LCLS Beam-Synchronous Acquisition Core Software

Till Straumann

July 10, 2018

1 Introduction

The SLAC "Beam-Synchronous Acquisition" (BSA) facility enables users to acquire readings across the entire machine in a synchronous fashion for a well-defined (albeit limited) number of beam pulses. The set of beam pulses on which data shall be acquired is called an "Event Definition" or "EDEF". Only a small number (the exact value differs between LCLS-1 and LCLS-2) of independent EDEFs is available and users have to coordinate access to this limited resource. The exact selection criteria (such as beam rate, time-slot etc.) for a particular beam-pulse to be included in an EDEF is beyond the scope of this document.

For each beam-pulse a bit-mask is broadcast over the timing-system (together with time-stamp and other information). Each bit represents one EDEF and communicates whether data acquired for the respective beam-pulse shall or shall not be stored in the BSA facility. An EDEF with its correspoinding bit set to '1' is called "active". Thus, on every beam-pulse data may be stored for zero, one or more active EDEFs. Therefore, the data store in BSA may be visualized as a two-dimensional array of buffers with each row representing a data source and each column representing an EDEF (fig. 2). When a new data item arrives it is dropped into all the columns that are active within the row corresponding to the particular data source.

An additional feature of the EDEFs is that data may be *averaged*. In fact, the timing system broadcasts additional bitmasks (along with the "active" mask) to support averaging as well as hints for how data with abnormal status shall be handled by BSA.

Once all the data requested by an EDEF are stored in BSA buffers they can be downloaded by the user (usually via EPICS).

The BsaCore software implements the functionality described above and has three interfaces (see fig. 1):

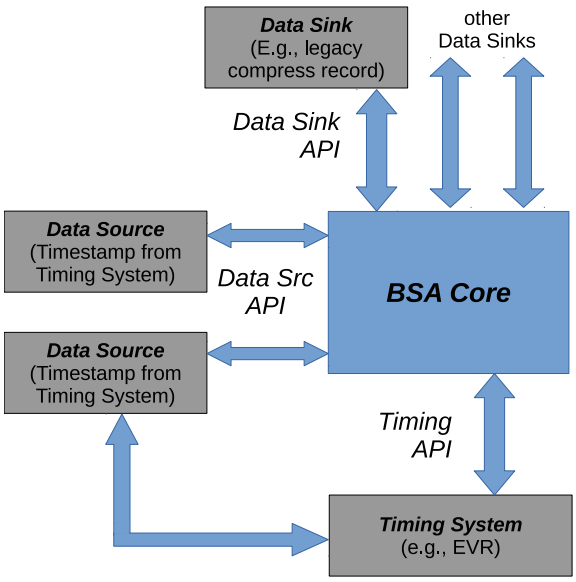

Figure 1. BsaCore with APIs and surrounding modules.

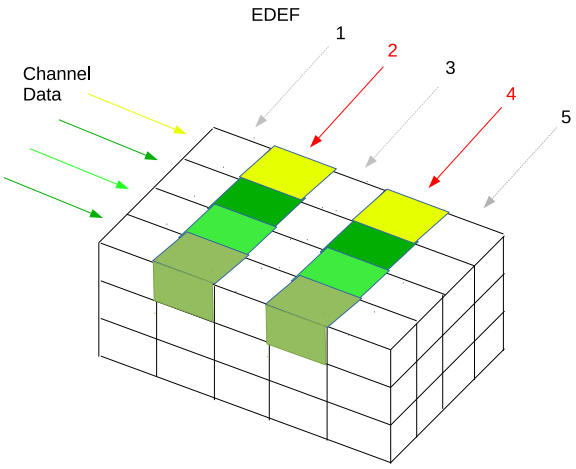

Figure 2. Data-store "matrix". Incoming data are "dropped" (stored) into the boxes corresponding to active EDEFs 2 and 4, respectively.

1. with the *timing system* in order to receive the BSA-related bit-masks as well as the time-stamp.
2. with the *data source* from which BSA receives time-stamped data items.
3. with one (or more) *data sinks* that consume the filtered BSA data.

The legacy implementation of BSA was based on EPICS record-processing and had stringent real-time requirements which were often difficult to meet. This led to data loss and inconsistencies with data sets across multiple systems no longer being synchronous.

The BsaCore implementation discussed in this document aims at improving the behaviour by introducing various levels of buffering that allow for relaxing the real-time requirements significantly. Multi-threaded parallelism improves throughput on multi-core CPUs.

Note that in the context of lcls2 the BsaCore software discussed here is *only* responsible for *slow* data sources which can be processed in software. BSA for high-performance systems (HPS) which is delegated largely to FPGA firmware is handled by a *different* software module, not BsaCore.

2 Requirements

The following list of requirements/design-goals was defined for BsaCore:

Modularization The legacy BSA implementation was tightly coupled with the timing system (which in-turn was tied to the support of a specific hardware). For obvious reasons this monolithic approach is problematic and a new BSA implementation should become an independent module with well-defined interfaces.

Support for LCLS-i as well as LCLS-ii The BSA API as well as its implementation should be able to support LCLS-i as well as (slow) LCLS-ii devices and timing.

Backwards Compatibility The design of BSA should allow for an easy migration path for existing (LCLS-i) applications. Ideally, it should be possible to port an IOC application *without* making any changes, i.e., it should be sufficient to link against new libraries[1].

BsaCore Software EPICS agnostic The BsaCore software should be a stand-alone component and avoid the use of EPICS records.

Reduce RT-Requirements The BsaCore software design should reduce real-time processing requirements as much as possible.

[1] This approach would, however, require a minimal amount of EPICS record-processing in real time; with small modifications to the application such record-processing can be avoided.

C-Language API The BsaCore APIs shall use the C-language.

3 API

Three interfaces are defined for the BsaCore (see fig. 1):

Timing System which notifies the core of the arrival of a new timing pattern (time-stamp and BSA-relevant bit-masks).

Data Source which notifies the core of the arrival of a new, *time-stamped* data item. The data may be discarded or stored for one or more EDEFs, depending on the flags contained in the pattern with a time-stamp that matches the data time-stamp.

Data Sink is notified by BsaCore by means of a user-defined callback that filtered and averaged BSA data are available for consumption.

These APIs shall now be described in more detail.

3.1 Timing API

BsaCore defines an indirect method of registering a callback function with the timing system. The timing system's API defines a callback which is executed on arrival of each fiducial:

```
/**
 * BSA Timing Pattern data
 */
typedef struct BsaTimingData
{
    TimingPulseId       pulseId;            /**< 64bit pulseId  */
    epicsTimeStamp      timeStamp;          /**< TimeStamp for this BSA timing data */
    uint64_t            edefInitMask;       /**< EDEF initialized mask  */
    uint64_t            edefActiveMask;     /**< EDEF active mask */
    uint64_t            edefAvgDoneMask;    /**< EDEF average-done mask */
    uint64_t            edefAllDoneMask;    /**< EDEF all-done mask */
    uint64_t            edefUpdateMask;     /**< EDEF update mask   */
    uint64_t            edefMinorMask;      /**< EDEF minor severity mask   */
    uint64_t            edefMajorMask;      /**< EDEF major severity mask   */
}   BsaTimingData;

/**
 * BsaTimingCallbacks get called w/ 2 parameters
 *   - pUserPvt is any pointer the callback client needs to establish context
 *   - pNewPattern is a pointer to the new BSA timing data
 *
 * Timing services must guarantee the BSA timing pattern data in this structure is all
 * from the same beam pulse and does not change before the callback returns.
 */
```

```
typedef void (*BsaTimingCallback)( void * pUserPvt, const BsaTimingData * pNewPattern );

/**
 * RegisterBsaTimingCallback is called by the BSA client to register a callback function
 * for new BsaTimingData.
 *
 * The pUserPvt pointer can be used to establish context or set to 0 if not needed.
 *
 * Timing services must support this RegisterBsaTimingCallback() function and call
 * the callback function once for each new BsaTimingData to be compliant w/ this
 * timing BSA API.
 *
 * The Timing service may support more than one BSA client, but is allowed to refuse
 * attempts to register multiple BSA callbacks.
 *
 * Each timing service should provide it's timing BSA code using a unique library name
 * so we can have EPICS IOC's that build applications for multiple timing service types.
 */
extern int RegisterBsaTimingCallback( BsaTimingCallback callback, void * pUserPvt );
```

and BsaCore provides an implementation of this callback which is not directly exposed, however. Instead, the user must connect the callback to the timing system by calling

```
/**
  * Obtain the timing callback function provided by BSA
  */
int
BSA_TimingCallbackRegister ( int (*registrar) (BsaTimingCallback, void *) ) ;
```

where usually `RegisterBsaTimingCallback` (note that this function is exported by the *timing system* whereas `BSA_TimingCallbackRegister` is provided by BsaCore()) is passed as the `registrar` argument:

```
#include <bsaApi.h>

...
status = BSA_TimingCallbackRegister( RegisterBsaTimingCallback );
...
```

This way of indirect registration ensures that a valid "context" is always passed to `RegisterBsaTimingCallback`.

3.2 Channels

BSA-sources or -sinks are associated with *channels*. A BSA channel identifies a particular data source that is acquired into BSA. A channel corresponds to a row in the two-dimensional array mentioned in the introduction (the columns of this array represent EDEFs). Channels are identified by *IDs* which are arbitrary, user-defined strings (names). An EPICS *PV* name is customarily used as an ID.

The following entry points are defined for creating and identifying channels:

```
/**
  * Create or find a BsaChannel. If a channel with the given ID already
  * exists then a handle for the existing channel is returned. Otherwise
  * the handle for a newly created channel is returned.
  */
BsaChannel BSA_CreateChannel(const char *id);

/**
  * Look up a BsaChannel. If a channel with the given ID already
  * exists then a handle for the existing channel is returned.
  * Otherwise NULL is returned.
  */
BsaChannel BSA_FindChannel(const char *id);

/**
  * Get the ID of a channel.
  */
const char *BSA_GetChannelID(BsaChannel channel);
```

3.3 Data Source API

Every time a data source (associated with a BSA channel) produces fresh data it must notify the BsaCore by calling `BSA_StoreData()`, providing a timestamp as well as status information:

```
/**
  * Returns 0 on success, nonzero on error.
  */
int
BSA_StoreData(
    BsaChannel      bsaChannel,
    epicsTimeStamp  timeStamp,
    double          value,
    BsaStat         status,
    BsaSevr         severity
);
```

Note that currently only IEEE double-precision floating point values are supported.

For convenience, BsaCore exports a routine to determine status and severity based on comparing a value to "alarm limits". This routine replicates the (unfortunately non-public) algorithm used by EPICS' *AI-record*:

```
typedef struct BsaAlarmLimitsStruct {
        double  lolo, low, high, hihi;
        double  lalm, hyst;
        BsaSevr llsv, lsv, hsv , hhsv;
} BsaAlarmLimitsStruct, *BsaAlarmLimits;

/**
```

```
  * This routine is intended to be fast; if
  * access to the BsaAlarmLimits needs to be
  * protected the caller can employ a spinlock
  * or mutex.
  * NOTE: 'laml' is modified by this routine.
  *       'status' and 'severity' are updated
  *       (i.e., they already must have valid
  *        content since the severity is only
  *        increased by this routine).
  */
void
BSA_CheckAlarms(double val, BsaAlarmLimits levels, BsaStat *status, BsaSevr *severity);
```

3.4 Data Sink API

A BsaCore *sink* consumes averaged and decimated ("filtered") BSA data. These data are aggregated with additional information such as time-stamp, the number of averaged readings and status information. The user registers a callback table with BsaCore and is notified when certain events occur:

- when a new acquisition is started the `OnInit()` callback is executed and provided a time-stamp which identifies the pulse on which the acquisition starts.
- when new results become available the `OnResult()` callback is invoked from the context of a worker thread. Note that under some circumstances multiple results are aggregated and passed to this callback for sake of efficiency.
- when an acquisition is terminated abnormally then `OnAbort()` is invoked with additional status information.

Fig. 3 illustrates the flow of execution of the various callbacks. A sink (callback table) is registered with

```
/*
 * Register a sink with a BSA channel for a given EDEF index.
 *
 * 'maxResults' specifies how many results may be delivered
 * to a single call of 'OnResult'.
 *
 * Returns: status (0 == OK)
 */
int
BSA_AddSimpleSink(
    BsaChannel         bsaChannel,
    BsaEdef            edefIndex,
    BsaSimpleDataSink  sink,
    void              *closure,
    unsigned           maxResults
);
```

A sink is added for a specific channel and a specific EDEF, i.e., to an individual "cell" in the two-dimensional matrix mentioned in the introduction.

Note that multiple sinks can be attached to each such "cell", i.e., the same channel/EDEF combination may have multiple sinks attached. BsaCore iterates over all the sinks when a relevant events needs to be dispatched.

The `maxResults` argument specifies how many "results" BsaCore *may* accumulate before dispatching the `OnResults()` callback. This merely sets a limit but BsaCore may actually deliver less than this limit. It is thus mandatory that the user check the *actual* number of results by inspecting the respective argument to the `OnResults()` callback.

Aggregation of results is desirable for efficiency reasons.

If `maxResults` is set to zero then the implementation is free to pick a value.

The complete prototypes for the callbacks as well as type definitions of the arguments to `BSA_AddSimpleSink()` can be found in the appendix or the current version of the `BsaApi.h` header.

4 Implementation Notes

In this section we discuss some features of BsaCore which are not reflected in the API but are related to the implementation.

4.1 Backwards Compatibility and Latency Remarks

One design goal for BsaCore was backwards compatibility, i.e., making it easy to switch from the legacy, EPICS-record based implementation to BsaCore without changing the upper-layer EPICS-record interface which interacts with the user.

However, since another design goal was relaxing the real-time requirements any user application which implicitly or explicitly relies on BSA processing in real-time will most likely be impacted by a switch from legacy BSA to BsaCore.

Relaxing the real-time constraints required the introduction of several layers of *buffering* (see fig. 4) within BsaCore which has the consequence that the timing at which BSA results are published are more relaxed than with the legacy implementation. This relaxed timing is in addition to any networking delays that a remote user (EPICS OPI) may observe. In particular, there is no guaranteed latency from data being stored into BsaCore to when processed results are published to the sink(s).

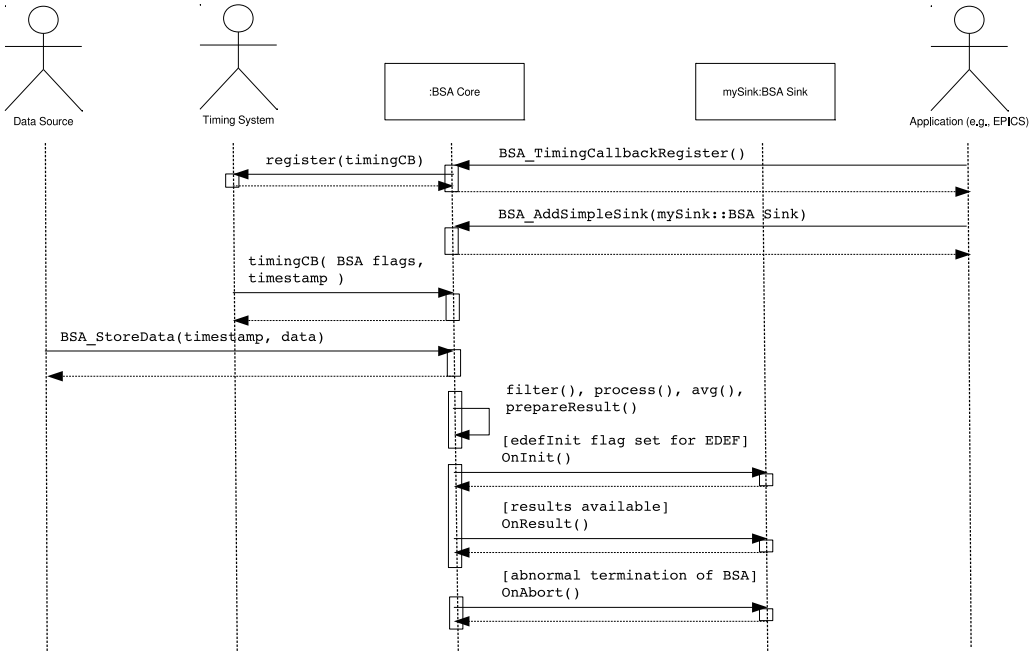

Figure 3. UML sequence diagram showing registration and execution of callbacks. Note that in the BsaCore there are multiple threads of execution; boxes on life-line represent sequential execution from a single thread. I.e., `OnInit()` is guaranteed to precede (one or more calls of `OnResult()` associated with the same acquisition).

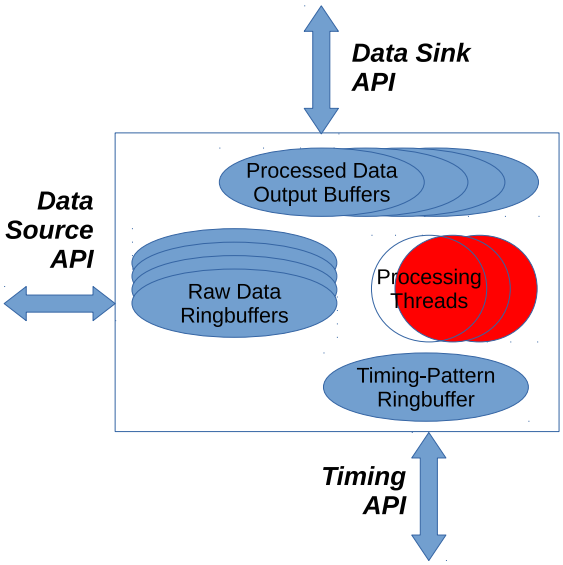

Figure 4. BsaCore implementation details. Timing patterns and raw data samples are stored in ring-buffers for deferred processing. A thread pool filters and averages raw data and feeds the sink API via output buffers.

E.g., the author has heard of users who are using a PV hosted on the EVG to synchronize with termination of BSA. I.e., it was assumed that all (distributed!) BSA data (for a particular EDEF) are available as soon as the aforementioned EVG PV indicated that the EVG had finished the EDEF in question. Obviously, this method relies on real-time processing of BSA on all the IOCs.

A more robust algorithm for synchronizing BSA buffers would e.g, monitor the number of elements acquired and synchronize the main thread:

```
monitor:
  if ( number_of_elements >= desired ) then
    // read the actual data
    ca_get_bsa_channel( this_channel );
    if ( 0 == atomic_clear_flag( this_channel, &outstanding_mask ) then
      // last monitor gets the desired number of elements signals
      signal();
    end if
  endif

start_edef_and_wait:
  atomic_set_flag( all_channels_mask, &outstanding_mask );
  start_edef();
  wait_for_signal_or_timeout();
```

4.2 BsaCore Configuration and Initialization

BsaCore uses lazy initialization, i.e., the software is initialized during the first execution of `BSA_CreateChannel()` or `BSA_FindChannel()`.

Several internal parameters of the BsaCore such as thread priorities and buffer depths can be configured albeit only prior to initialization. The respective calls have the prefix `BSA_Config` and are listed in the `BsaApi.h` header. Note, however, that it is not recommended to change default configuration (maybe with exception of the thread priorities).

4.2.1 Sink Timeout

A sink with `maxResults` > 1 may take a long time (if a BSA runs at a slow rate) or even forever (if a BSA terminates before filling up the last aggregate of results) to update. To prevent this from happening BsaCore periodically flushes all outstanding results and ensures that users don't have to wait excessively. This global period can be adjusted (`BSA_ConfigSetUpdateTimeoutSecs()`) but be aware that the default was chosen to provide an optimal compromise between user-convenience and efficiency.

Appendix

The BSA API Header

```
#ifndef BSA_API_H
#define BSA_API_H

#include <bsaCallbackApi.h>
#include <stdio.h>

#ifdef __cplusplus
extern "C" {
#endif
/*
 * Objectives:
 *     - BSA shall be a separate module/library (unbundled from timing,
 *       EPICS, data acq.);
 *     - interfaces to timing, data source, data sink via APIs
 *     - API should work for lcls-1 and (slow) lcls-2 timing
 *     - make migration of legacy lcls-1 BSA easy
 *     - support low-level interface for data sources (non-EPICS. E.g.,
 *       a ATCA DAQ system might receive pulse-ID/timestamp along with
 *       data)
 *     - EPICS/records layered on top of core API/implementation.
 *
 *
 * Architecture:
 *
 *
 *                                          [Data Sink ]
 *                                            ^
 *                                            | [Data Sink ]
 *                                            |   ^
 *                              Data Sink API |   | [Data Sink ]
 *                                            |   |   ^
 *                                            |   |   |
 *                                            v   v   v
 *              -------------               -------------
 *             | Data Source | <--------> |             |
 *             | (Internal   |Data SRC API|  BSA Core   |
 *             |  TimeStamp) |    ------> |             |
 *              -------------      |        -------------
 *                                 |             ^
 *              -------------      |             | Timing API
 *             | Data Source | <--              v
 *             | (TimeStamp  |              ---------------
 *             |  from EVR)  | <--------> | Timing (EVR) |
 *              -------------               ---------------
 *
 * Timing API:  The timing source provides the BSA core with bsa-relevant timing
 *              information (EDEF bitmasks, timestamp, pulse-id).
 *              Because BSA is not necessarily linked to an application the
 *              interface might be represented by a callback.
 *              Alternatively, a 'no-BSA' library of stubs could be added.
 *
 * Data Source API: Data sources feed timestamped readings to the BSA core.
```

```
 *              The core then stores or discards these readings based on their
 *              unique pulseID/timestamp and qualifiers associated with the
 *              pulseID/timestamp (which were obtained via the Timing API).
 *
 * Data Sink API: Data sinks consume BSA history data; sinks are notified
 *              when e.g., the history changes or a pre-defined number of
 *              items has been produced.
 *
 * Rationale:   The sketched API lends itself to an implementation which
 *                - keeps a history of BSA-relevant timing data (obtained
 *                  from the timing API)
 *                - when a data source stores a new (timestamped) item in
 *                  BSA (via the SRC API) then the implementation can
 *                  look up the timing data associated with the data's
 *                  pulseID and perform BSA operations (averaging, storing)
 *                  when appropriate.
 *              As long as timestamping the data can be done in real-time
 *              such a BSA implementation would be rather insensitive to
 *              processing latencies in BSA itself.
 */

#include <stdint.h>

typedef uint64_t BsaPulseId;

typedef uint16_t BsaSevr;
typedef  int16_t BsaStat;
typedef   int8_t BsaEdef;

/*
 * Opaque BSA channel object (specific to one variable)
 */
typedef struct BsaChannelImpl *BsaChannel;

/*
 * Create or obtain a BsaChannel; if the same name is used for
 * a second time then the same BsaChannel is returned (with incremented
 * internal reference count, i.e., all instances returned by
 * BSA_CreateChannel() must eventually be released (BSA_ReleaseChannel)!).
 *
 */
BsaChannel
BSA_CreateChannel(
        const char *id
);

/*
 * Look up a channel; returns NULL if not found.
 *
 * NOTE: The reference count is incremented if the
 *       channels was found, i.e., the caller must
 *       eventually BSA_ReleaseChannel()
 */
BsaChannel
BSA_FindChannel(
        const char *id
```

```
);

/*
 * Decrement reference count and release all resources associated
 * with this channel.
 */
void
BSA_ReleaseChannel(
        BsaChannel bsaChannel
);

/*
 * Get ID of a channel
 */
const char *
BSA_GetChannelId(
        BsaChannel bsaChannel
);

/*
 * Data Source API; every time a data source has produced a
 * new item it can be stored in BSA with this call. Thus, a
 * low-level driver may store data w/o using any EPICS.
 *
 * Returns: status (0 == OK)
 */
int
BSA_StoreData(
        BsaChannel      bsaChannel,
    epicsTimeStamp timeStamp,
        double          value,
        BsaStat         status,
        BsaSevr         severity
);

/*
 * Helper Routine (which replicates EPICS' aiRecord's 'checkAlarms')
 */
typedef struct BsaAlarmLimitsStruct {
        double  lolo, low, high, hihi;
        double  lalm, hyst;
        BsaSevr llsv, lsv, hsv , hhsv;
} BsaAlarmLimitsStruct, *BsaAlarmLimits;

/*
 * This routine is intended to be fast; if
 * access to the BsaAlarmLimits needs to be
 * protected the caller can employ a spinlock
 * or mutex.
 * NOTE: 'laml' is modified by this routine.
 *       'status' and 'severity' are updated
 *       (i.e., they already must have valid
 *        content since the severity is only
 *        increased by this routine).
 */
void
```

```
BSA_CheckAlarms(double val, BsaAlarmLimits levels, BsaStat *status, BsaSevr *severity);

/*
 * Data Sink API;
 *
 * Simple first version -- assumes *external* history buffers.
 *
 * However, the separation of Src and Sink APIs also allows
 * an implementation e.g., to buffer 'AvgDone' transactions
 * internally on a queue and dispatch the callbacks of the
 * Sink API from a worker thread so that the execution
 * of 'BSA_StoreData' and 'OnAvgDone' are only loosely
 * coupled:
 *
 *     BSA_StoreData()
 *          -> stores data on a (input) queue
 *
 *                          BSA worker thread(s); works
 *                          on input queue computing averages;
 *                          output is stored in output queue.
 *                                  -> store avgs. in output queue
 *
 *                                       Dispatcher thread works on
 *                                       output queue(s) and dispatches
 *                                       Sink callbacks.
 */

struct BsaResultStruct {
        double          avg;
        double          rms;
        unsigned long   count;
        unsigned long   missed; // # of pulses with active EDEF but no data was received
        epicsTimeStamp  timeStamp;
        BsaPulseId      pulseId;
        BsaStat         stat;
        BsaSevr         sevr;
};

typedef const struct BsaResultStruct *BsaResult;

/*
 * Release a result or an array of results; the
 * user may have their own queues to store results.
 * When done, the result must be released.
 *
 * Notes: Results are read-only and could be shared
 *        by multiple sinks.
 *
 *        An array of results is only released once
 *        (the individual members are not released)
 */
void
BSA_ReleaseResults(
        BsaResult results
);
```

```
struct BsaSimpleDataSinkStruct {
        /* called when a new BSA starts */
        void (*OnInit)(
                BsaChannel            self,
                const epicsTimeStamp *initTime,
                void                 *closure
        );

        /* called when one or more results are available
         * to be consumed by this sink.
         *
         * Note: 'results' is an array of multiple
         *       results then BSA_ReleaseResults()
         *       must only be called once.
         */
        void (*OnResult)(
                BsaChannel            self,
                BsaResult             results,
                unsigned              numResults,
                void                 *closure
        );


        /* called with error status (TBD) if a BSA is
         * terminated.
         */
        void (*OnAbort)(
                BsaChannel            self,
                const epicsTimeStamp *initTime,
                int                   status,
                void                 *closure
        );
};

typedef const struct BsaSimpleDataSinkStruct *BsaSimpleDataSink;

/*
 * Register a sink with a BSA channel for a given EDEF index.
 *
 * 'maxResults' specifies how many results may be delivered
 * to a single call of 'OnResult'.
 *
 * Returns: status (0 == OK)
 */
int
BSA_AddSimpleSink(
        BsaChannel        bsaChannel,
        BsaEdef           edefIndex,
        BsaSimpleDataSink sink,
        void             *closure,
        unsigned          maxResults
);

/*
 * Remove a sink
 *
```

```
 * Returns: status (0 == OK)
 */

int
BSA_DelSimpleSink(
        BsaChannel         bsaChannel,
        BsaEdef            edefIndex,
        BsaSimpleDataSink  sink,
        void              *closure
);

/*
 * Obtain the timing callback function provided by BSA
 */
int
BSA_TimingCallbackRegister(int (*registrar)(BsaTimingCallback, void*));

/*
 * Notify BSA that the timing callbacks will no longer be called
 */
int
BSA_TimingCallbackUnregister();

/*
 * Dump statistics (stdout if a null arg is passed)
 *
 * If 'bsaChannel' is NULL then statistics for all
 * channels are printed.
 */
void
BSA_DumpChannelStats(BsaChannel bsaChannel, FILE *f);

void
BSA_DumpPatternBuffer(FILE *f);

/*
 * Configuration or BsaCore parameters.
 *
 * NOTE: These parameters can ONLY be set before the
 *       core is instantiated (= used).
 */

/*
 * Set the depth of the pattern buffer; the
 * value given is the log2(depth).
 */
int
BSA_ConfigSetLdPatternBufSz(unsigned val);

/*
 * Set the (EPICS) priority of the thread that
 * processes the pattern buffer.
 */
int
BSA_ConfigSetPatternBufPriority(unsigned val);
```

```
/*
 * Set the (EPICS) priority of the thread pool
 * that processes BSA filtering.
 */
int
BSA_ConfigSetInputBufPriority(unsigned val);

/*
 * Set the number of threads in the pool that
 * processes BSA filtering.
 */
int
BSA_ConfigSetInputBufPoolSize(unsigned val);

/*
 * Set the (EPICS) priority of the thread pool
 * that processes BSA results.
 */
int
BSA_ConfigSetOutputBufPriority(unsigned val);

/*
 * Set the number of threads in the pool that
 * processes BSA results.
 */
int
BSA_ConfigSetOutputBufPoolSize(unsigned val);


/*
 * Set the (EPICS) priority of the all threads;
 * convenience wrapper which calls all of the
 * above routines.
 */
int
BSA_ConfigSetAllPriorites(unsigned val);

/*
 * Set the timeout at which aggregated
 * results are flushed.
 */
int
BSA_ConfigSetUpdateTimeoutSecs(double seconds);

#ifdef __cplusplus
}
#endif

#endif
```

Integration with the EPICS `event` module

Porting the legacy BSA (which historically has been part of the `event` module) to BsaCore was possible with minimal changes:

- `bsaRecord` was modified to support the BSA-related fields (`VAL`, `RMS`, `PID`, ...) being *arrays* of length `NELM`. A `NORD` field was added which communicates the *actual* number of elements contained in the arrays (`NELM` specifies the *maximum* of supported elements).

  These modifications are fully backwards compatible and the legacy BSA implementation may still be used (`NELM` is set to 1 in this case).

- A new device-support module for `bsaRecord` was created which implements a BsaCore sink using the `bsaRecord`'s `NELM` field to define the `maxResults` parameter. Thus, the problematic update rate of the chained `compress` records can be reduced simply by increasing `NELM` which causes the `compress` records to be updated less frequently but with multiple new values at once. EPICS automatically takes care of this without any further "special" code. A value of `NELM` ≈ 10 has proven to reduce CPU load significantly.

  Note that while reducing `NELM` also reduces the delay from data being stored into BsaCore to results being published this delay cannot be completely eliminated (even with `NELM` = 1) due to additional latency caused by internal buffering (see also above).

- A new device-support module for the `aoRecord` which forwards data into legacy BSA. While BsaCore-aware IOC applications are recommended to use the BSA API for storing data into BsaCore (`BSA_StoreData()`) this device-support module enables transparent porting of (BsaCore-unaware) applications.

Using the aforementioned components is possible to switch existing applications from legacy BSA to BsaCore simply by making small changes to the BSA-related database templates and linking the BsaCore library.