Event Driven Data Acquisition
for the
Recycler Ring BPM Front-end

Duane C. Voy

Introduction

The Recycler BPM front-end is designed to provide two types of data acquisition services:

- Command driven – on demand measurements through application program, and
- Event driven – autonomous measurements upon predetermined clock events.

The BPM front-end treats these two measurement types similarly with command driven being a special case of event driven acquisition. This note describes the operation of the event driven acquisition capability of the BPM front-end and touches on issues related to data retrieval by console application programs and the Shot Data Acquisition system.

Events

The events of interest are typically those related to the transfer of beam into and out of the Recycler for protons and antiprotons. Today transfers occur regularly between the Main Injector and the Recycler. Future transfers between the Accumulator and Recycler using the Main Injector as a transport line are anticipated as well. A list of the six most common transfers and associated clock system events follows.

Transfer type	Beam type	Tclk - arm	Bsync - trigger
Main Injector -> Recycler	Proton	$E2	$A2
Main Injector -> Recycler	Pbar	$E0	$A0
Recycler -> Main Injector	Proton	$E3	$A3
Recycler -> Main Injector	Pbar	$E4	$A7
Recycler -> Accumulator	Proton	$96	$A6
Accumulator -> Recycler	Pbar	$E1	$A1

The scheme used to implement event triggered data acquisition involves an arm event transmitted over the Tclk system followed by a trigger event transmitted over the Bsync system. The Tclk arm and Bsync trigger events are always treated as a pair with each arm event occurring a minimum of 400 mSec before its related trigger event. The

400 mSec specification for the time between arm and trigger allows the BPM front-end to abort any untriggered previous measurement and then configure the data acquisition hardware for the new measurement. Tclk arm events are dedicated indicators of beam state and do not have differing meaning across various machine operating contexts. All of the 254 possible Tclk arm events are available, but each enabled Tclk arm event must adhere to the minimum timing and exclusive use requirements stated above. (Note that although supported by the BPM front-end most Tclk events are not appropriate for use as arm events.) The Tclk arm event and Bsync trigger event pair will be referred to simply as an event in further discussion.

The BPM front-end supports data acquisition on a maximum of sixteen events with each capable of making any of the defined position measurements. Events are uniquely identified by the **eEventIndex** enumeration having the following definition:

```
typedef enum {
  kEventIndexMin = 0,
  kEventInteractive   = kEventIndexMin,
  kEventRepetitive,
  kMainInjectorProtonToRecycler,
  kMainInjectorPbarToRecycler,
  kRecyclerProtonToMainInjector,
  kRecyclerPbarToMainInjector,
  kRecyclerProtonToAccumulator,
  kAccumulatorPbarToRecycler,
  kEventIndexMax = 15,
  kNumEventIndexs,
  kEventIndexDefault = kEventInteractive
} eEventIndex;
```

The text of the enumeration constants outlined above fairly succinctly documents the meaning of each value. Two enumeration values deserve additional discussion. The value **kEventInteractive** identifies command driven measurements made whenever a user at a console program issues a measurement request. Interactive measurements are unique in that their arm event is implied by the arrival of the measurement request at the front-end. The value **kEventRepetitive** identifies the (periodic) measurements often referred to as background measurements. If enabled the repetitive measurement runs whenever none of the other measurements are armed. All other enumeration values refer to measurements that will not occur until the related Tclk/Bsync event pair is realized.

Event Programming

The sixteen events described above represent sixteen measurements that may be programmed into the BPM front-end at any one time. The data to be acquired for each of the sixteen measurements is specified by an **Acquisition Specification**[1] having the following definition:

```
// structure sent by ACNet to specify measurement global parameters
class AcquisitionSpecification {
    eAcquisitionControl          _enable;
    eMeasurement                 _measurement;
    eBeamMode                    _beamMode;
    eBeamType                    _beamType;
    eMeasurementType             _measurementType;
    eArmEvent                    _armEvent;
    eTriggerEvent                _triggerEvent;
    ePretriggerControl           _pretriggerEnable;
    eTriggerDelay                _triggerDelay;
    eGlobalDelay                 _globalDelay;
    float                        _intensityThreshold;
    eTriggerTimeout              _timeout;
};
```

Each member of the class is a four byte quantity resulting in a total size of 48 bytes. The definition of each member follows.

```
typedef enum {             // range of _enable  member
  kAcquisitionControlMin = 0,
  kAcquisitionOff   = kAcquisitionControlMin,
  kAcquisitionOn,
  kAcquisitionControlMax = kAcquisitionOn,
  kNumAcquisitionControls,
  kAcquisitionControlDefault = kAcquisitionOff
} eAcquisitionControl;
```

The `_enable` member allows for enabling and disabling acquisition on the associated event. This can be used, for example, to disable the background measurement.

```
typedef enum {      // legitimate range of _measurement  member
  kMeasurementMin = 0,
  kRepetitive  = kMeasurementMin,
  kOneShotTurnByTurn,
  kTimingScan,
  kTurnToTurnPeriod,
  kMeasurementMax = kTurnByTurnPeriod,
```

[1] Acquisition specifications are generally defined by machine personnel and remain unchanged for long periods of time. They may in fact be treated as constants.

```
  kNumMeasurements,
  kMeasurementDefault = kBackground
} eMeasurement;
```

The `_measurement` member describes the type of position measurement to be made. There are two beam position measurement types: repetitive which makes a single turn measurement at a specified frequency, and one-shot turn-by-turn which makes a multiturn measurement upon a specified trigger event. Additional measurement types are for system diagnostics and testing.

```
typedef enum {      // legitimate range of _beamMode  member
  kBeamModeMin = 0,
  kAntiproton  = kBeamModeMin,
  kProton,
  kCalibration,
  kBeamModeMax = kCalibration,
  kNumBeamModes,
  kBeamModeDefault = kAntiproton
} eBeamMode;
```

The `_beamMode` member describes the timing characteristics of acquisition on the associated event. The measurement system uses different temporal offsets from the Bsync trigger event for proton or antiproton (and calibration) measurements. Note that when the calibration source is enabled this field will be ignored and unique calibration timing will be used.

```
typedef enum {             // range of _beamType member
  kBeamTypeMin = 0,
  kInjectExtract   = kBeamTypeMin,
  kHot,
  kHotHead,
  kHotTail,
  kCold,
  kColdHead,
  kColdTail,
  kBeamTypeMax = kColdTail,
  kNumBeamTypes,
  kBeamTypeDefault = kInjectExtract
} eBeamType;
```

The `_beamType` member describes the timing characteristics of acquisition on the associated event. The measurement system uses temporal offset information derived from the Mdat system to measure the various types of beam (e.g., injected, hot or cold). This member specifies which Mdat message is to be used.

```
typedef enum {             // range of _beamMeasurement member
  kMeasurementTypeMin = 0,
  k2_5MHzEnsemble   = kMeasurementTypeMin,
  k2_5MHzBunchbyBunch,
```

```
  k2_5MHzNarrowBand,
  kUnbunchedEnsemble,
  kUnbunchedHeadTail,
  k89KHzNarrowBand,
  kCount,
  kRaw,
  kRawLong,
  kMeasurementTypeMax = kRawLong,
  kNumMeasurementTypes,
  kMeasurementTypeDefault = k2_5MHzBunchEnsembleWB
} eMeasurementType;
```

The `_measurementType` member describes the frequency characteristics of acquisition on the associated event. The analog acquisition subsystem uses various digital filters to make measurements. This member specifies the filter to be used.

```
typedef enum {            // range of _armEvent member
  kArmEventMin = 0x00,
  kArmTclk0x00  = kArmEventMin,
  kArmTclk0xfd  = 0xfd,
  kArmAutomatic  = 0x100,
  kArmEventMax = kArmAutomatic,
  kNumArmEvents,
  kArmEventDefault = kArmInteractive
} eArmEvent;
```

The `_armEvent` member describes the arm event characteristics of acquisition on the associated event. Measurement arm events can be any of the (reasonable) Tclk events, or in the case of interactive and repetitive measurements the measurement request itself.

```
typedef enum {            // range of _triggerEvent member
  kTriggerEventMin = 0x00,
  kTriggerBsync0x00   = kTriggerEventMin,
  kTriggerBsync0xff   = 0xff,
  kTriggerPeriodic,
  kTriggerExternal,
  kTriggerEventMax = kTriggerExternal,
  kNumTriggerEvents,
  kTriggerEventDefault = 0xda
} eTriggerEvent;
```

The `_triggerEvent` member describes the trigger event characteristics of acquisition on the associated event. Measurement trigger events can be any of the (reasonable) Bsync events, an external trigger input to the BPM front-end or the periodic trigger described below.

```
typedef enum {            // range of _pretriggerEnable member
  kPretriggerControlMin = 0,
  kPretriggerDelayOff   = kPretriggerControlMin,
  kPretriggerDelayOn,
  kPretriggerControlMax = kPretriggerDelayOn,
```

```
  kNumPretriggerControls,
  kPretriggerControlDefault = kPretriggerDelayOn
} ePretriggerControl;
```

The _pretriggerEnable member allows for enabling and disabling a delay, measured in turns, between the specified trigger event and the beginning of data acquisition. This pretrigger delay is normally only enabled for injection and extraction measurements where it is used to adjust for kicker system to first turn latency. In the Recycler this value is nominally 33 turns.

```
typedef enum {            // range of _triggerDelay member - turns
  kTriggerDelayMin   = 0,
  kPeriodicFrequencyMin   = 2,
  kTriggerDelayMax   = 65535 - 535,
  kNumTriggerDelays,
  kTriggerDelayDefault = 0,
  kPeriodicFrequencyDefault = 200,
  kPeriodicFrequencyMax   = 500
} eTriggerDelay;
```

The `_triggerDelay` member describes the timing characteristics of acquisition on the associated event. The trigger delay member has differing meaning depending upon the value of the `_triggerEvent` member described above. If a Bsync trigger event is specified then trigger delay is in units of turns and represents a delay between the Bsync trigger event and the first measured data. Normally on event triggered acquisitions the BPM front-end acquires data from the first 2048 turns after the trigger. The trigger delay allows this 2048 sample 'window' to be moved out in time by up to an additional 65,535 turns. If an external trigger event is specified then trigger delay has no meaning. If a periodic trigger event is specified then trigger delay is in units of Hertz and represents the repetitive trigger frequency.

```
const unsigned long int kRfBucketsPerTurn = 588;
typedef enum {            // range of _globalDelay member - RF buckets
  kGlobalDelayMin   = kRfBucketsPerTurn * -2,
  kGlobalDelayMax   = kRfBucketsPerTurn * 2,
  kNumGlobalDelays,
  kGlobalDelayDefault = 0
} eGlobalDelay;
```

The `_globalDelay` member describes the timing characteristics of acquisition on the associated event. The global delay is a signed value in units of 53MHz buckets that represents a delay between the Bsync trigger event and the first measured data. This is a finer adjustment than the trigger delay member described above. The normal value for the global delay member is **ZERO**.

```
The _intensityThreshold member is a single precision float that is used for
intensity discrimination.  This one value is used for all channels on all
measurements.  In turn-by-turn mode the comparison is made against the first
```

```
turn value only.  Note that the intensity data does not have an absolute
calibration of particles per bunch.  Use with CAUTION!

typedef enum {            // range of _timeout member - seconds
  kTriggerTimeoutMin   = 1,           // one second
  kTriggerTimeoutMax   = 5 * 60,      // five minutes
  kNumTriggerTimeouts = kTriggerTimeoutMax - kTriggerTimeoutMin + 1,
  kTriggerTimeoutDefault = 4 * 60,    // four minutes
  kTriggerWaitForever   = 0xffffffff
} eTriggerTimeout;
```

The `_timeout` member describes the timing characteristics of acquisition on the associated event. The timeout value is in units of seconds and specifies how long the BPM front-end should wait after the arrival if an arm event for an associated trigger event. If the trigger does not arrive within the timeout period the measurement will be aborted to allow the repetitive flash to resume.

The sixteen acquisition specifications are stored in an array and indexed by the **eEventIndex** enumeration described above. The first two acquisition specifications (element zero – **kEventInteractive** and element 1 - **kEventRepetitive**) are reserved for use by interactive application programs. The **armEvent** field of these acquisition specification must always be set to **kArmAutomatic**; conversely, all other acquisition specifications must not have an **armEvent** field value of **kArmAutomatic**.

The `_enable` member of each acquisition specification controls data acquisition on all subsequent events of that type. Events may be reconfigured at will by reprogramming the appropriate acquisition specification. When setting an acquisition specification all fields must be set at once – individual acquisition specification fields are not available to application programs.

Data Acquisition

The BPM front-end maintains an individual raw data buffer for each of the sixteen possible acquisition specifications. As a result the data acquired on any given event will be available for retrieval until the next time that measurement becomes armed. Each raw data buffer contains sufficient information to support the three classic position measurements:

- Flash (turnNumber = 1..2048),
- Closed Orbit (beginTurn = 1..2048, numTurns = 1..2048) and
- Turn-by-turn (beginTurn = 1..2048, numTurns = 1..1024).

For closed orbit measurements the **beginTurn** and **numTurns** values must combine to be less than or equal to 2048, and for turn-by-turn measurements the total number of turns must not exceed 1024.

Data Readout

The raw data collected with each acquisition provides the opportunity to obtain data for many different position measurements. After an acquisition has completed the application program specifies the desired position measurement with a **Readout Specification**. Readout specifications identify the raw data to be used, the position algorithm to be employed and the amount of data to be processed. Readout specifications have the following definition:

```
// structure sent by ACNet to request specific measurement data
class ReadoutSpecification {
    eEventIndex                 _eventIndex;
    eDataType                   _dataType;
    eTurnNumber                 _beginTurn;
    eTurnCount                  _numTurns;
    eDataChannel                _channel;
};
```

Each member of the class is a four byte quantity resulting in a total size of 20 bytes. The definition of each member follows.

```
typedef enum {            // range of _eventIndex member
  kEventIndexMin = 0,
  kEventInteractive   = kEventIndexMin,
  kEventRepetitive,
  kMainInjectorProtonToRecycler,
  kMainInjectorPbarToRecycler,
  kRecyclerProtonToMainInjector,
  kRecyclerPbarToMainInjector,
  kRecyclerProtonToAccumulator,
  kAccumulatorPbarToRecycler,
  kEventIndexMax = 15,
  kNumEventIndexs,
  kEventIndexDefault = kEventInteractive
} eEventIndex;
```

The `_eventIndex` member identifies the event that produced the data of interest. This is the same eEventIndex described above. The user can request data associated with any of the sixteen triggering events.

```
typedef enum {            // range of _dataType member
```

```
  kDataTypeMin = 0,
  kBunchedData  = kDataTypeMin,
  kBunch1Data,
  kBunch2Data,
  kBunch3Data,
  kBunch4Data,
  kHotData,
  kHotHeadData,
  kHotTailData,
  kColdData,
  kColdHeadData,
  kColdTailData,
  kDataTypeMax = kColdTailData,
  kNumDataTypes,
  kDataTypeDefault = kBunchedData
} eDataType;
```

The _dataType member identifies the data type of interest. The value specified with this member must be coordinated with the value in the specified event's AcquisitionSpecification _beamType and _beamMeasurement members. For example you can't specify cold beam data if the acquisition specification had requested a bunched beam measurement.

```
typedef enum {             // range of _beginTurn member - turns
  kTurnNumberMin   = 1,
  kTurnNumberMax   = 2048,
  kNumTurnNumbers = kTurnNumberMax - kTurnNumberMin + 1,
  kTurnNumberDefault = 1
} eTurnNumber;
```

The _beginTurn member identifies the turn or turns of interest for flash, closed orbit and turn-by-turn measurements. For flash measurements it specifies the single turn of interest in the range 1 to 2048. For closed orbit and turn-by-turn measurements it specifies the first turn of interest and works in combination with the _numTurns member. Users should note that the sum of the _beginTurn and _numTurns member values may not exceed 2048.

```
typedef enum {             // range of _numTurns member - turns
  kTurnCountMin   = 1,
  kTurnCountMax   = 1024,
  kNumTurnCounts = kTurnCountMax - kTurnCountMin + 1,
  kTurnCountDefault = 1,
  kClosedOrbitCountDefault = 100,
  kTurnByTurnCountDefault = 1024
} eTurnCount;
```

The _numTurns member identifies the number of turns of interest for closed orbit and turn-by-turn measurements and works in combination with the _beginTurn member. Users should note that the sum of the _beginTurn and _numTurns member values may not exceed 2048.

```
typedef enum {             // range of _channel member
  kDataChannelMin   = 0,
  kDataChannelMax   = 47,
  kDataChannels
} eDataChannel;
```

The `_channel` member identifies the channel of interest for turn-by-turn measurements. The turn-by-turn data type returns position and intensity data for a single channel only.

There are six readout specification devices and six readout data devices, one for each of the BPM data types. Data devices are provided with values scaled to engineering units and with values normalized to percent of full scale. The table below indicates the names for the devices.

Specification Device	Scaled Device	Normalized Device	Data Type
R:BPxBFS	R:BPxBFV	R:BPxBFN	background
R:BPxBCS	R:BPxBCV	R:BPxBCN	background closed orbit
R:BPxFLS	R:BPxFLV	R:BPxFLN	flash
R:BPxCOS	R:BPxCOV	R:BPxCON	closed orbit
R:BPxTBS	R:BPxTBV	R:BPxTBN	turn-by-turn
R:BPxTSS	N/A	R:BPxTSN	timing scan

With this scheme the client specifies the desired data by setting the readout specification device for the desired BPM data type and then reads the result from the associated readout data device[2]. The analog and timing diagnostic data devices may be read at any time without the requirement of first setting an associated specification device.

Shot Data Acquisition

Data for the Shot Data Acquisition system is handled through the cooperative efforts of the Machine Sequencer and Shot Data Acquisition applications. The Machine Sequencer makes measurement requests and the Shot Data Acquisition program reads the resulting

[2] Because ACNET is a connectionless protocol that does not support atomic request-reply operations a simple protocol for a blocking set-read has been developed for requesting beam position data. See Appendix A for a description of this protocol.

data at the appropriate time. This collaborative effort will be referred to simply as SDA in further discussion.

SDA can program any or all of the sixteen acquisition specifications described above, but has unique readout specification devices and readout data devices. The unique devices prevent conflicts with interactive users enabling SDA to read data without the need for device arbitration. SDA has access to the same data as interactive users. The table below indicates the ACNet device names for the SDA data devices.

Specification Device	Data Device	Data Type
R:BPxBFC	R:BPxBFD	background
R:BPxBCC	R:BPxBCD	background closed orbit
R:BPxFLC	R:BPxFLD	flash
R:BPxCOC	R:BPxCOD	closed orbit
R:BPxTBC	R:BPxTBD	turn-by-turn

Command Driven Acquisition

Requests for position measurements coming from interactive users at console application programs are treated similarly to event driven requests. The first (element zero) acquisition specification described above is a special acquisition specification device that acts as a software arm event. When this device is set the BPM front-end will configure the specified measurement, wait for the trigger event and collect the specified data.

Priorities and Preemption

The functional specification for the BPM system requires that newly arriving measurement requests have priority over and abort any previously armed measurements that have not completed. This philosophy is applied to acquisition specifications set by interactive users and SDA. All events are processed in the order of arrival. An event must be armed, triggered and acquired to be complete. If the BPM front-end has been armed for an event but not triggered when a new arming event arrives the previous measurement will be aborted and the newly armed measurement will be processed. Interactive user measurement requests will be bumped by event or SDA generated requests that come along while the front-end is waiting for the interactive request to be triggered!

APPENDIX A

The BPM front-end attempts to provide as much position information as possible for each acquisition cycle by buffering enough raw data to produce Flash, Closed Orbit and Turn-by-turn measurements from each trigger. Clients must specify which of these measurements is desired before reading any data. This request-read activity can lead to collisions if multiple clients are requesting data because ACNet does not support atomic request-reply operations. A simple protocol has been devised to support data requests. The protocol works as follows:

Client side:

- read BPM status to see if measurement is completed
- set readout specification
  - if error delay a while and try again
  - else read data ASAP

This is over simplified, it would be good style to include a maximum loop count for example.

Front-end side:

- when measurement is completed set BPM status appropriately
- receive readout specification
  - if waiting for a read of this readout specification return error
  - else begin watchdog timer on new readout specification and wait for read
- if readout specification watchdog timer expires invalidate the readout specification
- receive data read request, cancel timer and return data according to readout specification

With this protocol client requests normally proceed rapidly with a setting of a request and a reading of the associated data. If another client tries to request data before the first client has completed its read the second client will get a busy status. That's OK because the second client will wait a moment and try again. When the first client has completed its data read the BPM front-end will become not busy and be able to accept the second client's request. The watchdog timer used by the front end is set to a nominal value that allows a client to successfully do a setting followed by a reading under normal ACNET conditions – perhaps 2, 3 or more 15 Hz ticks.

End.