20450 Century Boulevard
Germantown, MD 20874
Fax: (301) 515-7954

BCP Driver

Software Design Specification (SDS)

Revision A

3/02/11

Revision Record	
Document Title: **Software Design Specification**	
Revision	Description of Change
A	i. Initial Release (Release v 1.0.0.5)

Note: Be sure the Revision of this document matches the QRSA record Revision letter. The revision letter increments only upon approval via the Quality Record System.

TABLE OF CONTENTS

1 Scope

This document describes the design and usage of the BCP driver and its APIs.

2 References

The following references are related to the feature described in this document and shall be consulted as necessary.

No	Referenced Document	Control Number	Description
1	bcp_users_guide_v0_4_0.pdf	Version 0.4.0 Feb, 2011	KeyStone Bit Co-Processor (BCP)
2	Navigator_Users_Guide_060.pdf	Version 0.6.0 Oct, 2010	TMS320TCI66xx Multicore Navigator (CPPI)

Table 1. Referenced Materials

3 Definitions

Acronym	Description
API	Application Programming Interface
BCP	Bit Processing Coprocessor
CPPI	Communications Port Programming Interface
LLD	Low Level Driver
HLD	High Level Driver
QMSS	Queue Manager Sub System
LTE	Long Term Evolution
WCDMA	Wide-band Code Division Multiple Access
TD-SCDMA	Time Division Synchronous Code Division Multiple Access
3GPP	3rd Generation Partnership Project
OSAL	OS Abstraction Layer
Tx	Transmit
Rx	Receive

Table 2. Definitions

4 Overview

The Bit processing coprocessor (BCP) is an acceleration engine for wireless infrastructure. It accelerates those physical layer (Layer 1) functions that:

- consume a lot of DSP cycles when implemented in software
- have a little or no customer intellectual property
- simplify the data flow or use model

This typically includes bit processing functions like encoding, scrambling, interleaving, rate matching, soft slicing and modulation.

The BCP supports LTE, FDD WCDMA, TD-SCDMA, and WiMAX 802.16-2009 standards.

BCP has several sub-modules within it, each performing a distinct function. Following are the major sub-modules within BCP:

- CDMAHP – Navigator interface
- CRC – Cyclic redundancy check
- ENC – Turbo and convolutional encoder
- RM – Rate matcher
- MOD – Modulator
- INT – Interleaver (for WCDMA)
- COR – Correlation engine
- DNT – De-interleaver (for WCDMA)
- SSL – Soft slicer
- RD – Rate De-matching engine
- DIO – Direct I/O
- TM – Traffic manager

Each of these sub-modules has a corresponding header defined. These headers when put on a packet, specify the configuration parameters to use for processing the packet payload by a given sub-module. Packets arriving at BCP can take different paths inside BCP for processing based on the headers put on the packet and the order of the headers specified. Shown below is the BCP input packet's header format. The "global" and "TM" headers are mandatory while the rest are optional.

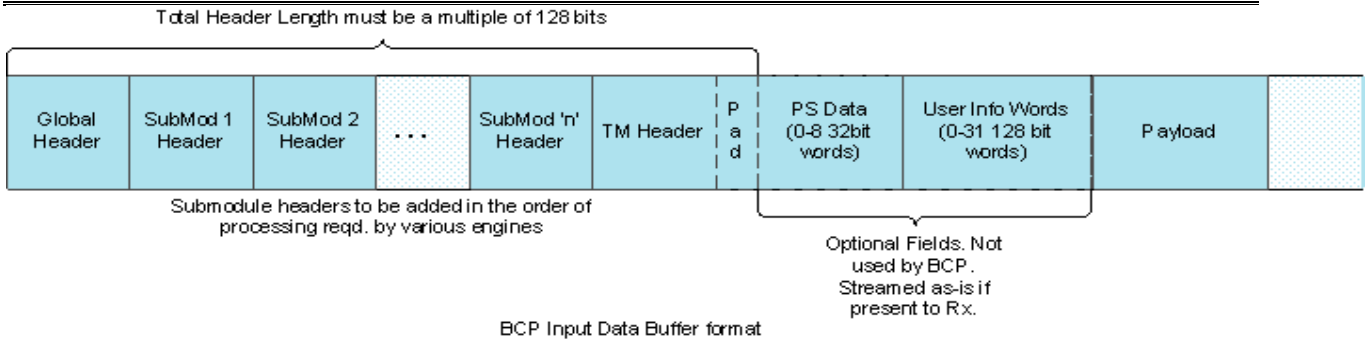

Figure 1. BCP Input packet header format

BCP H/w can be configured or communicated via the following two interfaces:

- Configuration/memory mapped registers (MMRs):

  Useful in setting up or initializing BCP hardware with parameters that are static, i.e., parameters that do not change on a per packet basis. These registers are meant to setup BCP features like error handling, data logger for debugging, and various sub-module initialization parameters. Please note that not all sub-modules within BCP have a MMR interface. In absence of an MMR interface, the only way to configure a sub-module would be using the corresponding sub-module header.

- Navigator (CPPI/QM) interface:

  The CPPI/QM interface is the data I/O interface for BCP, i.e., handles all data movement in and out of BCP. If an application wishes to send some data for BCP processing, it would have to wrap the data in a CPPI packet descriptor and push to BCP Tx queue for BCP to receive and process it. Similarly, the output from BCP is sent to a queue from which any other peripheral/DSP can read. Several hardware queues have been hard-wired for use by BCP for input. The queue to which BCP output must be sent to can be programmed by the application using CPPI flows.

The purpose of this document is to describe how to use the BCP driver in communicating with BCP H/w and its sub-modules using the two interfaces mentioned above.

This document assumes that the user has familiarized himself/herself with the BCP and Navigator (CPPI/QM subsystem) user's guides.

5 Design

5.1 Goals

The design goals for the BCP driver can be briefly summarized as follows:

1. ***Provide basic low performance impact APIs:***

Provide APIs at various levels of granularity, i.e., expose a higher layer abstract, task oriented APIs (for example, APIs to configure BCP queues, flows, submit BCP requests etc) and also a lower layer set of LLD APIs that can be used to quickly configure the hardware without much performance impact.

2. ***OS independent:***

   The driver should be designed to be OS independent to ease porting from one OS to another.

3. ***Multi-core aware:***

   The BCP accelerator is designed to be accessible across multiple cores in a multi-core chip. Hence, the BCP driver should be aware of the multi-core challenges and must provide for mechanisms to address them.

4. ***Unified APIs for BCP Remote and Local access:***

   BCP accelerator present on the Turbo Nyquist (TN) SoC can be accessed not only by applications on TN (BCP local access), but also can be accessed by applications/IPs present on other SoCs present on the device remotely via SRIO. The driver should be designed to handle the "local" and "remote" use cases seamlessly.

Keeping the design goals in mind, the BCP driver is organized as follows:

1. ***Higher Layer APIs:***

   The BCP higher layer exports APIs that can be used to initialize BCP, configure its Tx queues, setup its receive flows, submit BCP requests, and retrieve any output from BCP.

   These APIs in turn use:

   - CPPI and QMSS library APIs to setup the BCP CPPI DMA
   - BCP Lower layer (LLD) APIs to setup any BCP registers required and to format the configuration provided by the application to BCP headers that the hardware understands.
   - OS Abstraction layer (AL) APIs to keep up with OS independence
   - Device (SoC) specific API call-outs to initialize BCP appropriately for the SoC.
   - Transport layer call-outs to setup, send and receive data over SRIO if using BCP remotely.

2. ***OS Abstraction Layer (OS AL):***

   The OS AL defines APIs for memory management, multi-core and process synchronization. Separating these functions out of the BCP driver simplifies porting of the LLD between different OS.

3. ***Device Specific Layer:***

   The device specific layer implements the APIs required by the driver to perform SoC specific initialization.

4. ***Transport Layer APIs:***

   The transport layer implements APIs required to initialize, send, and receive data over SRIO. The transport layer call-outs are called from various points during BCP initialization and data path if BCP is being used remotely.

5. ***Low Level Driver (LLD) APIs:***

   This layer exposes various APIs useful in reading and writing values from/to BCP MMRs. It also exports various utility APIs that can be used by an application/driver to format user specified configuration data (3GPP parameters) to BCP understandable header formats. The BCP Lower layer APIs in turn use the BCP CSL Register Layer (RL) file from the CSL package for BCP memory mapped register overlay and sub-module header definitions

The BCP driver organization and its dependencies are as shown below:

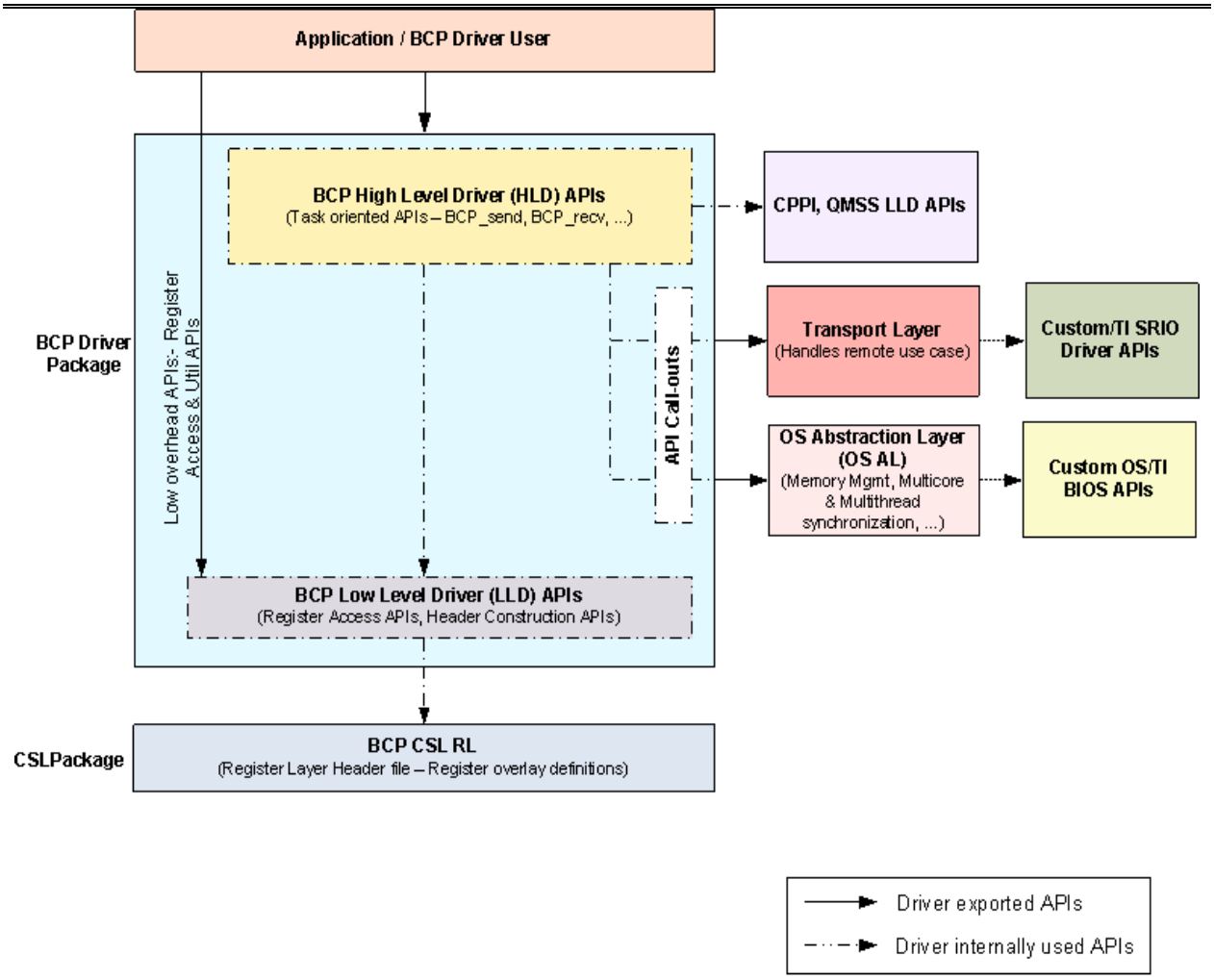

Figure 2. BCP Driver Organization

5.2 Driver Terminology

This section defines the commonly used terms through out the driver and this document.

1. ***BCP Transmit Object:***

    a. A BCP transmit object captures all the transmit (Tx) properties of an application that wishes to use the driver to send data to BCP.

    b. Some properties that characterize a Tx object are:

        i. BCP Tx queue number to use for the requests submitted using this object.

    c. Currently, the BCP driver doesn't manage Tx Free descriptor queue (FDQ) or Tx free descriptors (FDs). These are expected to be managed by the application.

d. Since the buffers allocated on the Tx object's Tx FDQ can be local to the core's memory, in a multi-core setup its recommended for the application to create at-least one Tx object (even if using the same BCP Tx queue) on each of the cores from which it wishes to send data to BCP using the driver.

e. One or more Tx objects can be created on any of the cores in the system.

2. BCP Receive Object:

a. BCP receive object captures all receive (Rx) properties of an application that wishes to use the driver to receive results from BCP.

b. Some properties that characterize a Rx object are:

i. Rx queue number to which BCP output must be sent to for application to use.

ii. CPPI flow configuration. Specifies Rx FDQ from where Rx buffers must be picked up by BCP, and Rx descriptor formatting rules such as whether PS info would be present on output descriptors, if so its location, tags to be put etc. The driver exposes the complete CPPI flow configuration to the application. The application is responsible for setting up the Rx FDs, Rx FDQ and setting up all other flow configuration. The driver programs the BCP CPDMA flow table with configuration provided as is.

iii. BCP Traffic manager flow configuration. Specifies output data formatting rules like final endiannessof data, any data swapping required, any special packet type or PS flag assignments to put on output packet by BCP etc. The driver programs the BCP TM flow table with configuration provided.

iv. If the application wishes to use interrupts to receive the result or if it'd like to poll instead.

v. QM Accumulator configuration. The driver exposes the complete accumulator configuration to the application. The application is responsible for setting up the accumulator list and its configuration. The driver programs the accumulator with the configuration provided as is. Also the application is responsible for doing the ISR handling for the accumulator its sets up in this mode.

c. Since the buffers allocated on the Rx object's Rx FDQ can be local to the core's memory, in a multi-core setup its recommended for the application to create at-least one Rx object on each of the cores from which it wishes to receive BCP output using the driver.

d. The driver also opens and maintains an Rx queue corresponding to the Rx queue number specified.

e. For the sake of consistency in the system, it is suggested that there be only one application in the system retrieving results from any given Rx queue.

f. One or more Rx objects can be created on any of the cores in the system.

5.3 Internal Data Structures

Internally the BCP driver maintains a global data base (shared across all the cores) of the following data structures:

1. BCP Instance Object:

a. This structure is maintained on a per BCP peripheral instance basis.

b. Holds the BCP peripheral instance number.

c. Reference count to track the number of applications using this instance. Used to ensure that the instance is not de-initialized while in use by some other application.

d. BCP driver mode. Tracks whether the application setup the BCP driver in "local" or "remote" modes during *Bcp_init ()*.

e. Maintains the BCP CPDMA's CPPI Object handle for the corresponding instance number.

f. Maintains the BCP LLD handle for the corresponding BCP instance.

g. Tracks relevant state information for all the Tx queues and Rx flows corresponding to this instance.

h. BCP transport layer callback functions call table.

The following data structures are maintained on a per BCP peripheral instance basis:

1. BCP Tx Queue State Objects:

a. There are 8 such object instances maintained in an array referenced by the Tx Queue Number in O (1) time for a given BCP peripheral instance.

b. Holds the transmit queue number (can range between 0 – 7 both inclusive) itself.

c. Hold the Reference counter for this queue object. The reference counter tracks the number of applications that are using the BCP Tx Queue object. This counter is incremented every time *Bcp_txOpen ()* API is called and returns successfully and is decremented whenever *Bcp_txClose ()* API is called. When this reference count reaches zero, the transmit queue configuration is cleaned up and this object is considered invalid and cannot be used for submitting BCP requests by an application unless initialized again using *Bcp_txOpen ()* API.

d. Maintains the CPPI Tx channel and transmit queue handles for this Tx queue object.

2. BCP Flow State Objects:

a. There are 64 such object instances maintained in an array referenced by the flow Id number in O (1) time.

b. Holds the CPPI Flow Id itself (can range between 0 – 63 both inclusive).

c. Reference count on the flow. Reference count for a flow is incremented whenever a Rx object is opened that uses it and is decremented when a Rx object that uses it is closed.

d. CPPI Rx flow handle.

e. Also tracks the PS location, PS info present settings from the flow configuration. Used when processing BCP output, in obtaining PS data from the data buffer.

3. BCP Rx Objects State Info:

a. Tracks all the Rx objects opened in the system.

b. Rx object Id to identify each of the Rx objects uniquely in the system.

c. Boolean flag to track whether the Rx object is valid or not. A Rx object is marked valid in the Rx object database when a Rx object is successfully opened and its info is stored in the database. This flag is cleared when the Rx object at the corresponding entry is closed.

d. Receive queue number and flow Id the Rx object was configured to use.

4. BCP Transport layer call-outs:

a. Tracks the call-back functions registered by the application at *Bcp_init ()* time for BCP transport layer over SRIO.

b. Depending on whether the driver was opened in "local" or "remote" modes, the driver invokes various call-back functions at different points of time.

5.4 External Driver APIs

Please refer to the Doxygen output from the BCP driver code deliverable for the data structure definitions, API prototypes and their details for both BCP LLD APIs and BCP Higher Layer APIs.

5.5 BCP LLD API Usage

The BCP low level driver APIs, i.e., BCP register access and header construction utility APIs are straightforward and hence are not elaborated here. Moreover, the parameters to be passed to the header construction APIs vary from one wireless standard to another and are application dependent and hence are not elaborated here. Please refer to the Doxygen comments and example for usage details.

5.6 BCP Higher Layer API Usage

This section discusses the BCP Higher layer APIs in detail. Please refer to the Doxygen output and the example code packaged in "*example*" directory for usage details.

5.6.1 Initialization

Following are the steps to be followed in setting up the BCP driver for BCP processing:

5.6.1.1 System Level Initialization

1. The first step to using the BCP driver would be to initialize it. This MUST be done only once during the system startup using the API '*Bcp_init ()*'.

   The inputs to this function are:
   a. Instance Number: The BCP peripheral number to initialize.

   b. Driver Mode: Mode in which BCP driver must be opened. Possible options are "Bcp_DrvMode_LOCAL" or "Bcp_DrvMode_REMOTE".

      - Set to "Bcp_DrvMode_LOCAL" if BCP driver is being initialized on a SoC on which BCP is present. In this mode, the driver forwards all BCP requests to BCP present on the same SoC directly.

- Set to “Bcp_DrvMode_REMOTE” if BCP driver is being initialized on a SoC on which BCP is NOT present. In this mode, the driver forwards all BCP requests to BCP present on another SoC on the device via SRIO.

c. BCP Init Configuration: BCP peripheral initialization configuration.
   - cpdmaNum: CPDMA number corresponding to this BCP instance.
   - baseTxQueueNum: BCP base Tx queue number for this BCP instance.
   - cfgRegsBaseAddress: Configuration registers (MMRs) base address for this BCP instance.
   - Transport layer callback functions:

     The connection between BCP and DSP application on remote device can be viewed as a “tunnel”. All packets sent by the DSP application on remote device, get wrapped inside a SRIO header and are transported to the SoC containing BCP via SRIO switch, wherein the SRIO header gets stripped off and data gets forwarded onto BCP for processing. Similarly, on the reverse path, i.e., output from BCP is sent to SRIO first to be wrapped inside a SRIO header and sent over the SRIO switch to remote device, and from there sent back to the application’s Rx queue after SRIO strips off its header (assuming that BCP output must be sent back to the same DSP application from which the request had come).

     - *BcpTunnel_txOpen*:- Called from *Bcp_txOpen ()* API to initialize one endpoint of the Tx tunnel between SoC having BCP and remote device. Application developer should implement the necessary logic here to perform any SRIO setup required to send packets to BCP on the remote device as this device.
     - *BcpTunnel_txClose*:- Called from *Bcp_txClose ()* API to close a BCP tunnel Tx endpoint.
     - *BcpTunnel_rxOpen*:- Called from *Bcp_rxOpen ()* API to initialize one endpoint of the Rx tunnel between device having BCP and remote device. Application developer should implement the necessary logic here to perform any setup required to receive packets from BCP on the same/remote device as this device.
     - *BcpTunnel_rxClose*:- Called from *Bcp_rxClose ()* API to close a BCP tunnel Rx endpoint.
     - *BcpTunnel_send*:- Called from *Bcp_send ()* API to send out a packet through the tunnel to a remote BCP device. Invoked by BCP driver only if it was initialized in "remote" mode. On BCP “local” device, the

packets are sent directly to BCP Tx queue, SRIO intervention is not required.

- *BcpTunnel_recv*:- Called from *Bcp_recv ()* API to receive output from BCP using tunnel from a remote device. Invoked by BCP driver only if it was initialized in "remote" mode.

- *BcpTunnel_freeRecvBuffer:-* Called from *Bcp_rxFreeRecvBuffer ()* API to free an Rx packet obtained using *Bcp_recv ()* API. Invoked by BCP driver only if it was initialized in "remote" mode.

2. Validates input and resets the driver's global data structures and state info for the instance number specified.

3. It opens the BCP CPDMA corresponding to the instance; saves the transport layer call table provided and initializes all instance related internal data structures.

5.6.1.2 Application Level Initialization

1. Each application that would like to use the BCP driver would need to obtain a BCP driver handle using *Bcp_open ()* API.

   The inputs to this function are:
   a. BCP Instance Number: BCP peripheral instance number to use for this application.

   b. BCP driver configuration: Not used currently by driver. Place holder for any future extensions to this API.

   The *'Bcp_ open ()'* API does the following:

   i. This API returns a driver handle containing all information pertinent to the BCP instance the application is using. This handle must be provided in all future calls to the driver APIs.

2. **Tx Initialization:**

   This step applies only if the application would like to act as a sender, i.e., be able to send data to BCP.

   The application can submit BCP requests to the BCP H/w by opening a "***Tx Object***". The Tx object captures all relevant Tx parameters of the application such as the BCP Tx queue to use for processing.

   A Tx object can be created by a sending application using "*Bcp_txOpen ()"* API by specifying the driver handle obtained from "*Bcp_open ()"* API and the transmit configuration

captured in "*Bcp_TxCfg"* data structure. The configuration details follow:

a. Choose the transmit queue number (0-7) using which the BCP requests would be submitted.

b. Specify suitable BCP tunnel Tx endpoint configuration for the device.

   i. On BCP local device, a valid Tx endpoint configuration must be passed in 'pTxEndpointCfg' if the BCP Tx queue being opened needs to be made accessible from remote device too. If no remote access is required for this BCP Tx queue, then 'pTxEndpointCfg' parameter can be set to NULL.

   ii. On BCP remote device, the BCP Tx queue configuration 'pBcpTxCfg' must be set to NULL since remote device has no direct access to BCP CDMA or its Tx queue. Configuration required to open a BCP Tx remote endpoint must be passed in 'pTxEndpointCfg' and the driver in turn passes this as-is to the registered Tx endpoint open API.

c. Finally, call *Bcp_txOpen ()* API with the above input parameters.

This API sets up the BCP Tx queue if driver was setup in local mode. In remote mode, the driver calls *BcpTunnel_txOpen()* API, the callback function registered at init time to setup BCP tunnel's Tx endpoint. On success, this function returns a Tx handle that must be used to send data to BCP using the driver. On error, the function returns NULL to indicate an error and an appropriate error is printed on the console.

3. **Rx Initialization**:

This step applies only if the application would like to act as a receiver, i.e., be able to receive output from the BCP using the driver APIs

Setup the "*Rx object*" configuration to indicate which queue the application wants it results to be buffered and the result configuration itself using the data structure '*Bcp_RxCfg*'.

a. Fill in the Rx / Destination Queue number where the BCP output should be placed for the application in '*rxQNum'* parameter. If the application has no preference for a specific Rx queue number, it can specify "-1" in which case the driver would pick the next available queue for the destination queue to buffer BCP output.

b. Specify the complete CPPI flow configuration in *'flowCfg'*. Please note that the driver does not manage the Rx FDs or Rx FDQs. The application is responsible for setting these up appropriately while filling in the CPPI flow configuration.

c. Specify the BCP TM flow configuration in *'tmFlowCfg'*.

d. Pick a mode to retrieve results using this Rx object. Possible modes are:

- *Use Interrupts* – The application can choose interrupts for the Rx object by setting '*bUseInterrupts'* to 1. Also specify the complete accumulator configuration in *'accumCfg*' and the driver will program the accumulator with configuration provided as is.

  Note: Please note that there might be some restrictions on the use of accumulator channels on various cores for each SoC. Please consult the Accumulator firmware specification and Device level specification before picking a channel number to use here.

- *Use Polling* – The application can choose to poll on a Rx object for results. It can configure the Rx object for polling by configuring *'bUseInterrupts'* flag to 0. In this case, any Rx queue can be specified for this Rx object.

e. On Local device, a valid Rx object configuration MUST be provided in the input parameter 'pBcpRxCfg'. This API creates a BCP flow using the configuration provided. It also sets up the accumulator interrupts if interrupts are requested for this object. If any configuration needs to be done to redirect output from BCP to a remote device, then a valid configuration can be passed in 'pRxEndpointCfg' parameter and the driver will invoke the BCP transport layer's Rx endpoint open function *BcpTunnel_rxOpen ()*. This parameter can be set to NULL if BCP output is to be processed on the same device or if no additional transport layer configuration is required.

f. On Remote device, a valid Rx endpoint configuration must be passed in the 'pRxEndpointCfg' parameter. This API in turn calls the transport layer's Rx endpoint open API *BcpTunnel_rxOpen ()* to do the needful configuration to receive results from BCP on that device.

g. Finally call '*Bcp_rxOpen* ()' API with the BCP Rx object configuration and appropriate Rx tunnel endpoint configuration specified above and the driver handle obtained earlier in Step 1 of the initialization sequence.

The '*Bcp_rxOpen ()*' API based on the configuration specified, sets up a Rx queue, CPPI flow and TM flow on BCP local device. On remote device, Rx remote endpoint is opened.

On success, this API returns an Rx object handle that the application must use for receiving results from BCP using the driver APIs.

Note:

- The driver does no validation if the same destination (Rx) queue is used by more than one Rx objects. If two or more Rx objects are using the same Rx queue, then it's application's responsibility to ensure that results are retrieved using the correct Rx object for proper operation.

5.6.2 Transmit Path

Steps to follow to prepare and submit a BCP request using the driver APIs are as follows:

a. Prepare BCP headers:

- The first step to building BCP packets is to translate the application's L1 parameters (3GPP parameters) into BCP H/w parameters.
- BCP LLD exposes data structures for setting up various BCP sub-module parameters.
- Derive the LLD header configuration parameters (one to one mapped to BCP H/w parameters) from the 3GPP parameters using the application's utility APIs.
- Some sample 3GPP → LLD mapping APIs provided with driver in example directory. These can be used as a sample starting point for the application, but they might need customizations by application on a need basis.

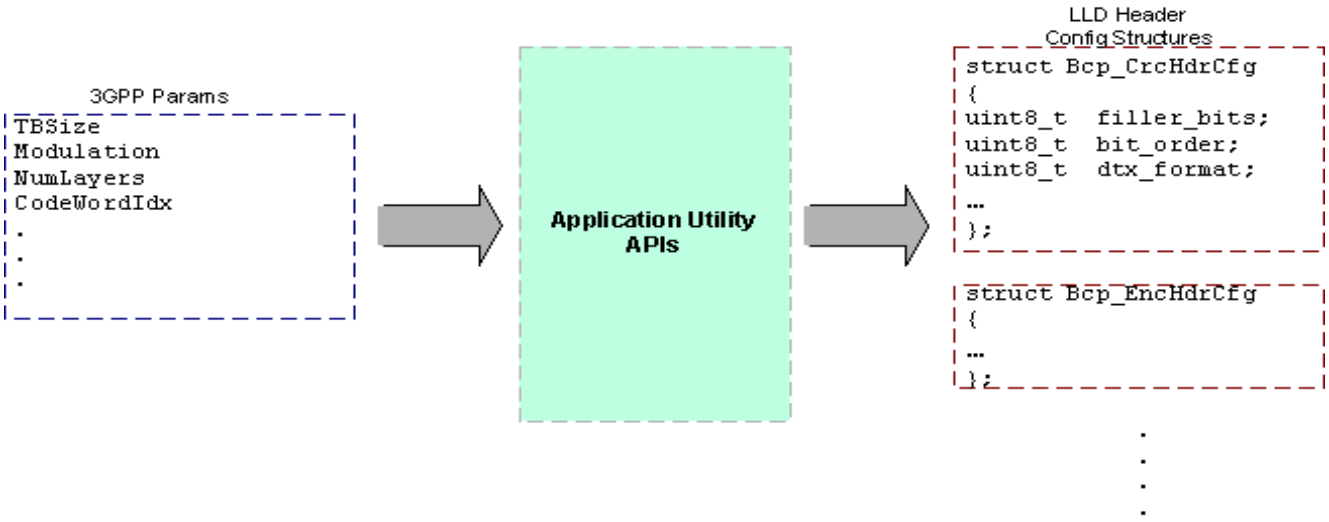

Figure 3. BCP header preparation

b. Add BCP headers:

- The next step is to use the BCP driver's LLD header construction APIs and add the various BCP headers to the packet.
- Input to the LLD header construction APIs is LLD header configuration parameters prepared earlier in previous step.

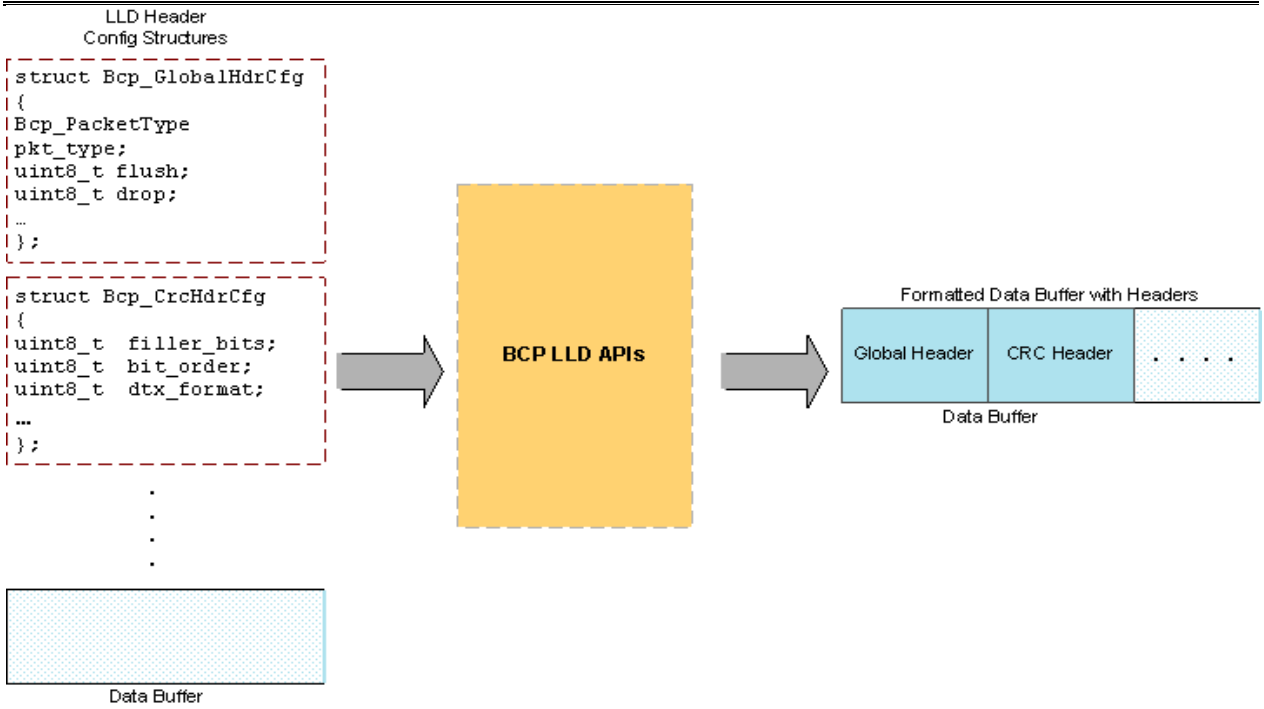

Figure 4. BCP header addition

c. Send Packet out to BCP:

- Finally, to send the packet to BCP, attach the data buffer containing BCP headers to a host CPPI descriptor.
- Use the BCP driver's HLD API *Bcp_send ()* API to send the packet/descriptor to BCP.
- The API *Bcp_send ()* takes the following parameters as input:

  i. BCP Tx handle: Obtained from *Bcp_txOpen ()* API

  ii. Packet descriptor: Packet with BCP headers and payload to be sent to BCP for processing.

  iii. Data Buffer Len: Length of data being sent.

  iv. Destination address: Set to NULL if driver initialized in "local" mode. Set to a valid destination address for remote mode. Indicates to which device the BCP request must be sent to. Not used/interpreted by driver, this parameter is as-is passed to BCP transport layer callback API *BcpTunnel_send()* for forwarding the packet onto BCP on remote device using SRIO tunnel.

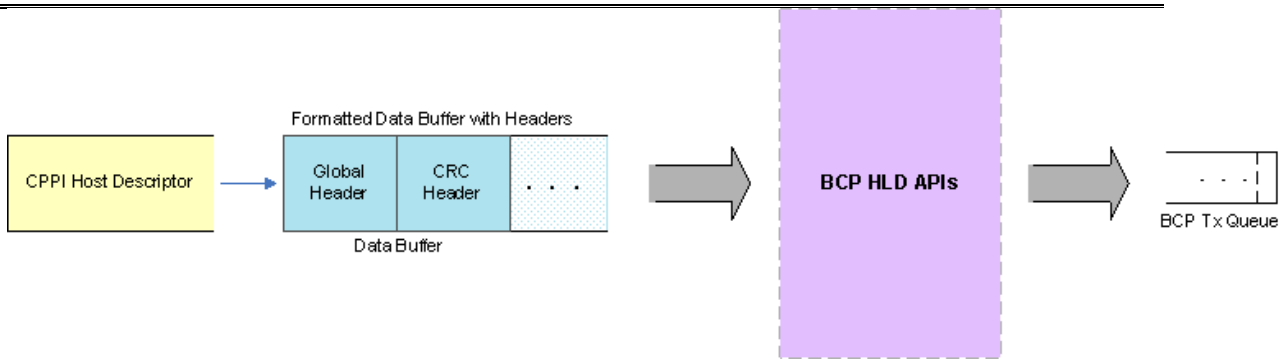

Figure 5. BCP send (local case)

Note:

- The application can choose to perform steps a, b, i.e., header preparation and addition apriori during control path and maintain these prepared header packets in its Tx FDQ. As and how packets arrive, then the application would have to just link the payload to these previously prepared BCP packets and just call *Bcp_send ()* API to send out the packet. This approach can save a lot of valuable cycles during data path. However, this approach assumes that the application knows the header parameters ahead of time (during control path) and these parameters don't vary from packet to packet.

- The BCP headers and payload can be all continuous memory in the same buffer in a descriptor or they can be split across two host descriptors, one containing just the BCP header and the second containing just the payload. Application can choose either of these approaches based on whether it is preparing BCP headers ahead of time and pre-stuffing descriptors with configuration or if it is building the entire packet during data path. Shown below are both possible options for data/header organization.

Figure 6. BCP header + payload all in one packet descriptor

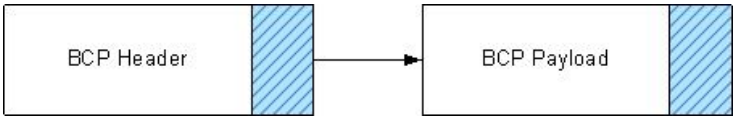

Figure 7. BCP header, payload split between 2 descriptors.

5.6.3 Receive Path

a. If the application wishes to poll on output from BCP, then can call the '*Bcp_rxGetNumPendingResults ()*' API from a process context to check on the destination

queue to see if a result is available. This API returns the number of results available for the Rx object to process.

b. The application can then retrieve the BCP output from the Rx queue using the '*Bcp_recv ()*' API. The API returns the pointer to BCP output descriptor, data buffer pointer, its length, the pointer to PS info pointer, PS info length from the result, flow Id on which the packet was received, and the source and destination tag information read from the descriptor.

c. If the application used accumulation interrupts instead of polling, then the application is responsible for ISR handling. The driver however provides a utility API *'Bcp_rxProcessDesc ()'* that the application can use with the descriptor it received from BCP to retrieve the output buffer, any PS info and source and destination tag information from it.

d. Finally once done processing the BCP output, the application could restore the descriptor using the '*Bcp_rxFreeRecvBuffer ()*' API.

- The application would have to provide the Rx object handle, CPPI descriptor or result handle obtained from '*Bcp_recv ()*' API.

- The '*Bcp_rxFreeRecvBuffer ()*' API returns the PS info buffer and result data buffer to the free result descriptor queue of the Rx object if driver is running in local mode, otherwise it calls the BCP transport layer's callback function *BcpTunnel_freeRecvBuffer ()* to free the descriptor. This API must be called by the application once it's done processing the result; otherwise the application could run out of free buffers and descriptors for the BCP to fill in the output eventually.

5.6.4 De-initialization

Any application that would like to de-initialize and close the driver would have to do so in the following order for proper cleanup:

1. Free any output buffers held with the application using the '*Bcp_rxFreeRecvBuffer ()*' API.

2. Close any Rx objects opened by the application using '*Bcp_rxClose ()*' API. This API frees up all the resources held by the object and closes the Rx queue and flow associated with this object. If driver is executing in remote mode, calls the transport layer *BcpTunnel_rxClose ()* callout.

3. Close any Tx objects the '*Bcp_txClose ()*' API. This API decrements the reference count on the Transmit queue used by the application, and when the reference count reaches zero (when all applications using the Tx queue close the Tx objects using this queue), the driver closes the transmit queue, and cleans up all associated resources. If driver is executing in remote mode, calls the transport layer *BcpTunnel_txClose ()* callout

4. Finally, close the BCP driver instance opened using the API '*Bcp_close ()*'.This API decrements the reference count on the BCP peripheral instance state, and frees up the application's driver handle.

5. Finally de-initialize the driver *using 'Bcp_deInit ()*' API. *When* the reference count on an BCP instance reaches zero, the driver closes the CPDMA associated with it and resets the BCP. Once this is done, no application in the system can use the BCP CPDMA unless opened again using '*Bcp_init ()*' API.

5.7 BCP OS Abstraction Layer APIs

The BCP driver defines an OS Abstraction Layer (OSAL) to log, obtain memory, and ensure synchronization to the BCP driver in an OS independent manner. This section covers all the OSAL APIs that need to be implemented by the System integrator in order to use BCP driver.

Following are the OSAL APIs that need to be ported by the System integrator for BCP:

5.7.1 Memory Management

1. Bcp_osalMalloc:

a. This API is called from the BCP driver to allocate memory for any of the driver's internal data structures and application handles such as Tx/Rx object handles.

b. The buffers allocated by this API MUST be global addresses and not core local addresses if the '*bGlobalAddress*' flag is set to 1.

c. By default, this API is implemented as follows in '*bcp_osal.h*':

```
#define    Bcp_osalMalloc        Osal_bcpMalloc
```

A sample implementation of the API '*Osal_bcpMalloc*' is provided in the '*example*' folder of the BCP code deliverable.

2. Bcp_osalFree:

a. This API is called from the BCP driver to free up the memory acquired using the '*Bcp_osalMalloc ()*' API.

b. The buffer pointer passed to this API is always a global address.

c. By default, this API is implemented as follows in '*bcp_osal.h*':

```
#define    Bcp_osalFree        Osal_bcpFree
```

A sample implementation of the API '*Osal_bcpFree*' is provided in the '*example*' folder of the BCP code deliverable.

5.7.2 Multi-core Synchronization

1. Bcp_osalMultiCoreCsEnter:

a. This API is called from the BCP driver to obtain a multi-core synchronization lock.

b. The driver expects that once this lock is obtained that no other process/thread on the current core or on any of the other cores can access the BCP driver APIs except for the process that has acquired it. The API should be implemented to reflect this.

c. This API is called from various control path APIs in BCP driver such as '*Bcp_init ()*', '*Bcp_open ()*' etc to ensure synchronous access to shared data structures.

d. By default, this API is implemented as follows in '*bcp_osal.h*':

```
#define    Bcp_osalMultiCoreCsEnter        Osal_bcpMultiCoreCsEnter
```

A sample implementation of the API '*Osal_bcpMultiCoreCsEnter'* is provided in the '*example*' folder of the BCP code deliverable.

2. Bcp_osalMultiCoreCsExit:

a. This API is called from the BCP driver to release a multi-core lock previously obtained using '*Bcp_osalMultiCoreCsEnter ()*' API.

b. This API is expected to reset the lock so that another process on any of the cores could acquire it to configure and use BCP library APIs.

c. By default, this API is implemented as follows in '*bcp_osal.h*':

```
#define    Bcp_osalMultiCoreCsExit        Osal_bcpMultiCoreCsExit
```

A sample implementation of the API '*Osal_bcpMultiCoreCsExit*' is provided in the '*example*' folder of the BCP code deliverable.

5.7.3 Interrupt Lock

1. Bcp_osalInterruptCsEnter:

a. This API is called from the BCP driver to obtain an interrupt lock, i.e., a lock that can protect the driver against context switches to an interrupt when its manipulating data structures shared between the application process and the interrupt service handler (ISR).

b. The driver expects that once this lock is obtained, no BCP interrupt occurs triggering the driver's ISR.

c. This API is currently not being used by driver.

d. By default, this API is implemented as follows in '*bcp_osal.h*':

```
#define    Bcp_osalInterruptCsEnter      Osal_bcpInterruptCsEnter
```

A sample implementation of the API '*Osal_bcpInterruptCsEnter'* is provided in the '*example*' folder of the BCP code deliverable.

2. Bcp_osalInterruptCsExit:

a. This API is called from the BCP driver to release an interrupt lock previously obtained using '*Bcp_osalInterruptCsEnter ()*' API.

b. This API is expected to enable the interrupts disabled earlier in '*Bcp_osalInterruptCsEnter ()*' API.

c. By default, this API is implemented as follows in '*bcp_osal.h*':

```
#define    Bcp_osalInterruptCsExit       Osal_bcpInterruptCsExit
```

A sample implementation of the API '*Osal_bcpInterruptCsExit*' is provided in the '*example*' folder of the BCP code deliverable.

5.7.4 Logging

1. Bcp_osalLog:

a. This API is called from the BCP driver to log useful debug information to the application.

b. This function can be mapped to an application desired print function to print the info on console or to a *syslog* kind of utility to stream it to a server.

c. By default, this API is implemented as follows in '*bcp_osal.h*':

```
#define    Bcp_osalLog          Osal_bcpLog
```

A sample implementation of the API '*Osal_bcpLog*' is provided in the '*example*' folder of the BCP code deliverable.

5.7.5 Process notification

1. Bcp_osalSemCreate:

a. This API is called from the BCP driver to create a software semaphore. This semaphore is used by the driver internally to block on a result for Rx object configured to use interrupts.

b. This function can be mapped to application's OS semaphore create function.

c. By default, this API is implemented as follows in '*bcp_osal.h*':

```
#define    Bcp_osalCreateSem          Osal_bcpCreateSem
```

A sample implementation of the API '*Osal_bcpCreateSem'* is provided in the '*example*' folder of the BCP code deliverable.

2. Bcp_osalSemDelete:

a. This API is called from the BCP driver to delete a semaphore obtained earlier using '*Bcp_osalSemCreate ()*' API.

b. This function can be mapped to application's OS semaphore delete function.

c. By default, this API is implemented as follows in '*bcp_osal.h*':

```
#define    Bcp_osalDeleteSem          Osal_bcpDeleteSem
```

A sample implementation of the API '*Osal_bcpDeleteSem'* is provided in the '*example*' folder of the BCP code deliverable.

3. Bcp_osalPendSem:

a. This API is called from the BCP driver to acquire a software semaphore obtained earlier using '*Bcp_osalSemCreate ()*' API.

b. This function can be mapped to application's OS semaphore acquire function.

c. By default, this API is implemented as follows in '*bcp_osal.h*':

```
#define    Bcp_osalPendSem          Osal_bcpPendSem
```

A sample implementation of the API '*Osal_bcpPendSem'* is provided in the '*example*' folder of the BCP code deliverable.

4. Bcp_osalPostSem:

a. This API is called from the BCP driver to release a software semaphore acquired earlier using '*Bcp_osalPendSem ()*' API.

b. This function can be mapped to application's OS semaphore release function.

c. By default, this API is implemented as follows in '*bcp_osal.h*':

```
#define    Bcp_osalPostSem          Osal_bcpPostSem
```

A sample implementation of the API '*Osal_bcpPostSem'* is provided in the '*example*' folder of the BCP code deliverable.

5.7.6 Cache Synchronization

1. Bcp_osalBeginMemAccess:

a. This API is called from the BCP driver before accessing (reading from) any multi-core shared data structures. Since the multi-core shared data structures could be placed in memory accessible across all cores, i.e., either in shared memory or external memory, this API is called from the driver to ensure that the read its performing on the memory block pointer it passed to this API is indeed read from the actual physical memory and not from a local cached copy on the core.

b. The application is expected to perform the necessary invalidate operation on the memory block passed to ensure that the driver reads correct data from physical memory.

c. Depending on the level of caches enabled in the system, this API can be mapped to the OS's corresponding Cache invalidate function.

d. By default, this API is implemented as follows in *'bcp_osal.h'*.

```
#define    Bcp_osalBeginMemAccess          Osal_bcpBeginMemAccess
```

A sample implementation of the API '*Osal_bcpBeginMemAccess'* is provided in the '*example*' folder of the BCP code deliverable.

2. Bcp_osalEndMemAccess:

a. This API is called from the BCP driver once it's done manipulating a multi-core shared data structure. Since the multi-core shared data structures could be placed in memory accessible across all cores, i.e., either in shared memory or external memory, this API is called from the driver to ensure that the write its performed on the memory block pointer it passed to this API is indeed updated in the actual physical memory too.

b. The application is expected to perform the necessary write-back operation on the memory block passed to ensure that the data updated by driver is reflected in the actual physical memory too.

c. Depending on the level of caches enabled in the system, this API can be mapped to the OS's corresponding Cache writeback function.

d. By default, this API is implemented as follows in *'bcp_osal.h'*.

```
#define    Bcp_osalEndMemAccess          Osal_bcpEndMemAccess
```

A sample implementation of the API '*Osal_bcpEndMemAccess'* is provided in the '*example*' folder of the BCP code deliverable.

3. Bcp_osalBeginDataBufMemAccess:

a. This API is called from the BCP driver before accessing (reading from) a data buffer on the data path. Since the data buffers could be allocated from shared memory or external memory, this API is called from the driver to ensure that the read its performing on the memory block pointer it passed to this API is indeed read from the actual physical memory and not from a local cached copy on the core.

b. The application is expected to perform the necessary invalidate operation on the memory block passed to ensure that the driver reads correct data from physical memory if the data buffer was allocated from the shared/external memory.

c. Depending on the level of caches enabled in the system, this API can be mapped to the OS's corresponding Cache invalidate function.

d. By default, this API is implemented as follows in *'bcp_osal.h'*.

```
#define    Bcp_osalBeginDataBufMemAccess
Osal_bcpBeginDataBufMemAccess
```

A sample implementation of the API '*Osal_bcpBegin*DataBuf*MemAccess'* is provided in the '*example*' folder of the BCP code deliverable.

4. Bcp_osalEndDataBufMemAccess:

a. This API is called from the BCP driver once it's done manipulating a data buffer on the Tx data path. The only data buffer manipulation done by the driver on Tx data path is to add a control header and any other configuration in the SOP of buffer. Since the data buffer could be allocated from shared/external memory, this API is called from the driver to ensure that the write it's performed on the data buffer memory block pointer it passed to this API is indeed updated in the actual physical memory too.

b. The application is expected to perform the necessary write-back operation on the memory block passed to ensure that the data updated by driver is reflected in the actual physical memory too. This is required only if the data buffers were indeed allocated from shared/external memory.

c. Depending on the level of caches enabled in the system, this API can be mapped to the OS's corresponding Cache writeback function.

d. By default, this API is implemented as follows in *'bcp_osal.h'*.

```
#define    Bcp_osalEndDataBufMemAccess
Osal_bcpEndDataBufMemAccess
```

A sample implementation of the API '*Osal_bcpEndDataBufMemAccess'* is provided in the '*example*' folder of the BCP code deliverable.

5. Bcp_osalBeginDescMemAccess:

a. This API is called from the driver before it makes a read access to CPPI descriptors on the Rx data path. If the descriptors were allocated in cacheable memory region, then the application would have to ensure that the cache is updated with the contents from actual physical memory. The application is expected to perform the necessary invalidate operation on the memory block

passed to ensure that the driver reads correct data from physical memory if the data buffer was allocated from the shared/external memory.

a. Depending on the level of caches enabled in the system, this API can be mapped to the OS's corresponding Cache invalidate function.

b. By default, this API is implemented as follows in *'bcp_osal.h'*.

```
#define    Bcp_osalBeginDescMemAccess      Osal_bcpBeginDescMemAccess
```

A sample implementation of the API '*Osal_bcpBeginDescMemAccess'* is provided in the '*example*' folder of the BCP code deliverable.

5.8 BCP Device Specific Layer APIs

The BCP driver provides a default device (SoC) specific initialization sequence in *'device'* folder as a sample. This is provided as a sample implementation only and must be customized by the system integrator for the SoC on which the application's being used. This API is not used by driver internally.

5.9 BCP Transport Layer APIs

To make BCP driver APIs uniform across "local"/"remote" BCP accesses, the BCP driver is designed such that it encapsulates all callouts required to send/receive data over SRIO using Type 9 messages in to a separate layer outside driver called "transport" layer. A sample implementation of this layer has been provided in the '*example'* folder. Description of the various call-back functions required for transport layer implementation have been defined and explained through the course of document during Driver initialization, Tx, Rx data paths.

6 Porting Guidelines

This section covers the porting considerations for BCP driver:

6.1 Data Types

The BCP driver by default uses TI's XDC/RTSC data types and is captured in the '*bcp_types.h*' header file. The system integrator must port this header file to use data type definitions that match his system.

6.2 OS AL APIs

The OS AL APIs used by BCP driver by default are defined in the '*bcp_osal.h*' header file and discussed in Section 5.7 in detail.

The BCP driver user / system integrator can take either one of the below mentioned approaches in porting the BCP OS AL to suit his native OS:

1. Use Pre-built BCP libraries:

a. To use pre-built BCP libraries, the BCP driver integrator would have to provide a suitable implementation in his native OS for all the default mapped OS AL APIs, i.e., '*Osal_bcpXxx ()*' defined in '*bcp_osal.h*'.

b. The application can then link the appropriate version of BCP pre-built library (choose between Big Endian and Little Endian to match his platform) and can use the BCP library APIs successfully.

c. A sample implementation of this approach has been provided in the '*example*' directory of the BCP deliverable.

d. This approach is better suited if the customer would like to use the TI provided pre-built BCP libraries right out of the box.

e. The demerit of this approach would be that, there is one extra level of indirection in getting to the actual OS AL API implementation from the BCP driver APIs.

For example:

BCP driver APIs use Bcp_osalMalloc internally, which is in turn mapped at pre-compilation to → Osal_bcpMalloc that contains actual implementation of the API in native OS.

2. Rebuild BCP from sources:

a. To do so, the system integrator must make a copy of the '*bcp_osal.h*' header file and modify it to directly map the '*Bcp_osalXxx*' APIs used by the BCP driver APIs to his inline native OS calls.

For Example:

The example shows a sample mapping assuming the OS used is BIOS. Here the Memory allocation API used by BCP driver, i.e., '*Bcp_osalMalloc'* is mapped to the corresponding memory allocation API from BIOS 6 '*Memory_alloc ()*'.

```
#defineBcp_osalMalloc        Memory_alloc
```

b. Once all the BCP OSAL APIs discussed in Section 5.7 are implemented as shown above using native OS calls, the integrator can include his new header file with his OS definitions and recompile the BCP sources.

c. The '*test*' directory provides a sample implementation of this approach using BIOS.

d. This approach eliminates the extra OSAL API indirection that was there in the previous approach and hence is better optimized for performance.

e. However, the downside of this approach is that the customer would have to rebuild the BCP library from sources.

6.3 Configuration Options

The BCP driver exposes a compile time configuration option "BCP_DRV_DEBUG" to control parameter validations on all fast path APIs.

The driver libraries by default are built with this option turned OFF, i.e., no parameter validations built into the fast path HLD and LLD APIs. If the library user would like to build a debug version of the library with parameter validations, he could include the driver sources into his application project, include this option in compiler's build definitions and rebuild his project. Alternatively, the customer can modify the library build files to include this option and re-build the library from command line to enable parameter validations in driver for the application.

7 Future Extensions

None