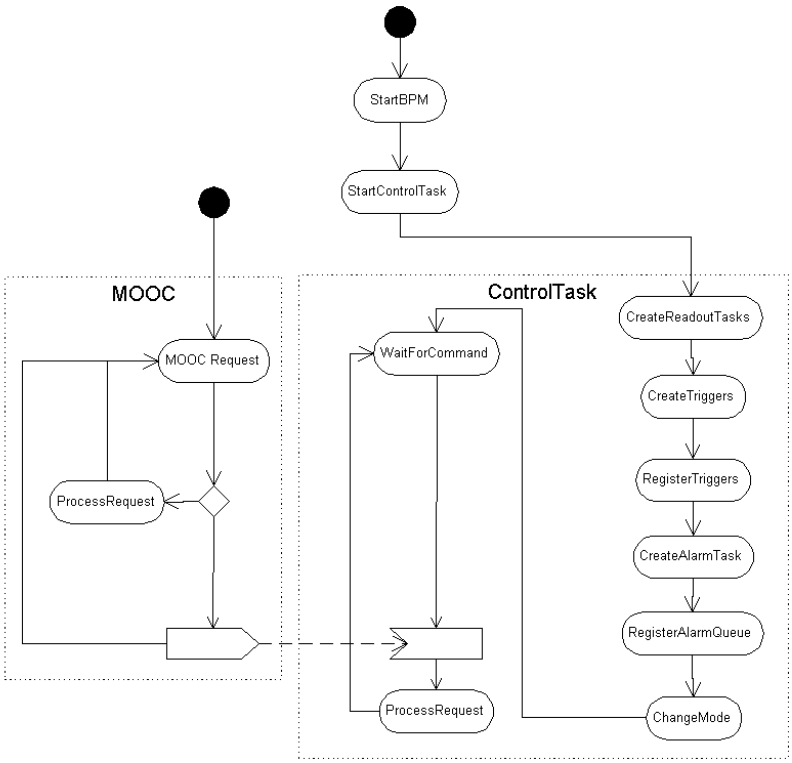

Figure 17 - *ControlTask* flow

Similarly, the *DataAcquisitionTasks* run in a closed loop waiting for *Events*. Upon the reception of an *Event*, 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 18 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 *Events* 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 19. A request comming from MOOC enters the callback, which in turn calls methods from the *BufferReadout* class. The class provide methods for retrieving the data from the internal buffers and a *Packer* is used to format the data according to the online applications (doc #860).

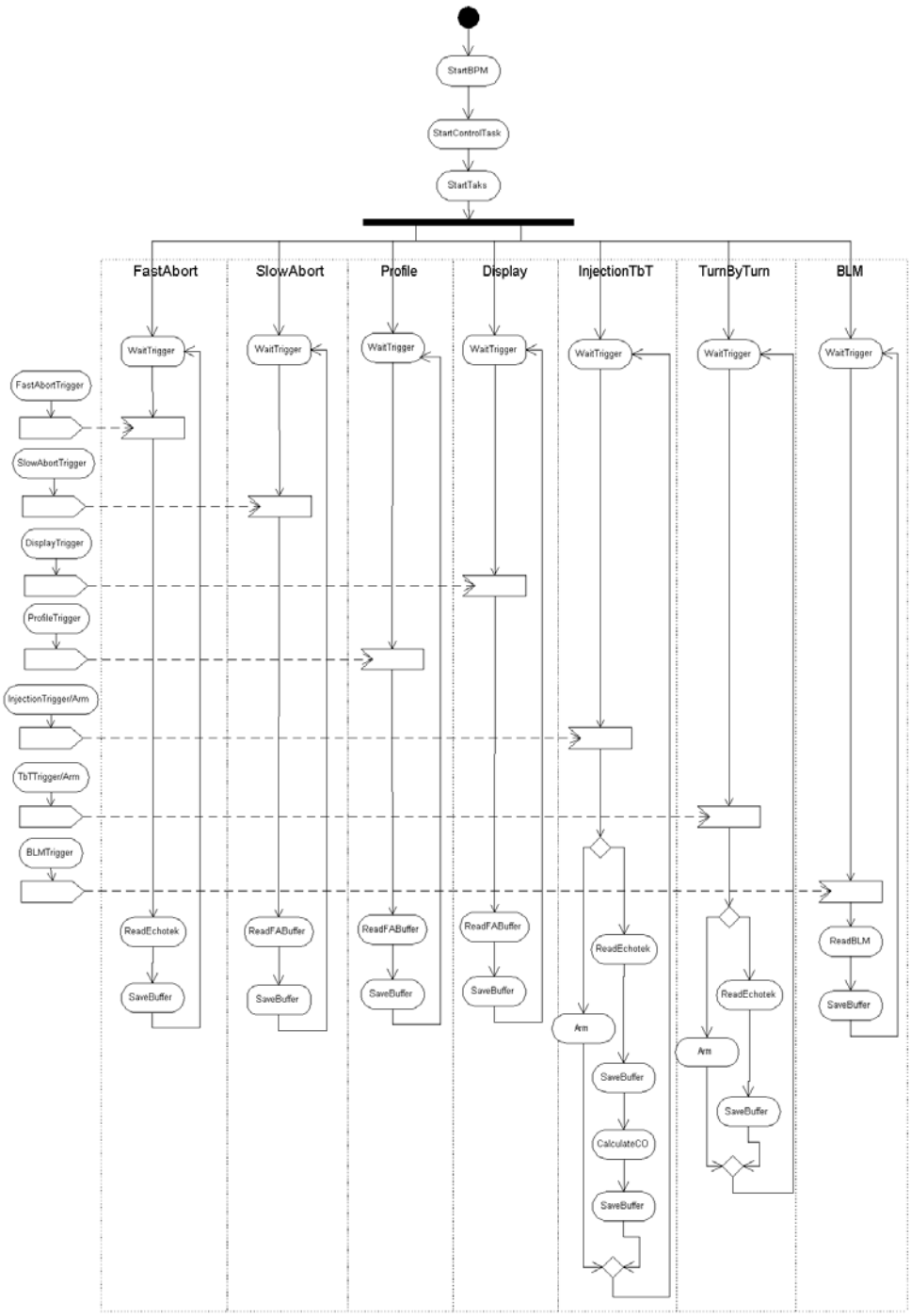

Figure 18 - *DataAcquisitionTasks* flow[2]

[2] 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).

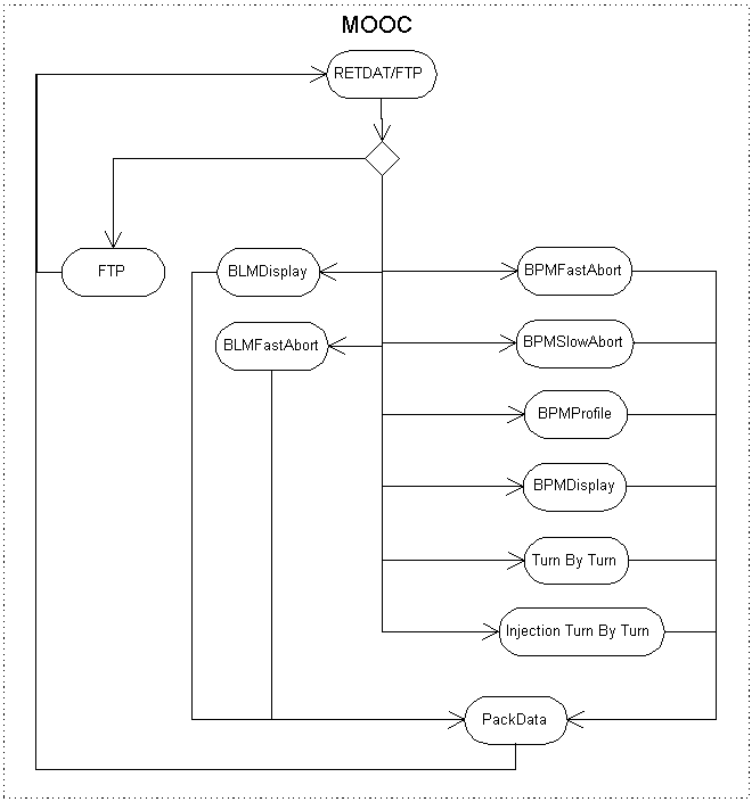

Figure 19 - *BufferReadoutTask* flow

3.10 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 and returns.

3.10.1 Initialization

Figure 20 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* delegates the actual work to the *Control* class, which is responsible for creating most of the objects within the system. It must instantiate the *DataAcquisitionTasks*, create the *AlarmTask*, *EventGenerators* among other initialization procedures.

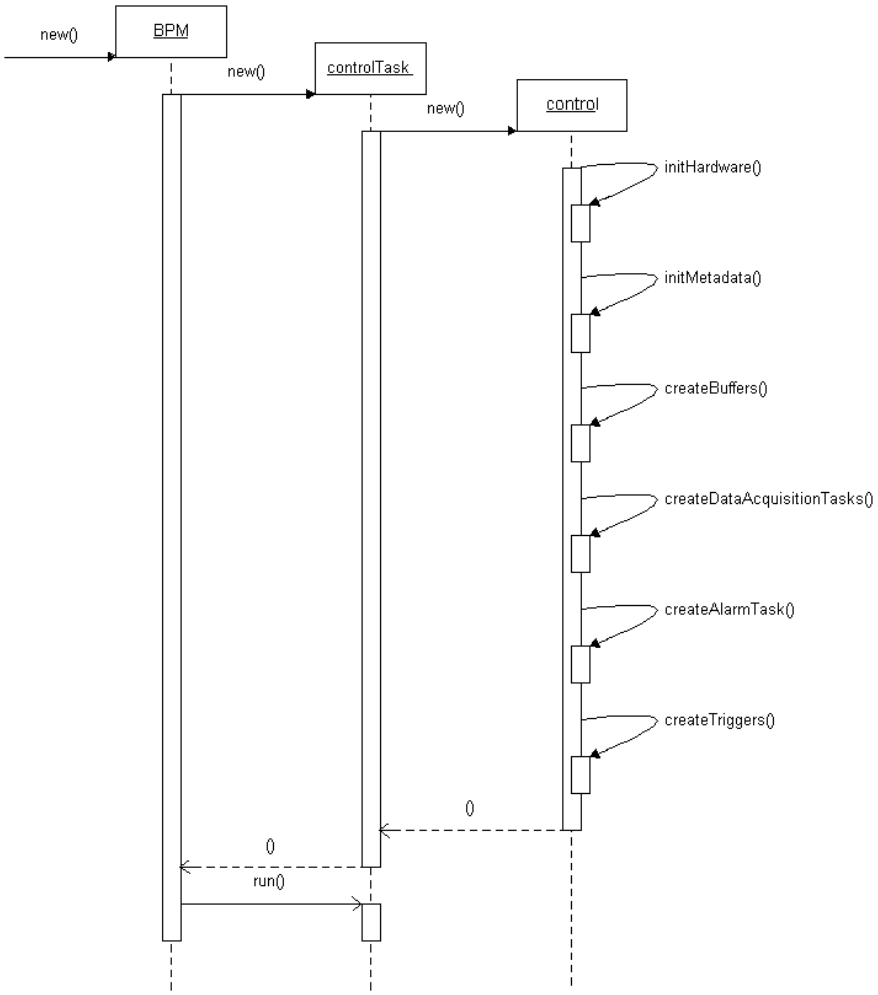

Figure 20 – Initialization sequence

The following pictures describe in more details every step the *Control* class goes through when being initialized. Its first task is to configure the hardware system. This is depicted in Figure 21. The EchoTek boards must be found and initialized (crates have different number of EchoTek boards), as well as the timing, calibration and diagnostics hardware.

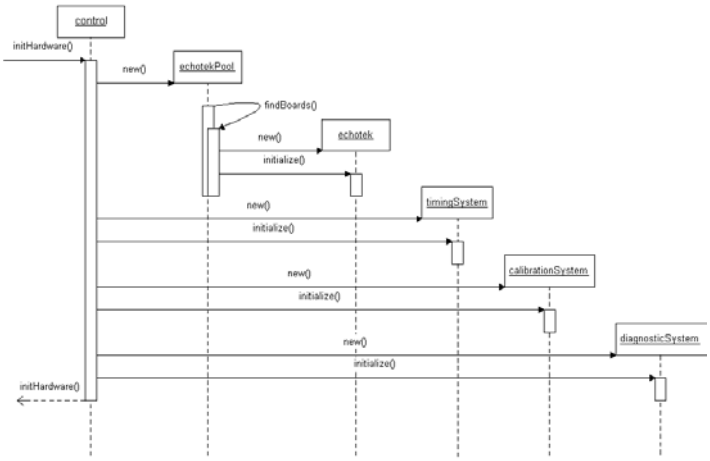

Figure 21 - Hardware initialization sequence

Following the hardware initialization, the system must get information about the initial state of its metadata (Figure 22). Most metadata is controlled by the system itself, such as the state and state of the EchoTek cards. Remaining metadata comes from outside the system and must be retrieved at initialization. An example of external metadata is the current Tevatron state, given by the ACNET variable V:CLDRST.

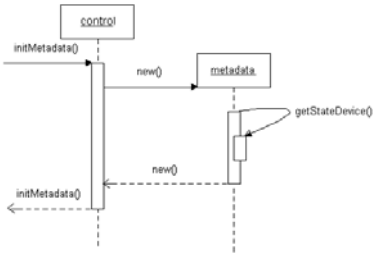

Figure 22 - Metadata initialization sequence

Figure 23 shows the creation of the several data buffers in the system. They don't need to be created at a particular order, but is at this point that their size is defined. During its creation the buffer allocates the memory that will hold the data read from the EchoTeks and the BLMs.

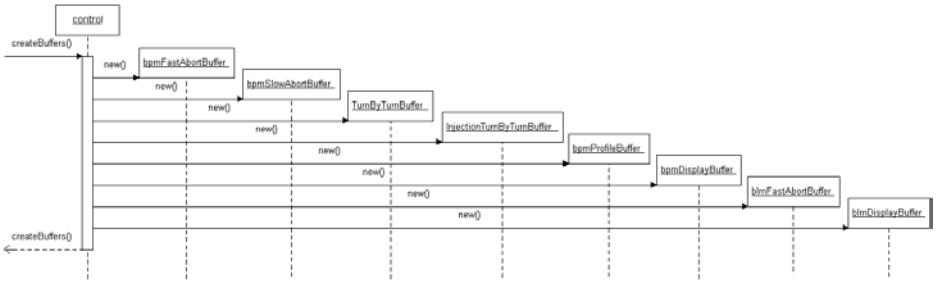

Figure 23 - Buffer initialization sequence

After creating the buffers, the data acquisition tasks are created (Figure 24). The tasks receive the data source and data buffers that will be used.

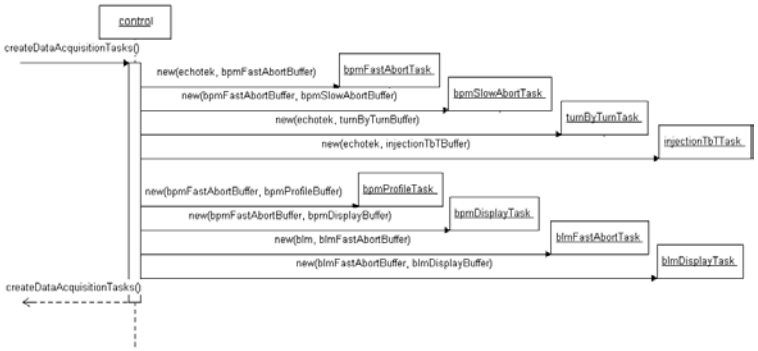

Figure 24 - Data acquisition tasks initialization sequence

Next step is the initialization the alarm scheme by instantiating the alarm task and informing other tasks in the system about the alarm queue. Alarms generated in the system are posted to the alarm queue, which is constantly read by the alarm task.

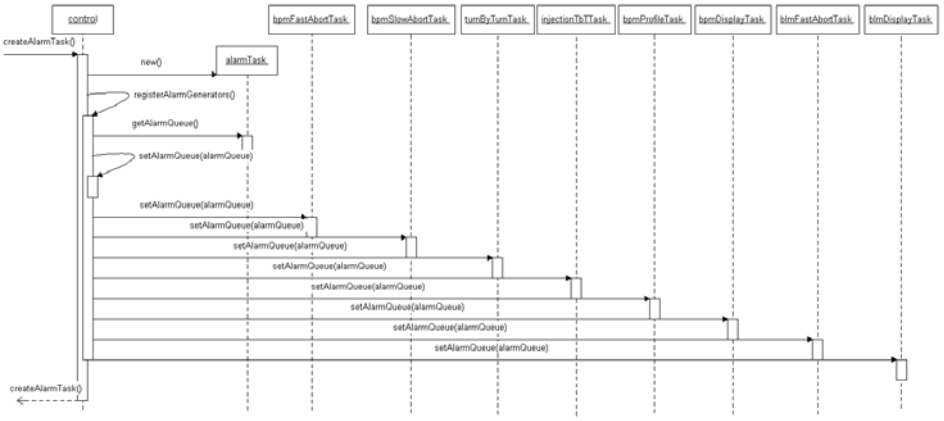

Figure 25 - Alarm initialization sequence

Following the alarm initialization, the event generators are created. They are basically interrupt handlers that will create events when called. The events are then placed into the input queues of the listeners. Figure 26 also shows what are the listeners for each event generator.

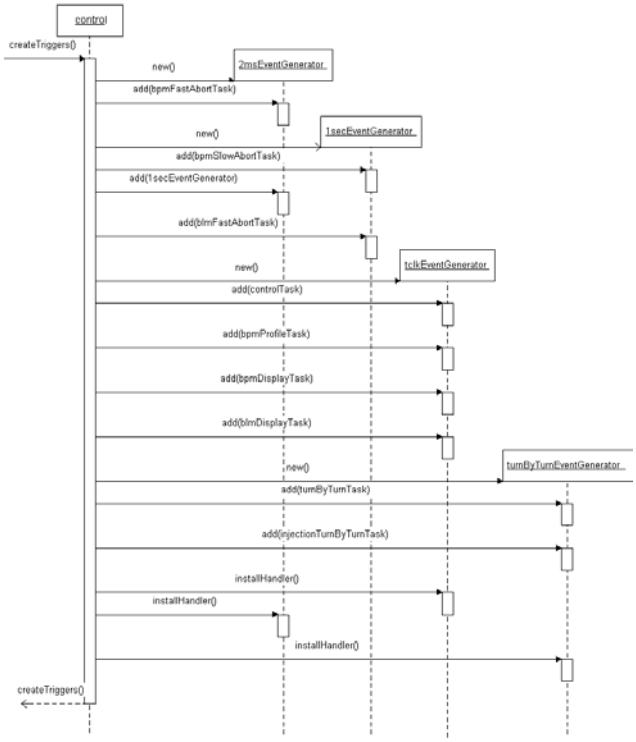

Figure 26 - Event generators initialization sequence

At last in the initialization sequence, the tasks are allowed to start and the system enters in the normal mode of operation, which is enabled through *changeMode ()* (Figure 27).

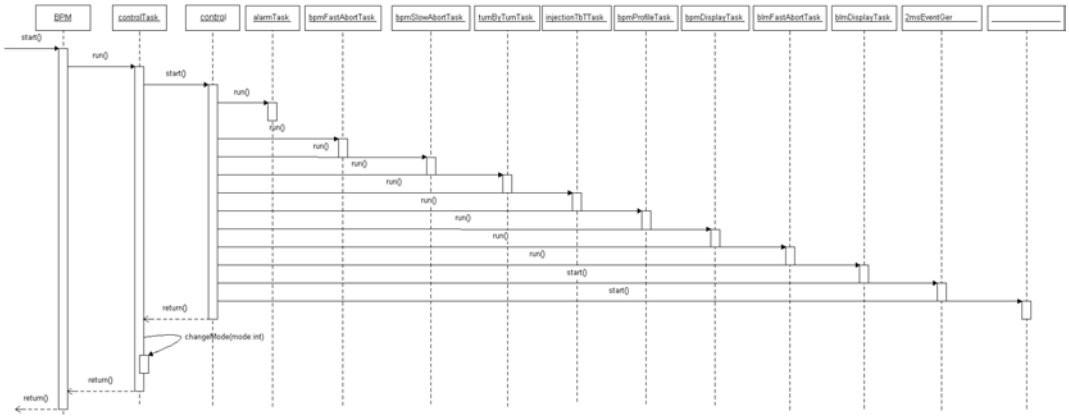

Figure 27 - Tasks initialization

3.10.2 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[3]. The closed orbit mode is the default mode of operation. Turn-by-turn mode is enabled on user request or on a programmed TCLK event. Figure 28 shows the sequence of operations when changing from the default mode to the turn-by-turn mode.

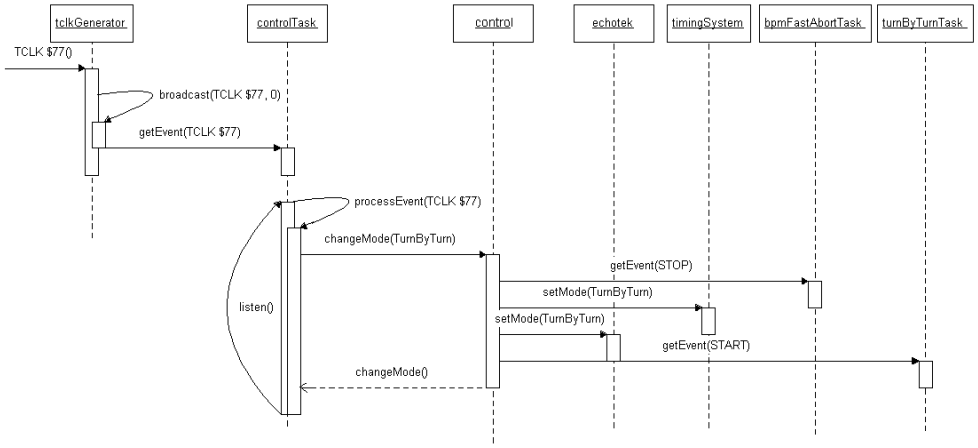

Figure 28 - Changing modes

In the particular case depicted on Figure 28, the mode change is triggered by the TCLK $77. The event will be passed down to the *controlTask* which calls *control*. In the *changeMode* method the mode of operation of the EchoTek boards are changed (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 events via their input trigger queue. When a *DataAcquisitionTask* receives a STOP command it will ignore any events from that moment on. Upon the reception of a START command, the *DataAcquisitionTask* starts processing events again.

Upon the completion of a turn-by-turn measurement, the system must return to its default mode closed orbit. The sequence illustrated in Figure 29 shows the steps taken by the system when returning to its default mode of operation.

[3] 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

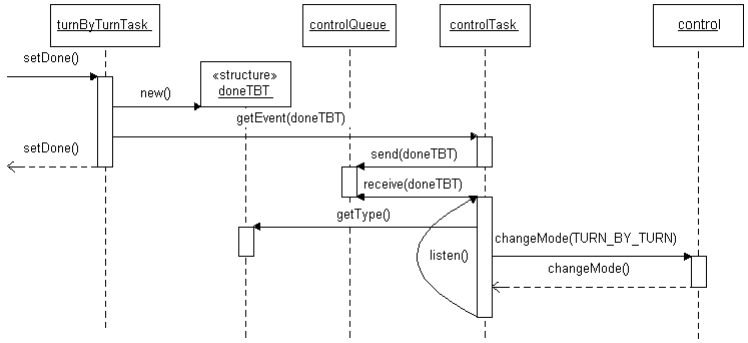

Figure 29 - Return to close orbit mode

3.10.3 Buffer Readout

Buffer readout operations follow the sequence defined in Figure 30. A user request comes through the MOOC framework, which invokes the *BufferReadout* class. The bufferReadout selects the data buffer according to the request specification and calls a *Packer* for arranging the data in the format expected by the online user.

The *Packer* object is able to compute the position and intensity of the beam for the data entries in the buffer, if configured to do so. Another option for calculating position and intensity is at readout time (see section 3.10.6). The calculation uses the selected calibration constants and selected algorithm to produce the position and intensity data.

For different types of user request a different packer can be used (e.g. closed orbit or turn-by-turn).

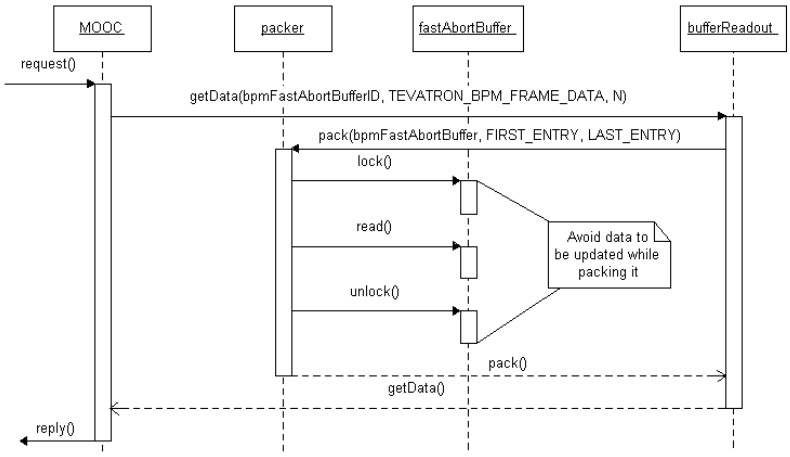

Figure 30 - Fast abort buffer readout

3.10.4 Alarms

All tasks in the system are capable of generating alarms. Figure 31 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.

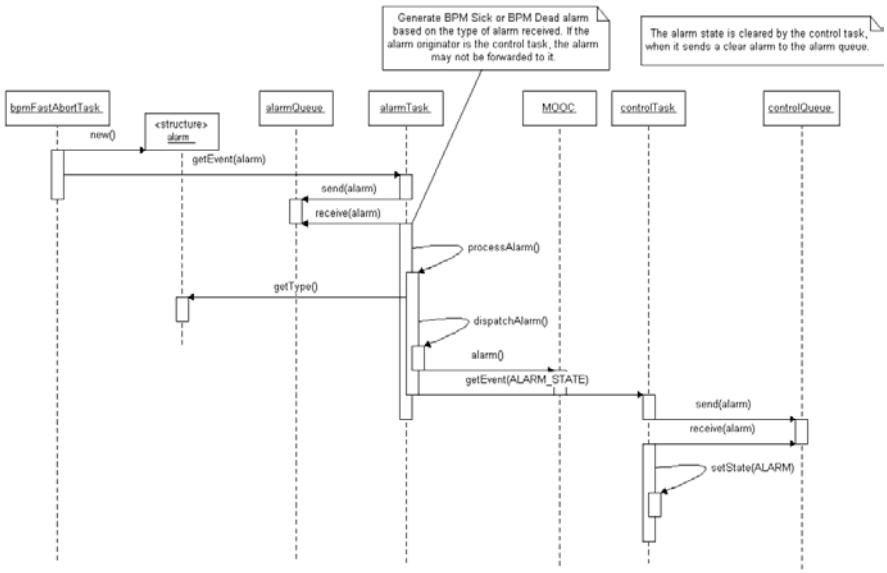

Figure 31 - Alarm generation

The alarm state can be cleared by the *ControlTask* or by the online user through sending a clear message to the *AlarmTask* (see Figure 32).

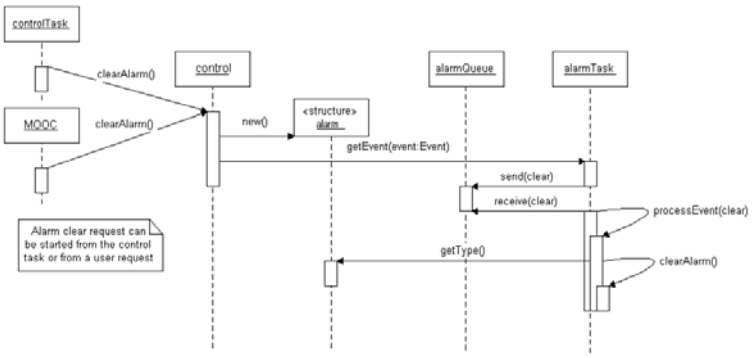

Figure 32 - Clearing an alarm

3.10.5 Events

Figure 33 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 *processEvent ()* method.

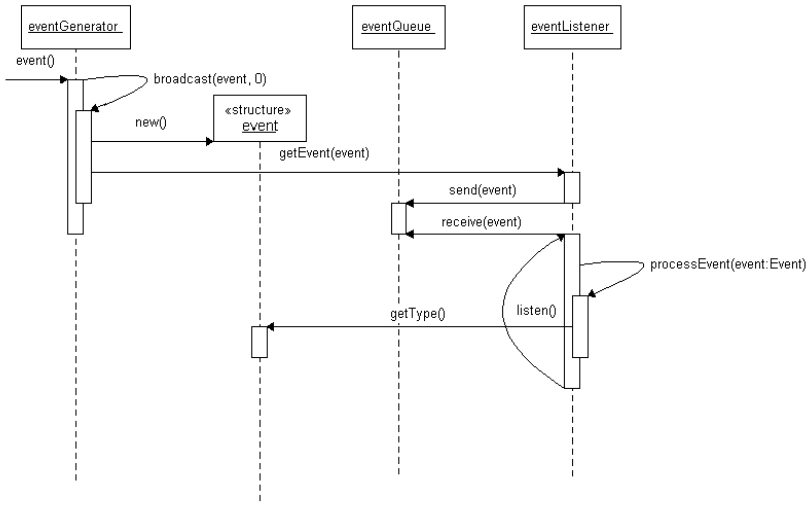

Figure 33 - Event generation

A particular case of event handling is shown in Figure 34, where the generator is of the type *StateChangeEventGenerator*, and the receiving side is the *ControlTask.*

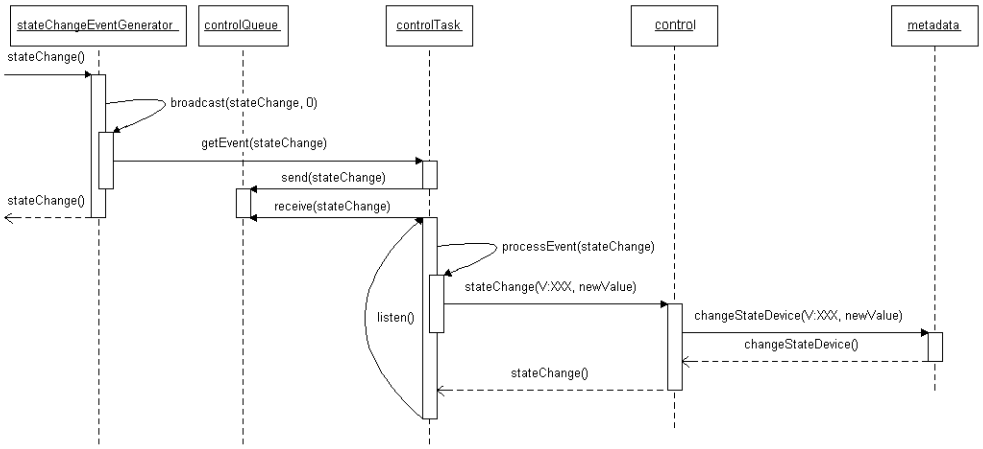

Figure 34 - State device change

3.10.6 Data Acquisition

The data acquisition process is similar to the event handling scheme on Figure 33. The event in question is a 2 millisecond event, generated by the *2msEventGenerator*. The event is sent to the *bpmFastAbortTask*, which in turn will acquire data from the *echoTek* and save it to the *fastAbortBuffer*. This process is illustrated in Figure 35. After the data is read from the *echoTek* object the I and Q values can be used to calculate the beam position and intensity. It will use a selected algorithm and calibration set defined for the current running state of the machine. The particle position and intensity can also be calculated when data is being read out from the front-end card.

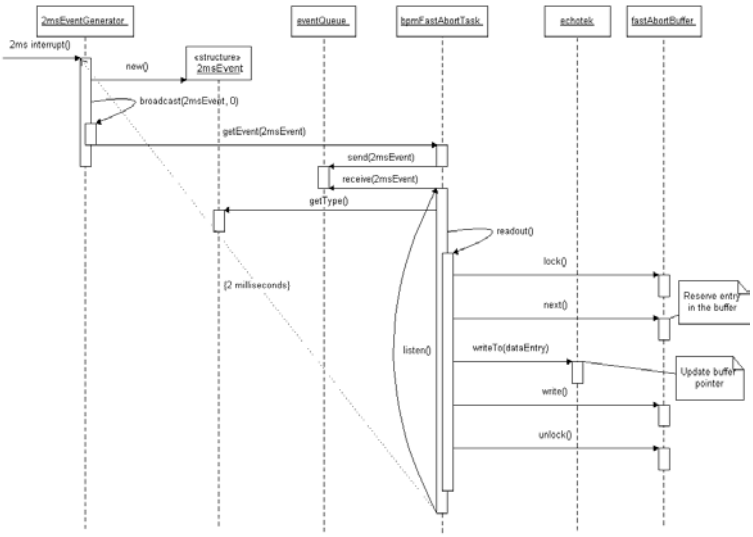

Figure 35 - Fast abort trigger generation

A similar diagram in Figure 36 is provided for the turn-by-turn case, where the *turnByTurnEventGenerator* receives the interrupt from the timing board signaling data ready, which is passed down to the *turnByTurnTask*, which carries out the data acquisition.

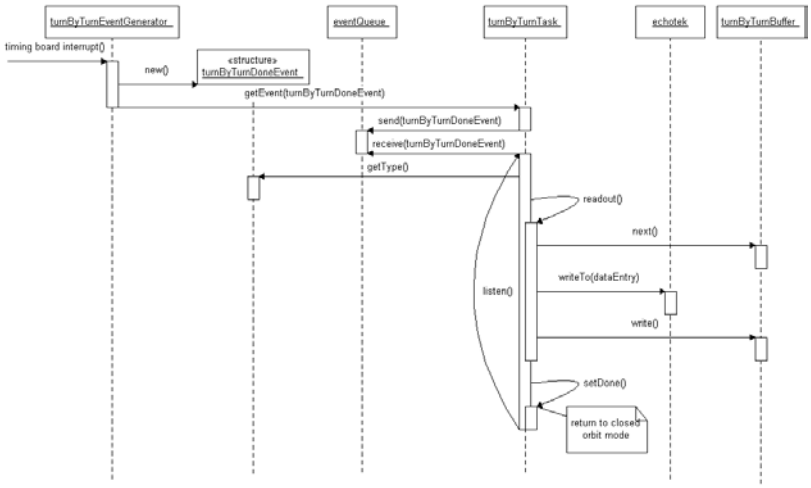

Figure 36 - Turn by turn data acquisition

3.11 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.

3.11.1 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
  - DataAcquisitionTask
  - AlarmTask
- Control
- Queue
- Semaphore
- EventGenerator
  - StateChangeEventGenerator
  - InterruptTriggerGenerator
  - TCLKGenerator
  - TimeTriggerGenerator
- AlarmGenerator

- Event
- DataSource
    - DataBuffer
        - CircularBuffer
- Metadata
- DataEntry
- Packer
- CalibrationSystem
- TimingSystem
- DiagnosticSystem
- BufferReadout

3.11.2 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 8 lists the classes that belong to the TBPM package.

Superclass	Subclasses
BPM	*TBPM*
ControlTask	*TBPMControlTask*
Control	*TBPMControl*
DataAcquisitionTask	*TBPMClosedOrbitTask* *TBPMTurnByTurnTask* *TBPMInjectionTurnByTurnTask* *TBLMFastAbortTask*
AlarmTask	*TBPMAlarmTask*
DataSource	*EchoTek*
TimingSystem	*TBPMTimingSystem (TSG)*
Metadata	*TBPMMetadata*
Packer	*TBPMClosedOrbitPacker* *TBPMTurnByTurnPacker* *TBPMTimeSlicedPacker* *TBLMPacker* *TBLMTimeSlicedPacker*
DataEntry	*TBPMData (EchoTekTurnData, TBPMPosition, TBPMChannel)* *TBLMData (TBLMChannel)*
BufferReadout	*TBPMBufferReadout*
CalibrationSystem	*TBPMCalibrationSystem*

Table 8 - TBPM classes

3.11.2.1 TBPM Buffers

Even though the data source of position, intensition 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 – circular (array)
- Slow Abort Buffer – circular (array)
- Fast Time Plot Buffer – circular for high readout rates
- Profile Frame Buffer – FIFO (array)
- Display Frame Buffer – FIFO (single entry)
- Snapshot Buffer – FIFO (single entry)
- Turn By Turn Buffer Buffer – FIFO (array)
- Injection Turn By Turn Buffer – FIFO (array)
- Injection Closed Orbit Buffer – FIFO (single entry)

BLM buffers:

- Fast Abort Buffer – circular (array)
- Display Buffer – FIFO (single entry)
- Fast Time Plot Buffer – circular for high readout rates

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(s) from the Fast Abort Buffer.

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

3.12 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.

3.12.1 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 *Packer*
   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 *Packer* will provide the appropriate algorithm for packing the data according to the users request.
g. Implement subclass of *Packer* (e.g. *TestPacker*)
   For completing the *Test* environment there is the need of a *Packer* 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 *Packer* 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 *InterruptEventGenerator*

Contains interface for install, enable and disable an interrupt handler. The interrupt handler is a method within the class.

e. Implement *TimeEventGenerator*

Subclass of *InterruptEventGenerator*. Configures software timer to call the interrupt handler.

f. Implement *TCLKEventGenerator*

Subclass of *InterruptEventGenerator*. Configures TCLK PMCUCD card and handles its interrupts.

4. Alarms
   a. Implement *AlarmGenerator*

      Provides the ability to send events to the *AlarmTask*.
5. Control
   a. Implement *Control* class

      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. An example is the current status of the system.
6. Tasks (requires on 2a)
   a. Implement ControlTask
   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 *AlarmTask*

      This task receives internal *Alarms*, and decide if external alarms should be generated.
7. Buffer Readout
   a. Implement *BufferReadout*

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

8. 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.
    c. Implement *DiagnosticSystem*
9. 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.

3.12.2 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 containing a BLM data entry.
   c. Implement *TBPMData*
      Class containing a BPM data entry.
   d. Implement *TBPMChannel*
      Structure containing information about a single BPM channel.
   e. Implement *TBLMChannel*
      Structure containing information about a single BLM channel.
   f. Implement *TBPMPosition*
      Structure containing information about a single BPM position (proton and pbar).
   g. Implement *TBPMClosedOrbitPacker*
      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.
   h. Implement *TBPMTurnByTurnPacker*
      Pack strategy for turn-by-turn measurements.
   i. Implement *TBPMTimeSlicedPacker*
      Pack strategy for time sliced closed orbit requests.
   j. Implement *TBLMPacker*
      Pack strategy for BLM requests.
   k. Implement *TBLMTimeSlicedPacker*
      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 *TBPMBufferReadout*

      Add functionality specific to the Tevatron BPM system (the generic *BufferReadout* may be enough).
8. 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*.
9. Calibration
   a. Implement *TBPMCalibrationSystem*

      This class contains the calibration constants being used during run time. There can be several sets of constants, which are used according to the mode and status of the system.

4 Appendix

4.1 Class Diagram

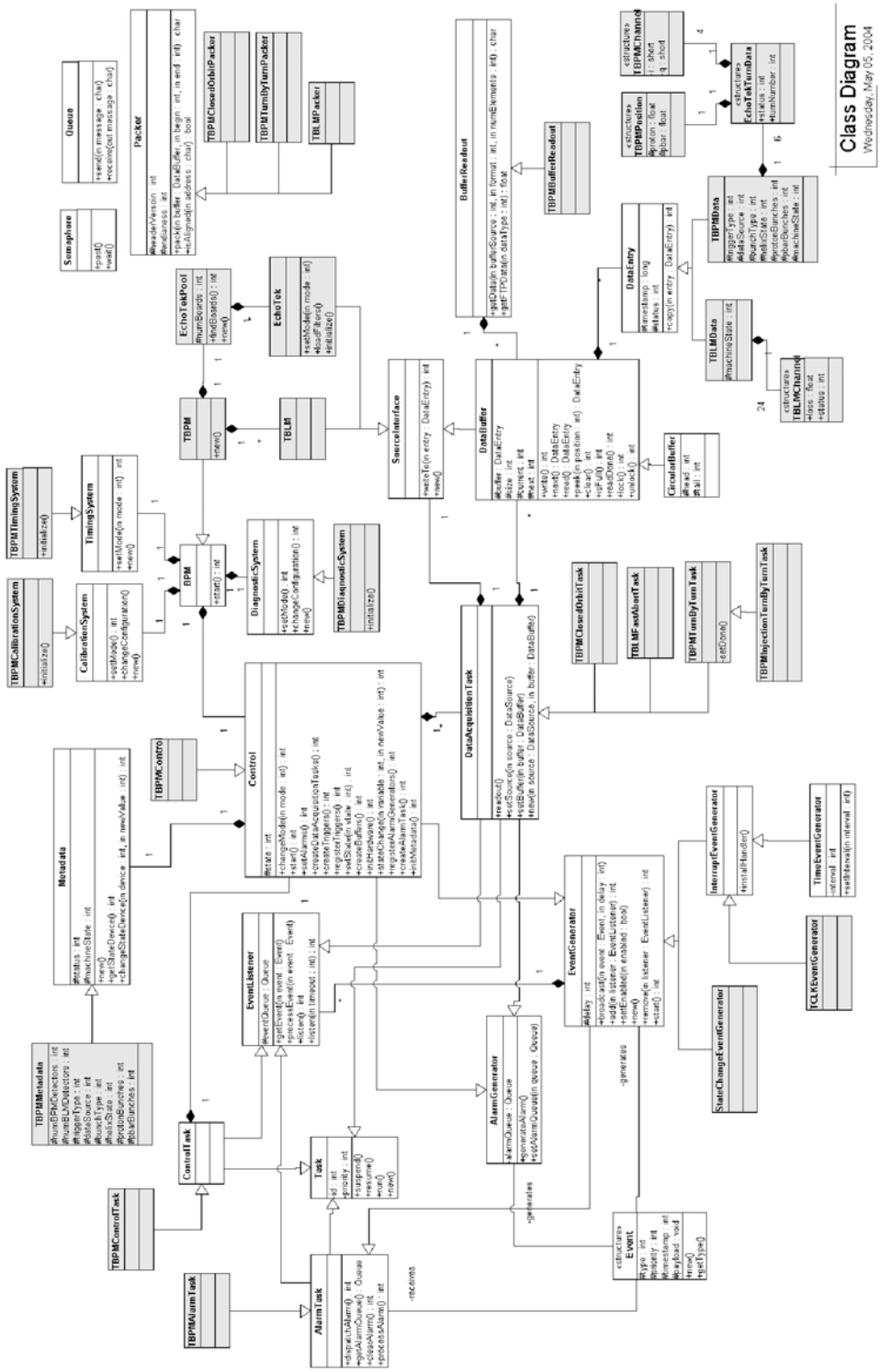

Figure 37 – Complete TBPM front-end software class diagram

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.

[Votava] Margaret Votava, et al. *Tevatron Beam Position Monitor Upgrade Software Specifications for Data Acquisition*, version 22, AD document #860.

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	

Concurrence

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

_____
Steve Wolbers, Project Manager (/ /)

_____
Bob Webber, Deputy Project Manager (/ /)

_____
Jim Steimel, Technical Coordinator (/ /)

_____
Brian Hendricks, Subsystem manager (/ /)

Following persons reviewed this document:

Jim Kowalkowski
Duane Voy
Charles Briegel