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

TCP3D Driver

Software Design Specification (SDS)

Revision B

Revision Record	
Document Title: **Software Design Specification**	
Revision	Description of Change
A	1. Initial Driver using Swi & List Modules – Code Drop Eng1 2. Driver re-designed using pseudo PaRAM approach – Code Drop Eng2 3. Packaged with version tracking, tested with BIOS CPINTC – Code Drop 1.0.0.0 4. Second TCP3D instance support added, not tested – Code Drop 1.0.0.1 5. Wrap-around mode and Split mode supported, Export name compliance added – Code Drop 1.0.0.2 6. EDMA channels optimization – Code Drop 1.0.0.3 eng 7. Wrap-around mode working after channel optimization, Notification Event support added – Code Drop 1.0.0.4 8. Run-time selection of optional outputs, Enqueue API optimization, New deinit API, Race condition fixes added – Code Drop 1.0.0.5 9. Instance parameters clean-up, few parameters renamed, API description updates – Code Drop 1.0.0.6. 10. Driver types changed to C99, support for mini package, minor bug fixes to the test code – Code Drop 1.0.0.7. 11. Added cache coherency hooks to example and test code. Changed the memory allocation to align with cache line size. – Code Drop 1.0.0.10. 12. Updated the enqueue and start API flow diagrams to reflect the latest changes and description at various places are updated. Also, update the test function flow diagrams according to the new testing sequences. – Code drop 1.0.0.12
B	1. Updated document to reflect the addition of next code block dummy and next code block dummy notification PaRAM entries to avoid a race condition for the current block when updating OPT and LINK fields.

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

TABLE OF CONTENTS

1 Scope

This document describes the functionality, architecture, and operation of the TCP3D Driver.

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	<Name & Version> PRD		Product Requirements
2	Nyquist/Shannon SAS	Version 0.5	Software Architecture Specification
3	TCP3D Users Guide	Version 1.0.0.0	User guide for 3rd Generation Turbo Coprocessor Peripheral for Decoding

Table 1. Referenced Materials

3 Definitions

Acronym	Description
API	Application Programming Interface
TCP3D	3rd Generation Turbo-Decoder Coprocessor Peripheral
DSP	Digital Signal Processor
OSAL	OS Abstraction Layer
LTE	Long Term Evolution
WiMax	Worldwide Interoperability for Microwave Access

Table 2. Definitions

4 Overview

The third generation Turbo Coprocessor (TCP3) is a programmable peripheral for decoding turbo codes used in 3G wireless systems like 3GPP, LTE, WiMax in an iterative manner. This peripheral, with two Maximum A-posteriori Probability (MAP) decoders, is integrated into Texas Instruments' DSP devices for use in wireless base stations and/or in user equipments.

Input to the TCP3 decoder (TCP3D) is channel soft decisions for systematic and parity bits (LLRs) and the outputs are hard decisions. Some applications may require the soft bit information for all the bits (systematic and parity) at the end of turbo decoding process. This

information can be used as a feedback between turbo decoder and equalization block in the receiver. TCP3D can output these bits as programmed in the input configuration parameter.

The TCP3D co-processor consists of the following blocks:

1. **Control registers:**

   Useful for specifying the general use of the decoder to tell the decoder about the technology (LTE, WiMax or 3GPP) and system configuration for which it will be used.

2. **Input configuration registers (INCFG):**

   Useful for specifying the configuration to be applied for performing the Decoding on input code block data, specifying the input and output formats, etc.

3. **Input data memory (LLR, INTLVR):**

   Used for transferring the channel soft bits (LLRs) and optional interleaver from the DSP for decoding.

4. **Output data memory (HD, SD):**

   Used for storing the hard decisions (HD) from the decoder engine and optional soft bits (SD).

5. **Output Status registers (STS):**

   Status of decoding will be stored in these registers and could be read by application if needed.

6. **Decoder Engine:**

   Performs the actual decoding on the LLRs and an event is generated at the end to notify the DSP.

This document assumes that the reader has familiarized him/herself with the TCP3D User's Guide and understands the programming of EDMA3 for data transfers.

The data transfer between DSP and TCP3D is best performed via EDMA transfers. The TCP3D has two synchronization events, REVT0 and REVT1 for Ping and Pong engines, connected to EDMA channel controller as shown in Figure 1. TCP3D requires a set of input configuration parameters (packed 15 registers) describing the code block to be decoded.

The basic decoding process requires minimum of two input transfers and one output transfer. These transfers are best done by using EDMA PaRAMs. There are optional input and output transfers that could be done depending on the system requirement/configuration.

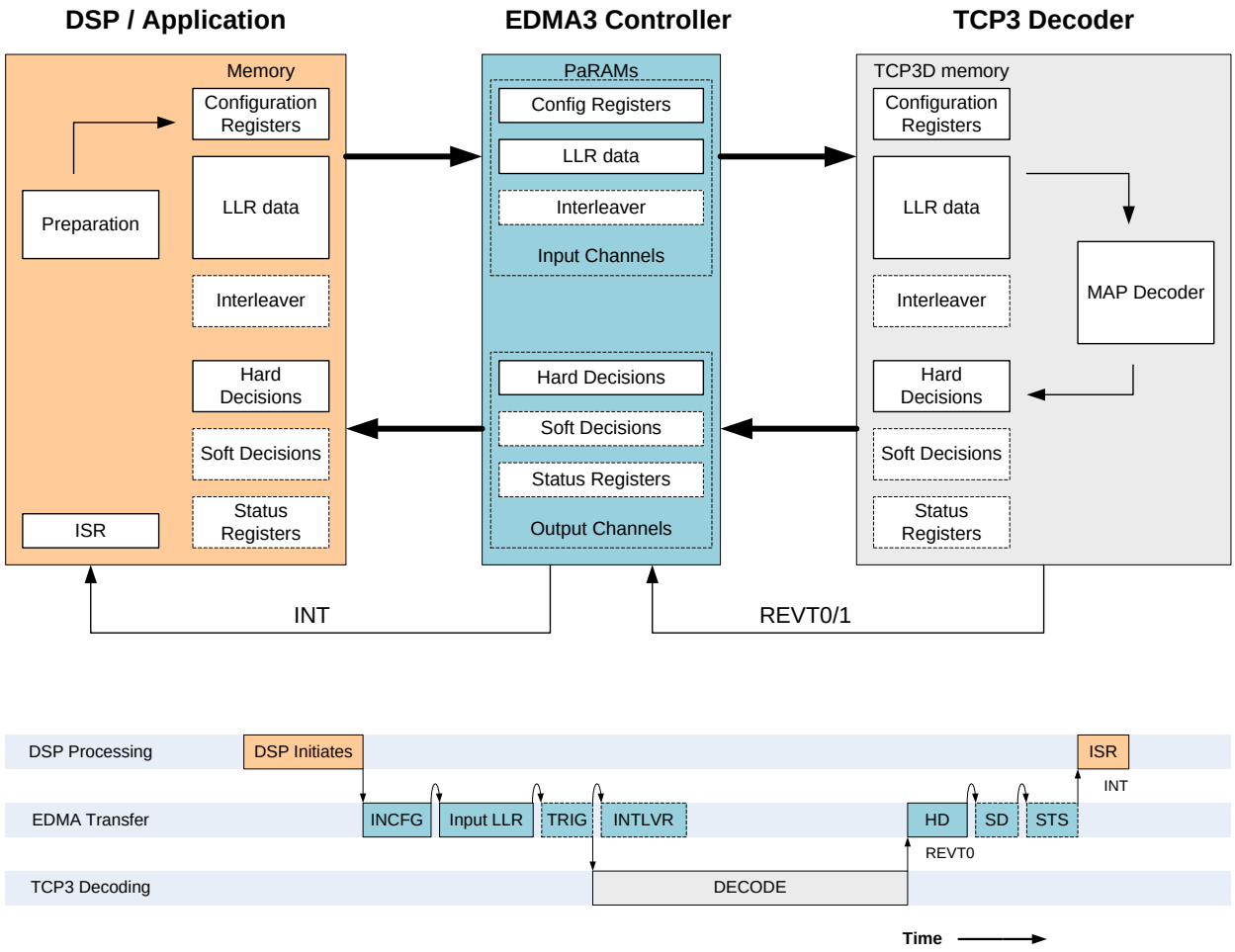

Figure 1 - Basic Decoding process for single code block

The decoding processing can be described as below and is repeated for every code block.

1. **inputs transfer**
   - DSP/Application initiates the process by manually triggering the EDMA input channel to start the input transfers via EDMA.
   - The EDMA transfers the input configuration registers to TCP3D memory first and then links in the PaRAM entry which transfers the input data (systematic and parity bit streams) for decoding to TCP3D memory.
   - If the optional inputs are used in the system they will be chained to LLR PaRAM.
2. **triggering the decoder**
   - If the mode control register bit for auto trigger is enabled, then the decoding starts automatically when the expected number of input bytes transferred into TCP3D configuration registers and then into the LLR memory successfully. This is the most preferred method for performance reasons and ease of use.

- If the auto trigger is disabled, TCP3D requires that after input data transfers, the trigger register must be written to start the decoding. This option was provided mostly for legacy reasons.

3. outputs transfer

- After the decoding is complete, TCP3D issues the REVT signal to EDMA controller. This could be used to trigger the output channel to initiate the decoded output data (packed bits) transfer from TCP3D memory to DSP memory.
- The optional outputs are transferred depending on the configuration bits set in the input configuration register by the DSP/Application.
- After the transfer is completed, the EDMA controller can be set to generate an interrupt for DSP to signal the end of the decoding process for the code block.
- The next code block input data to TCP3D memory cannot be transferred until the outputs for the previous code block were read from the TCP3D output memory. It calculates the number of bytes must be read from all the output memories from the configuration register values and waits until all of them are read to switch its internal state machine.

The above illustration gives the user the basic understanding of the data flow; however for all practical purposes, it may not yield the throughput requirements in a given system. To meet such requirements, the EDMA PaRAM linking and chaining features must be used and also efficient use the following TCP3 decoder features is needed:

- Two memories (PING and PONG) could be used to pipeline the input data while decoder engine is working on other memory.
- Depending on the mode (double buffer mode or split mode) the REVTs would be issued by the TCP3D different times and actions to be taken based on that would be different. Also, the complexity of programming the EDMA varies based on the mode.

The challenge lies in preparing the input configuration registers as fast as possible and get the input code block data to the TCP3D memory as soon as possible so the decoder engine could be engaged fully. The code blocks arrival for decoding may be in bursts and random depending on the system. To support any arrival pattern, the driver needs to support some kind of queuing mechanism so that the decoding can continue without DSP intervention and should have a programmable notification mechanism to wake-up DSP when a set of code blocks are decoded. This method helps if the TCP3D is slower than the input data ready.

This document is not intended to cover the details about the TCP3 decoder features and their usage such as modes and how to use them, details about various registers, etc. It is highly recommended to the user to refer to the user guide [3] for details.

5 Design

5.1 Goals

TCP3D driver design goals are listed below:

1. Driver runs in continuous mode, unaware of the frame boundary.
2. Driver involves minimum CPU intervention, minimum context switching.
3. Driver receives code block decoding requests at random times or in bursts.
4. Minimum latency completion notification and programmable.
5. Support multiple instances to use each peripheral instance independently.
6. Driver supports multi core input and multi core output destinations.

5.2 TCP3D Driver

This section describes one TCP3D driver design approach that is selected as a candidate that best satisfies the required design goals.

5.2.1 Decoding process in the uplink bit processing chain

The position of the TCP3D driver in the LTE uplink bit processing chain is illustrated in Figure 2 as an example.

The pre-processing involves more intensive calculations and it includes soft slicing, de-scrambling, de-multiplexing, code block segmentation and rate de-matching. At the end of rate de-matching, three bit streams (one systematic and two parity bits) will be ready for a code block along with the tail bits. These tail bits are used to calculate the initial beta state values which go into the packed input configuration registers and are transferred first to the TCP3D before transferring the data bit streams for decoding.

The post-processing mainly consists of the code block concatenation and CRC checking. TCP3D can internally calculate CRC (type B) for the code blocks. This would be sufficient for the transport blocks with one code block and if the transport block consisting of more than one code block, the transport block CRC (type A) has to be calculated by DSP.

Staging of pre-processing is the key to the system design. If the pre-processing is faster than TCP3 decoder, the code blocks have to be parked before the TCP3D is free. In case of LTE, availability of PING and PONG memories helps in pipelining one code block while the decoder engine is busy. However, this would not be sufficient in a typical application use, so the driver should be capable of queuing the code blocks in a non-blocking manner and should be able to work on the queue without DSP intervention. Current driver uses a method for staging the code blocks as soon as they are available into an input list while the decoder is busy working on previous code blocks. This can be viewed as an input queue to the TCP3 decoder and driver takes the code blocks from this queue and sends for decoding as soon as the decoder is free. Details of this method are discussed in the following chapters.

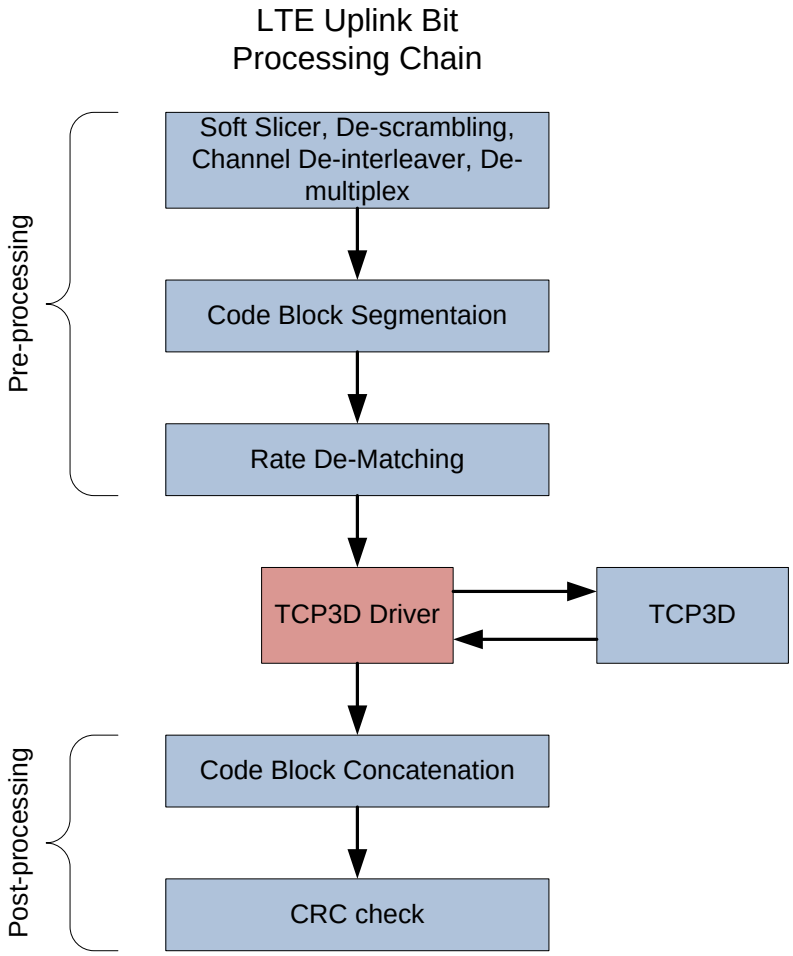

Figure 2 - TCP3D driver in the LTE uplink bit processing chain

Figure 3 shows the TCP3D driver usage example showing the pre and post processing task loads distributed among the multiple cores. Since the current TCP3D driver is not supporting the muti-core interface; the decoding requests are always sent from the master-core while the requests from other-cores can be routed through the use of queues.

The pre-processing task can be started when the soft data is ready from symbol processing task (not shown in the diagram) which could be done on different threads or on different cores for load balancing. At the end of rate de-matching, the code blocks are ready to be sent to TCP3D for decoding. The pre-processing task on the master core loads the code blocks prepared either on the same core or from the other core into the driver input list by using the Enqueue API as shown in the diagram.

For each code block, the TCP3D requires the input configuration registers to be loaded with the right values. These values range from specifying the input and output formats, flags to be set if any optional inputs or outputs are present to be transferred, to filling the beta state values. Refer to the user guide for detailed list of parameters. Enqueue function prepares a set of required EDMA PaRAMs for each code block and stored in the input list memory. Preparing all the 15

registers at run-time impacts the throughput performance and the current driver design defines some optimization tricks in the later chapters and the supporting utility functions for preparing the registers are also provided to do the minimum preparation at run-time.

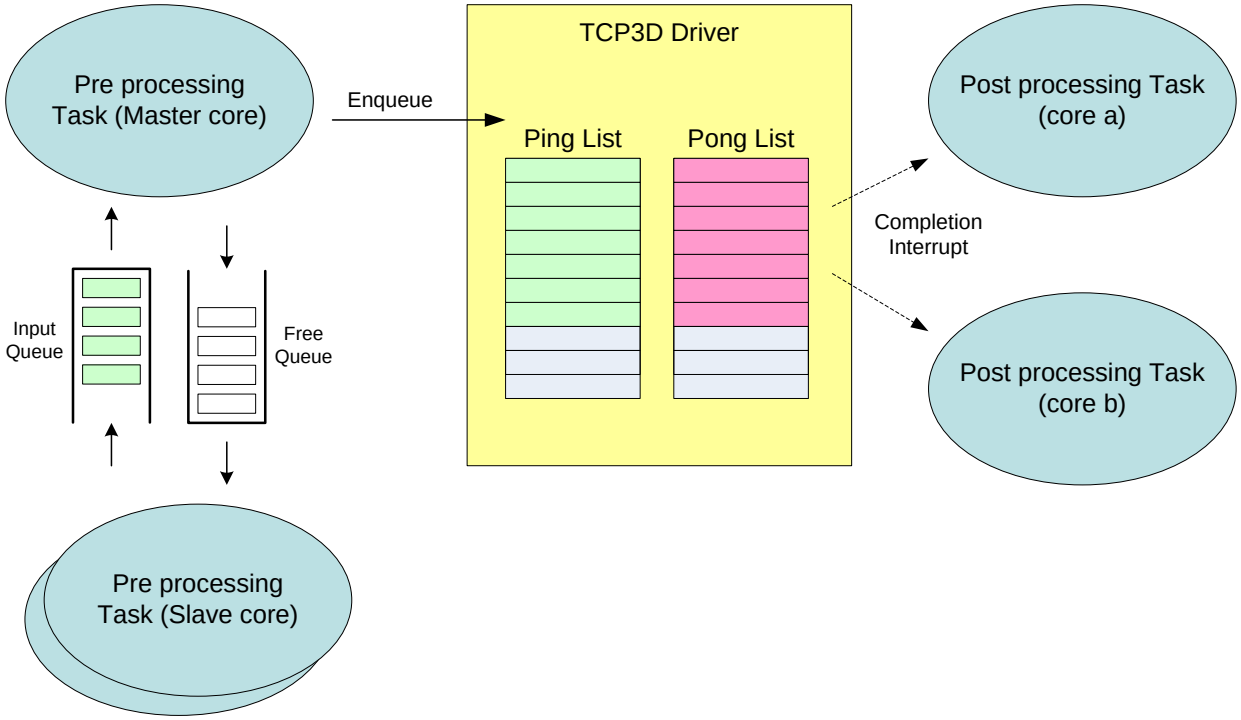

Figure 3 - Multi-task/Multi-core usage example for TCP3D driver

When the Driver is started, it transfers one code block data to TCP3D memory using one such EDMA PaRAM set and when the decoding is complete the output(s) are read from TCP3D memory using EDMA and the next block data is transferred. If the optional notification flag is set for any given code block, the event configured during init time will be generated by the driver. Application can use this event to trigger the post-processing tasks.

5.2.2 Pseudo PaRAM Input List Approach

For each code block, one set of EDMA PaRAMs are required for doing input and output transfers to and from the TCP3 decoder. They are prepared and stored in the input list first and when the decoder is free to take one, these PaRAMs are used for execution. There are two possibilities for where to keep these prepared PaRAM sets:

1. one approach is to store them in the EDMA PaRAM memory directly
2. another approach is to keep them in the DSP layer-2 (L2) memory and copy them to EDMA PaRAM memory during run-time

The second approach was chosen mainly for the following reasons while it requires some additional EDMA channels for run-time operations. The name pseudo PaRAM is used to represent that the list is used for placing the actual EDMA PaRAMs in another memory area.

- it costs less system resources - less EDMA PaRAMs

- gives more freedom to choose the input list size, not restricted by EDMA PaRAMs capacity but limited by DSP memory capacity
- run-time cycles for preparing and linking the PaRAMs will be much lesser compared to the other approach

Figure 4 gives the high-level flow diagram for the approach using two lists for PING & PONG separately. As you can see in the diagram, there are few optional features, represented as dotted boxes, supported by the driver for selectively reading the outputs and also for generating the notification event which could be programmed for each code block with Enqueue API.

The driver is designed to provide the user ease of use and less programming hassle to build the input queue using the EDMA PaRAMs through linking and chaining. The main features of this approach are listed below:

- TCP3D driver uses the input list as two pseudo PaRAM lists (as interleaved) for using with the PING and PONG memories separately.
- Maximum number of code blocks that could be queued is programmable during driver initialization (shared between the two lists).
- Driver builds the input list independent of the decoder running or not.
- Both the reserved EDMA channels for PING & PONG are used for transferring the code block data to/from the TCP3D memories. Allows the input transfers to be done while the decoder is working on the other memory.
- EDMA chaining feature is used to keep the TCP3 decoder engaged continuously until the end of input list.
- Two more EDMA channels are used for transferring the pseudo PaRAM sets into the EDMA PaRAM memory and triggering of these channels are done using the transfer completion chaining feature. This allows the application to fill the input list independently of the execution.
- Completion notification through the system event can be requested by the application per code block basis. Application can use this feature to start the post-processing upon decoding a specific code block and also to track the last block decoding.
- Driver provides a means to support the continuous decoding even if the input list is full. This feature is referred as wrap-around mode for driver.
- It has also the optional reset function to clear the input list and start using from the beginning of the list. This is useful for frame-based applications like LTE & WIMAX.
- Driver runs on a single core. Requests from other cores can be received via some queue mechanism and passed to the driver.
- Driver allows the application to choose when to start executing from the list. Once started, the driver keeps running as long as the list is not-empty and gets restarted from Enqueue if paused.

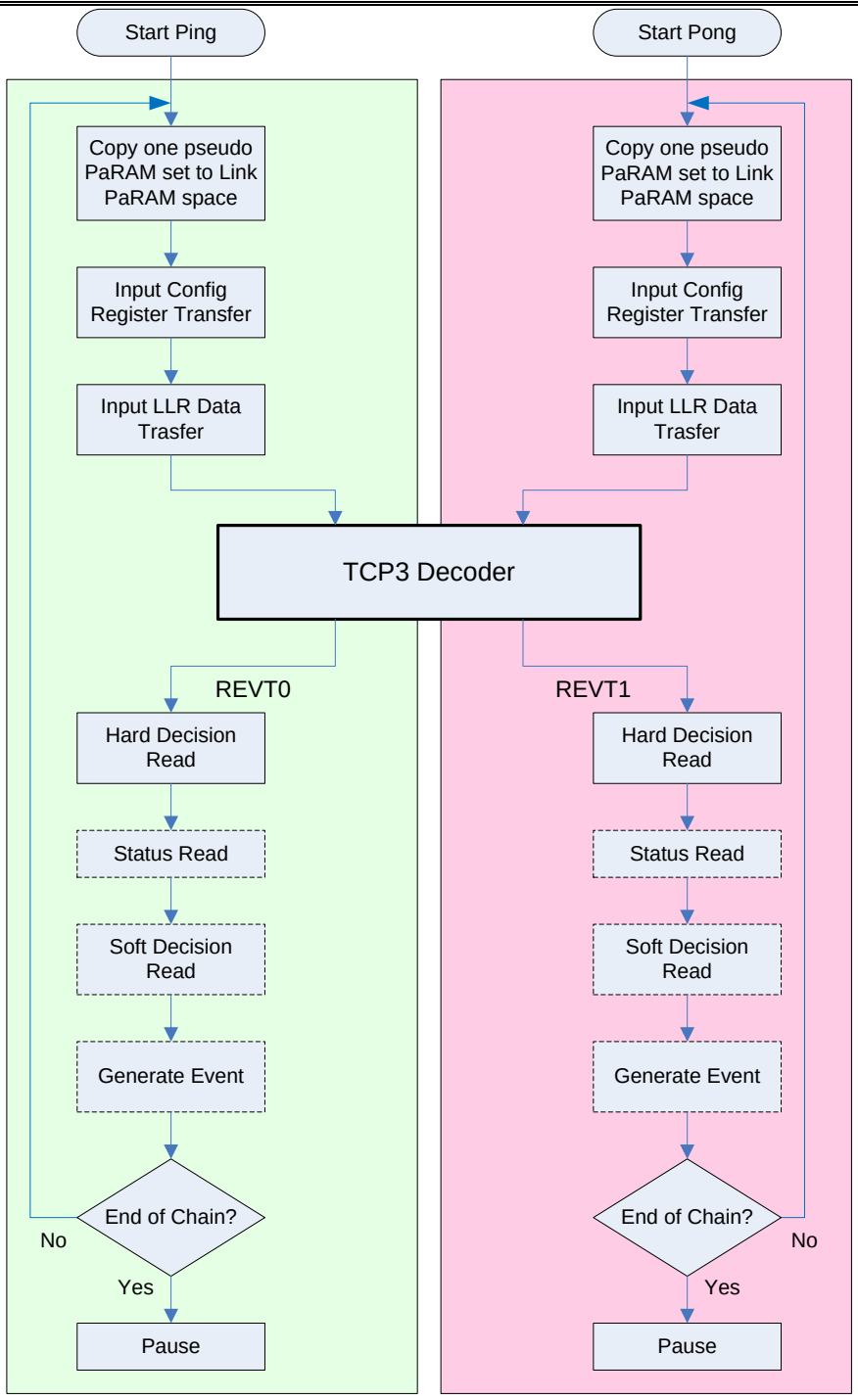

Figure 4 - High-level Flow diagram using the Input List approach

5.2.3 Driver States

TCP3D driver has very simple state machine as shown in Figure 5. When the driver is first started, it will transition from INIT to RUNNING after starting (one of the lists) and it will continue to be in this state as long as the chain to the next code block exists (in both the input

lists). If the chain to the next code block (in any one of the lists) is not available, the driver reaches the PAUSE state. There are few possibilities for this state and driver needs to detect the same and restart if needed.

1. There are no more code blocks in the list for execution (end of list).
2. Next code block was not available when the current code block was taken for execution from the list. But added to the list later.

If the second case, the driver needs to be started again, restart, to process the remaining blocks from the list. It can be done using start API which brings the driver state to RUNNING. This condition checking is done from the enqueue function if there are more blocks being enqueued or else the application has to do the checking while waiting for the last code block to be decoded.

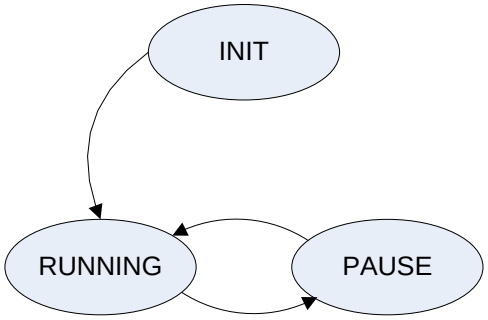

Figure 5 - Driver state diagram

5.2.4 EDMA Channel & PaRAM entry usage

Figure 7 illustrates the details of the various PaRAM entries used in the driver flow. The red arrows represent the trigger path, while blue arrows represent the link path.

5.2.4.1 Data PaRAM Entries

Each code block is handled with five pseudo PaRAM entries which are copied to the reserved EDMA PaRAM area for code block decoding using control channel, i.e. Ch L2P:

1) “Cfg_n” – This entry transfers the TCP3D input configuration registers (n: CB index). It is executed on the REVT channel, i.e. Ch Revt.
2) “Llr_n” – This entry transfer the three input data streams from DSP memory to TCP3D memory. It is executed on the Ch Revt triggered through chaining from “Cfg_n”entry.
3) “Hd_n” – This entry transfers the output decoded bits from TCP3D memory to DSP memory. It is executed on the Ch Revt triggered by the REVT event generated by TCP3D upon decoding the code block.
4) “Sts_n” – This entry transfers the three status register values from TCP3D memory to DSP memory. It is executed on the Ch Revt triggered through chaining from “Hd_n”entry.
5) “Sd_n” – This entry transfers the output soft decision values from TCP3D memory to DSP memory. It is executed on the Ch Revt triggered through chaining from “Sts_n”entry.

5.2.4.2 Control PaRAM entries

Several EDMA PaRAM entries and one physical EDMA channel, i.e. Ch L2P, is dedicated for each path execution and control operation as shown in Figure 7. The control PaRAM entries are listed below:

1) “Dummy” – Each code block execution starts with this entry. As the name indicates, it does nothing but links in the first link PaRAM of the code block, i.e. "Cfg_n", and triggers the Ch Revt (see Figure 7). This is the PaRAM entry sitting on the Ch Revt after driver initialization.
2) “Pause” – It copies the “stop” flag to the driver stop variable and also copies the "PAUSE" state value to the driver state variable, links in the “Dummy” entry. This halts the driver from executing in that path until started again. It is set to generate completion interrupt all of the time and passing the interrupt to the DSP must be controlled by using the TPCC_IER/TPCC_IERH bits to enable or disable as needed. It is disabled during initialization and is controlled at run-time using driver control function.
3) “Notify” – It copies the notification event number from the driver instance to the CP_INTC0 set index register thus generating a system interrupt to CP_INTC0. This entry links and triggers the "Pause" entry.
4) “NotifyD” – The operation of this entry is same as "Notify" except for the linking. It links in the “Dummy” entry and triggers to start the execution of the next code block.
5) “NextCBDum_n” – This entry is used if there is another code block to decode in the list, and links to the next code block’s “Dummy” entry. It chains to Ch L2P on completion. It is executed on the Ch Revt triggered through chaining from the “Sd_n” entry.
6) “NextCBNtfd_n” – This entry is used if there is another code block to decode in the list, and notification is required before decoding the next code block. This entry links to the “NotifyD” entry. It is executed on the Ch Revt triggered through chaining from the “Sd_n” entry.
7) “WrapCnt” – It copies the path count value to a driver variable that is used in determining the output execution status which is needed in wrap-around case for proper functionality. By default this links in the "Pause" entry and triggers the corresponding Ch Revt. If the link to the next block is established it links in the "Dummy" entry and triggers the Ch L2P to start the next code block.
8) “L2P” – It copies one set of PaRAM entries from the L2 memory (pseudo PaRAM buffer) to the reserved link PaRAM area and triggers the corresponding "REVT" channel for execution. When the end of list is reached, links in the "L2PReload" entry. This is the PaRAM entry sitting on the Ch L2P after driver initialization. In the wrap-around mode, this PaRAM could be used for generating interrupts as needed by application to get notified when an entry is taken out of input list. This could be useful in the LIST FULL case.
9) “L2PReload” – Copy of the Ch L2P PaRAM entry "L2P" used for wrap-around case.

Pseudo PaRAM entries 5) and 0 above are present in the list, though only one of them will be executed, as determined by the notification indication request in the Enqueue API call. Figure 6 depicts the use of “NextCBDum” and “NextCBNftd”. Depending on the need for notification from the previous code block, the LINK address for that code block is updated to link to either “NextCBDum” or “NextCBNftd”. Keeping the linking channel the same for either PaRAM

means that the last code block PaRAM doesn't require an update to its transfer options, therefore an 1 word update to the LINK field can be made to the PaRAM for the next code block enqueued, without fear of running into a race condition of needing to update 2 words.

In the case where no further code blocks are present in the list, the "Notify" PaRAM entry can be optionally enabled.

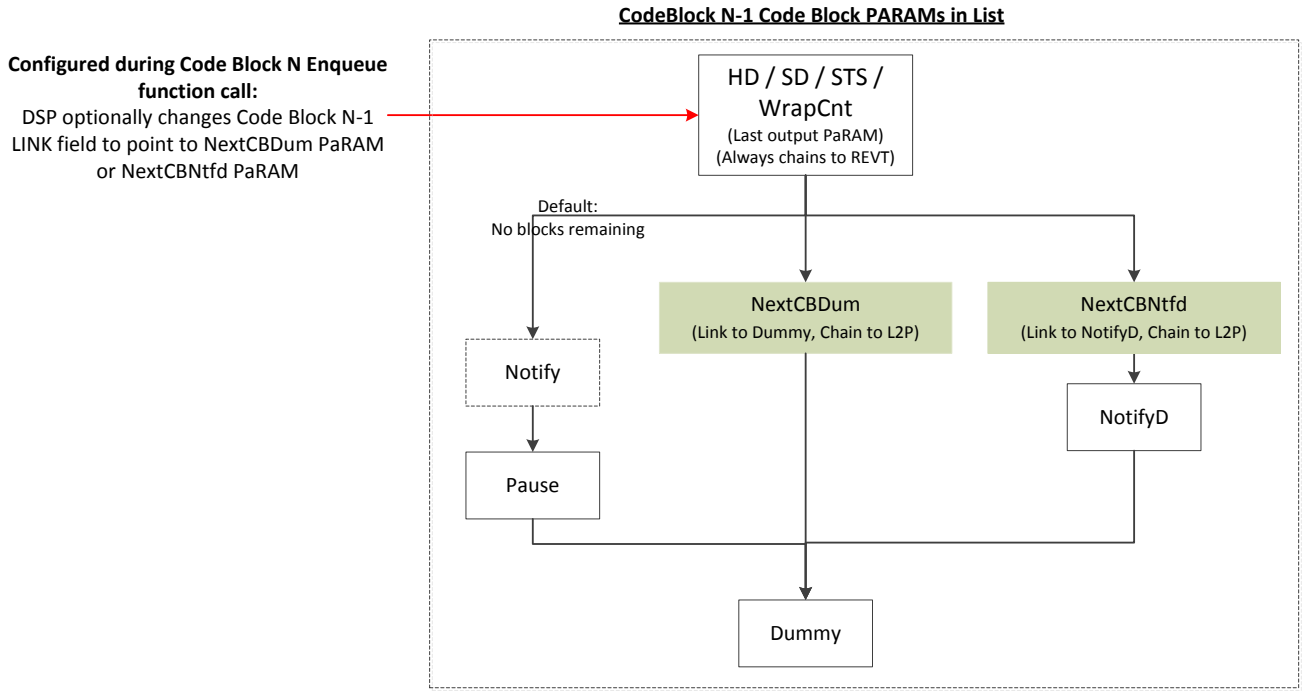

Figure 6 - Next Code Block Handling

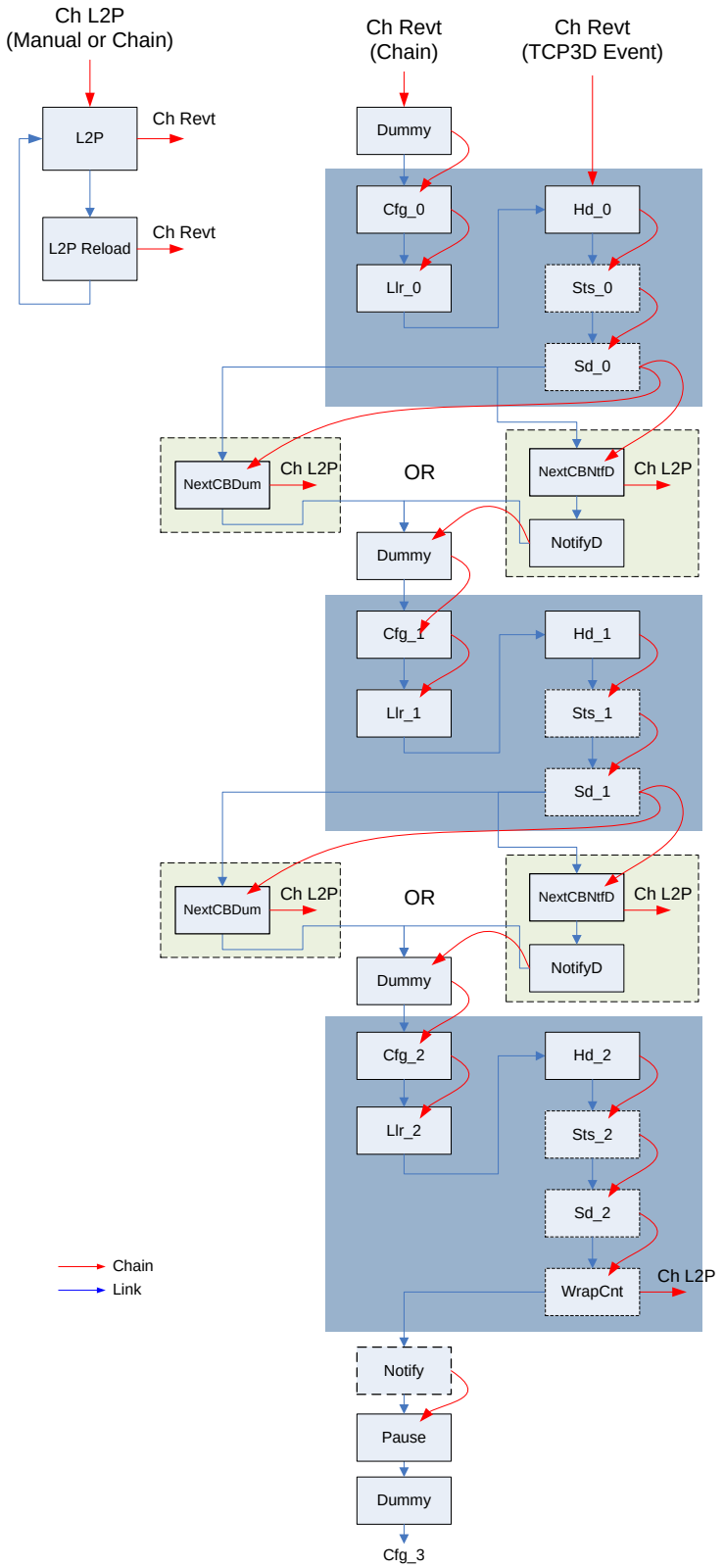

Figure 7 - Linking and triggering details of the driver Channels and PaRAM entries

5.2.5 Resource Requirements

5.2.5.1 EDMA Channels requirements

The EDMA channels required by the driver are summarized in Table 3. The number of required physical and linked (PaRAMs) channels for using different TCP3D instances is listed separately.

Resource Description	TCP3D_A	TCP3D_B
Revt0 Channel	0	34
Revt1 Channel	1	35
L2P Channel 0	Any Channel	Any Channel
L2P Channel 1	Any Channel	Any Channel
# of linked Channels (PaRAMs)	2 * (5 + 6 + 2)	2 * (5 + 6 + 2)

Table 3 - EDMA Channel requirements

5.2.5.2 Memory requirements

The TCP3D driver memory requirements are summarized in Table 4. The maximum number of code blocks is denoted as M.

Resource Description	TCP3D_A or TCP3D_B
Instance Memory	644
Pseudo PaRAM Buffer Size (Bytes)	32 * 5 * M

Table 4 - Memory requirements

The TCP3D driver instance memory includes

a. Driver instance related data parameters
b. Variables used for run-time control operations
c. Array to keep the EDMA specific data – Channels, PaRAM physical addressed, Control register pointers
d. Debug flags

5.2.6 Input Configuration registers preparation

TCP3D requires all the 15 input configuration registers to be transferred for each code block before transferring the input data (LLR) streams to its memory. Preparing all these registers with all the bit-fields at run-time would be expensive and some of registers does not change for a given system configuration, some of them are known upfront at the frame boundary, etc. Figure 8 gives details on each register and its dependency for LTE or WIMAX modes and Figure 9 gives details for 3GPP mode.

For all modes, the registers IC2, IC3, IC8-IC11 are typically fixed at the system initialization time and so these registers are not required to be prepared during run-time. For 3GPP modes, IC12, IC13, IC14 are not at all used, so these registers become don't cares. These registers are represented with mesh boxes in the figures.

We are left with few registers that have to be prepared after initialization. Another thing to note here is that all of these registers depend on the block size, so if you know the block sizes upfront these registers can be prepared in advance.

- IC0, IC1, IC4-IC7, IC12-IC14 for LTE and WIMAX modes
- IC0, IC1, IC4-IC7 for 3GPP modes

Last thing to note is that the registers IC4-IC7 depend on the actual input data (tail bits) for each code block, so those registers can only be prepared after the input data is available and beta state values are computed as described in the user guide [3].

TCP3D driver has bunch of utility functions for preparing these registers for different combinations as described here depending on user needs ranging from preparing all registers, prepare only fixed registers, prepare block size dependent, prepare beta state registers, etc.

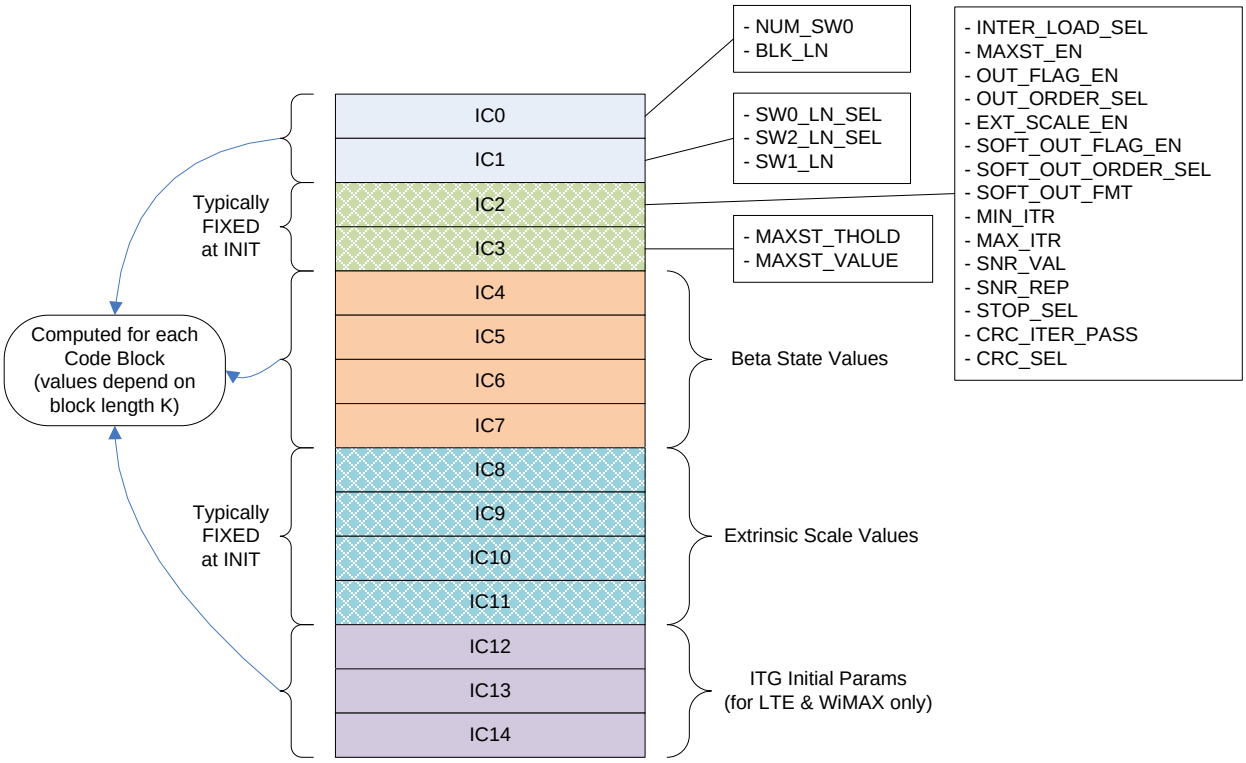

Figure 8 - Input Configuration Registers - LTE/WIMAX mode

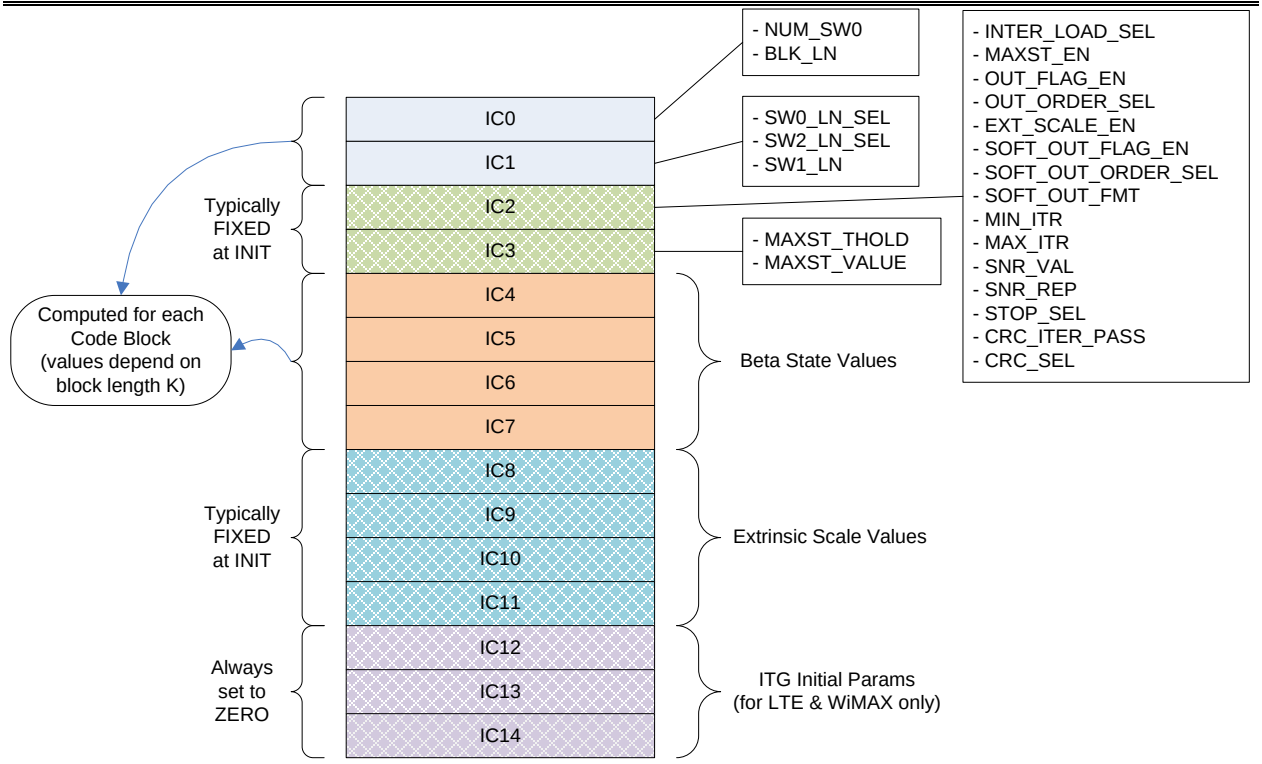

Figure 9 - Input Configuration Registers - 3GPP mode

5.2.7 Notification Mechanism

The notification mechanism used in the driver design is to generate a system event using an EDMA transaction. Application provides the system event number that goes to the interrupt controller (CP_INTC0) during the initialization time and driver stores the same into the instance memory.

The interrupt controller has a feature where writing the event number word into the register STATUS_SET_INDEX_REG causes a system event. Driver relies on this fact and used a control PaRAM to write the stored event number into this register as programmed during enqueue function.

It is the responsibility of the application to handle the registration, de-registration of an ISR for the event with the operating system and take necessary actions. Figure 10 gives an example how the event is caused by Channel 0 or Channel 1 and possible connections up to GEM cores for a chosen event number 7.

We use separate notification PaRAMs for each path. There is a possibility in the Split Mode, where both the PING and PONG engines run independently, both the PaRAMs might issue writes to the INTC0 register at close proximity in time. In such case, one event could be lost and so it is recommended to check the output memories to determine if the expected block is decoded or not when waiting for notification events.

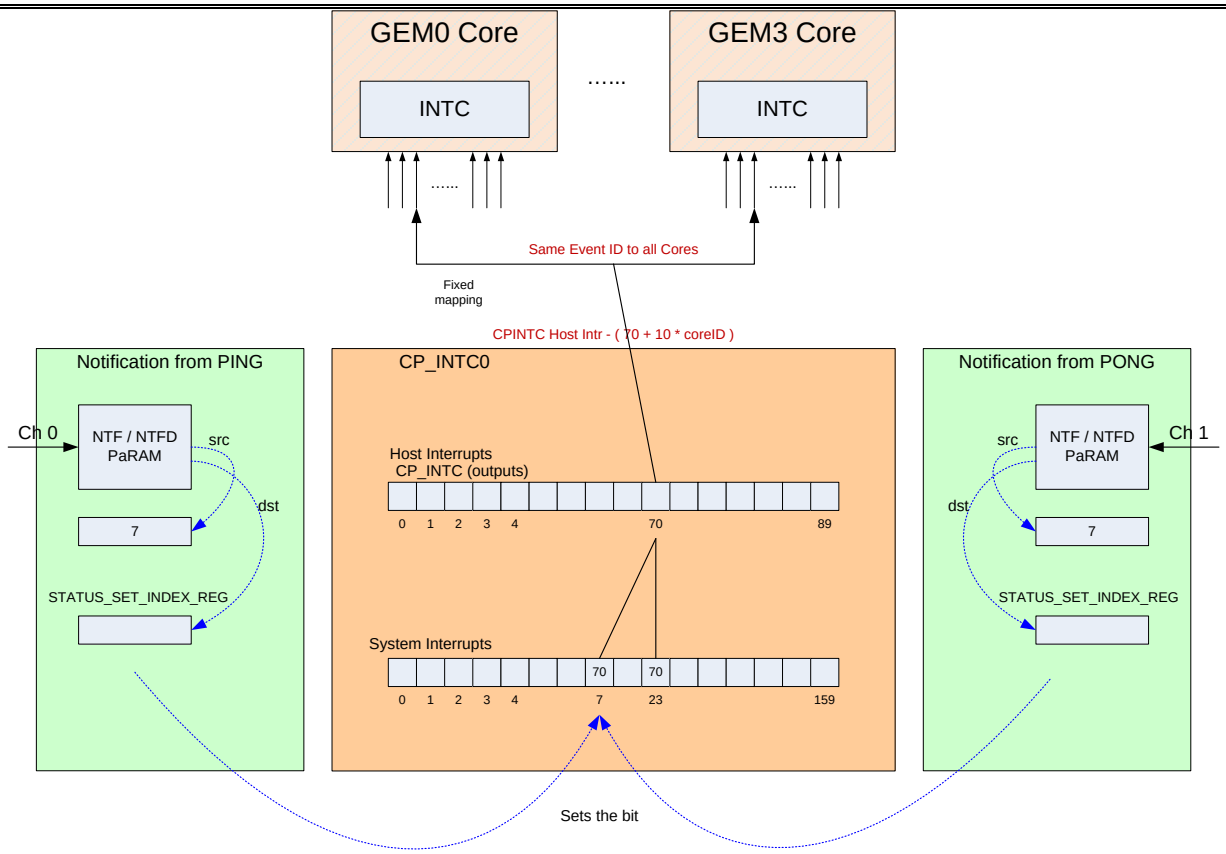

Figure 10 – Notification Mechanism Example

5.3 TCP3D Driver Interface

This chapter covers the details about the important interface features and usage. For specific details about all the APIs and its description can be obtained from the Doxygen generated API document in the driver installation. Some of the APIs are briefly addressed in the following sections as necessary for the illustration.

5.3.1 Initialization Sequence

Before using the driver the application performs several initialization steps as given below:

1) Initialize the EDMA low level driver including the registration of interrupts.
2) Open EDMA physical and linked channels required by driver.
3) Register call back functions associated with the REVT channels for control purposes.
4) Allocate memory for the driver by calling the driver APIs.
5) Initialize all the required configuration parameters in the driver initialization structure for the given instance.
6) Call driver initialization function.
7) Register the notification event with a call back function for CP_INTC0.

This is illustrated in Figure 11. These steps have to be done for using each TCP3D peripheral instance. Test application shows how to use the TCP3D initialization functions in a sample init

function to initialize either TCP3D_N peripherals, where N can be A, B, 0, 1, 2 etc. depending on which device is being used.

Application can use the two APIs, shown below, defined by the driver for querying the memory requirements of the driver and then allocate them as requested. The parameters passed by the structure *Tcp3d_SizeCfg* to the driver are used by the driver in computing the requirements.

```
Tcp3d_Result Tcp3d_getNumBuf (IN Tcp3d_SizeCfg  *cfg,
                              OUT Int16         *nbufs)

Tcp3d_Result Tcp3d_getBufDesc ( IN Tcp3d_SizeCfg    *cfg,
                                OUT Tcp3d_MemBuffer bufs[])
```

The allocated buffers are passed to the driver with *Tcp3d_init()* API using the buffer description structure *Tcp3d_MemBuffer*.

```
Tcp3d_Result Tcp3d_init( IN  Tcp3d_MemBuffer     bufs[],
                         IN  Tcp3d_InitParams    *drvInitParams)
```

Some of the important things done in the TCP3D driver initialization function are listed below:

- It does various checks and returns with appropriate messages defined by the *Tcp3d_Result* enum structure.
- Most of the pseudo PaRAM fields are pre-filled during initialization based on the parameters provided by the application. This minimizes the time to update the pseudo PaRAM set in the enqueue function.
- It resets and starts the TCP3D peripheral state machine by writing into its control registers with the prepared values.

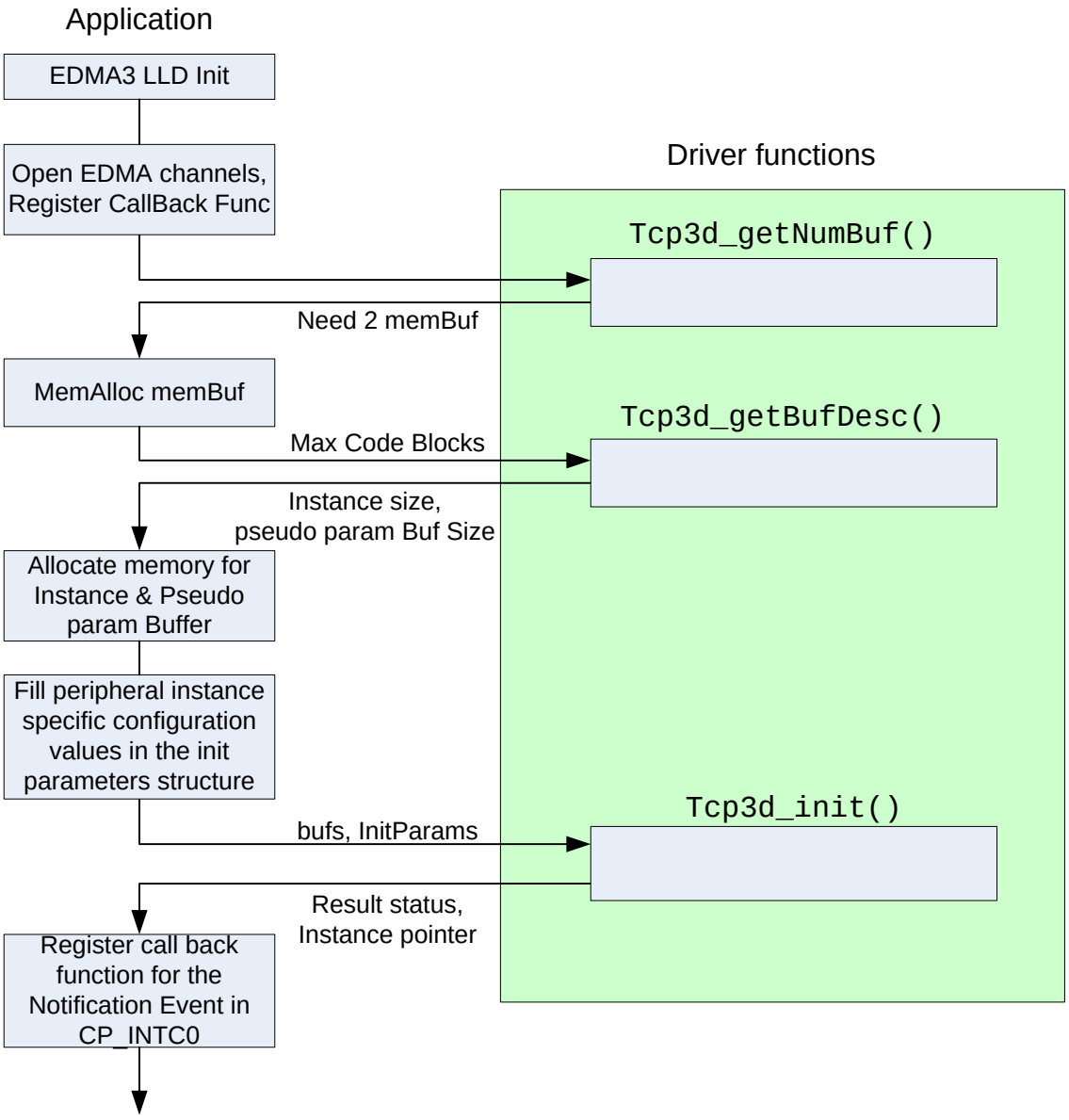

Figure 11 - TCP3D driver initialization sequence

5.3.2 Enqueue Function

Figure 12 shows the high-level flow diagram of the tasks done in the enqueue function and the PaRAM Update, Link and Chain operations are expanded in the Figure 13. Refer to the doxygen generated API documentation for function description with respect to the parameters.

This function primarily has two modes:

1) “Normal mode” – this is the mode where the function is called less than the allocated list capacity. It always returns with status TCP3D_DRV_NO_ERR in this mode.

2) "Wrap-around mode" – this is the mode where the function is called more than the allocated list capacity. It is possible that the function might return with status TCP3D_DRV_INPUT_LIST_FULL in this mode when the expected list does not have free entries.

Irrespective of the mode, the function tries to call the *Tcp3d_start()* function at the end when required flags are set at the time of the check.

As shown in Figure 12, this function has mainly two decision points. First check is done to find availability of free entry (altered between ping or pong lists) and the second one is for calling the *Tcp3d_start()* function. When a free entry is available, the main enqueue operations are done as illustrated in the flow diagram shown in Figure 13. These operations include pseudo PaRAM set updates for the given code block, linking the optional output PaRAMs, chain notification PaRAM and any required chaining to the previous block, storing necessary flags, updating counters, etc.

The API syntax is given below.

```
Tcp3d_Result Tcp3d_enqueueCodeBlock(IN  Tcp3d_Instance  *tcp3dInst,
                                    IN  UInt32          blockLength,
                                    IN  UInt32          *inputConfigPtr,
                                    IN  Int8            *llrPtr,
                                    IN  UInt32          llrOffset,
                                    IN  UInt32          *hdPtr,
                                    IN  Int8            *sdPtr,
                                    IN  UInt32          sdOffset,
                                    IN  UInt32          *statusPtr,
                                    IN  UInt8           ntfEventFlag)
```

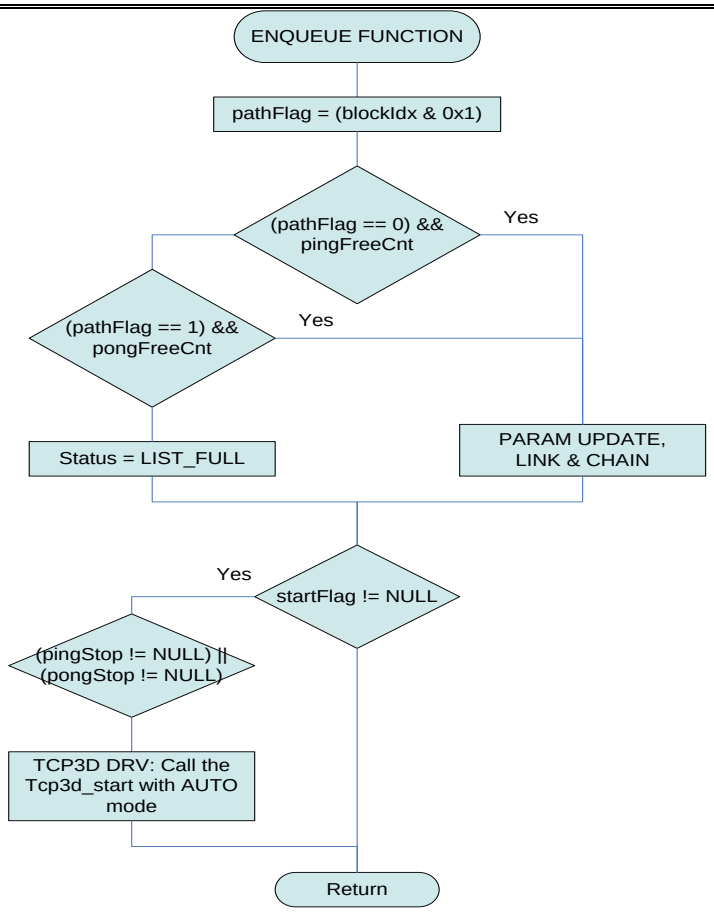

Figure 12 - Enqueue function flow diagram - high level

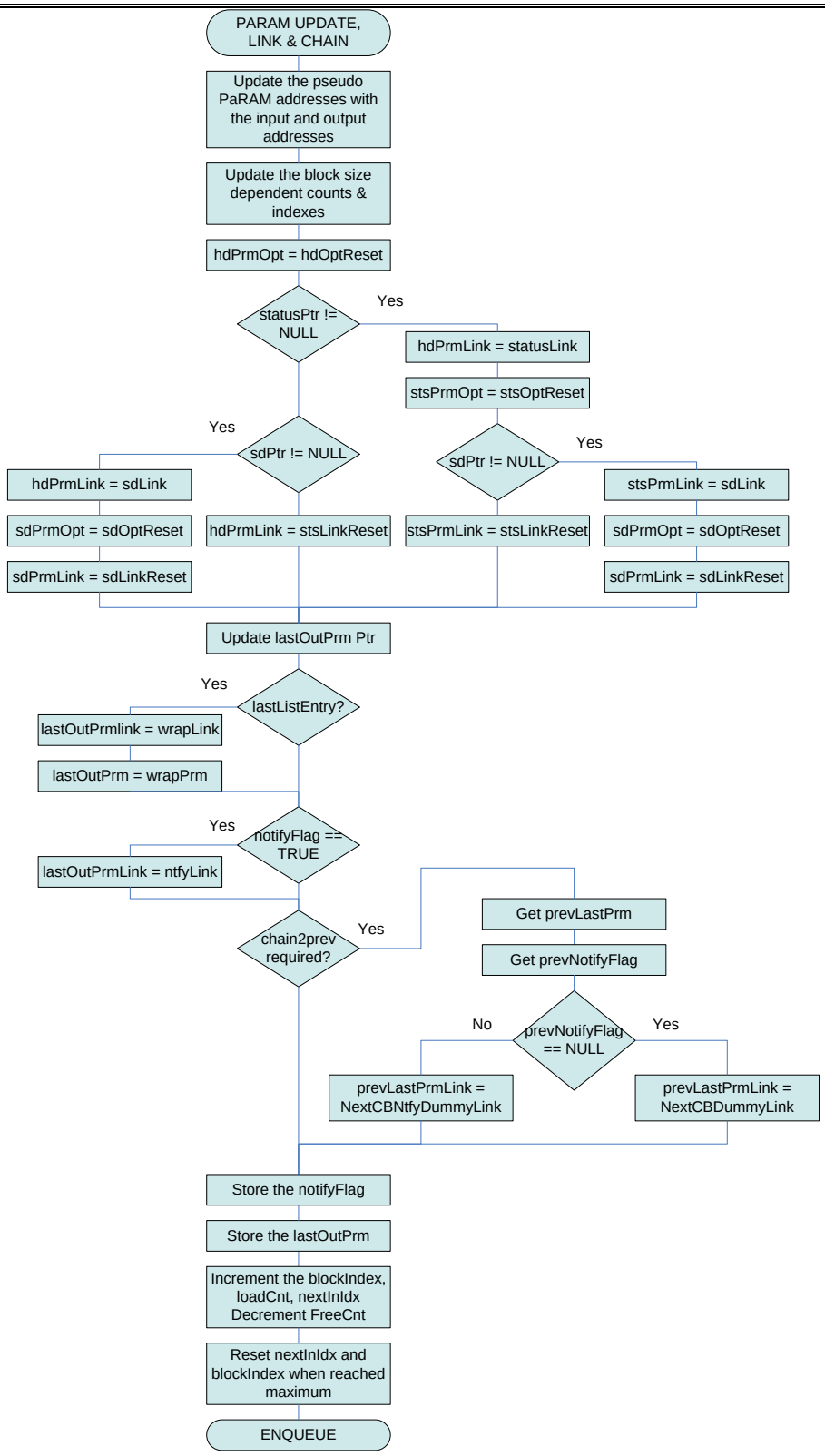

Figure 13 - Enqueue function flow diagram - PaRAM Update, Link & Chain

5.3.3 Start Function

After initialization the driver is in IDLE state. The start function can be used for doing two things which could affect the driver state.

1) Used first time after Enqueuing at least one code block to the input list by the application to bring the driver from IDLE to RUNNING state.
2) Used from bringing the driver state from PAUSE to RUNNING if there are more code blocks to be decoded in the input list.

The first condition happens only once after the driver initialization and the second condition can occur in two scenarios:

1) the application is enqueuing more code blocks and driver reached PAUSE state. In this case the start function gets called from the enqueue function with AUTO mode;
2) the application finished enqueuing all its code blocks and waiting for all the code blocks to be decoded and driver reached PAUSE state before completing the list. In this case the start function is called from the channel call back function, also referred to as the pause ISR.

The TCP3D driver will be in the RUNNING state as long as the input list has chained code blocks. If the decoding process is running at faster pace than the pre-processing, the driver will go to PAUSE state, and it will have to be restarted from the next call to the enqueue function.

The flow diagram of the start function is shown in Figure 14, Figure 15 and Figure 16. Its functionality mainly depends on the input parameter (startMode) and also depends on the stop flags (pingStop or pongStop). Table 5 gives the details about the various modes and their operations.

The start function syntax is given below.

```
Tcp3d_Result Tcp3d_start (  IN  Tcp3d_Instance  *inst,
                            IN  UInt8           startMode);
```

Mode	Description
AUTO START	1) Checks both PING & PONG input lists to see if any more code blocks are there for decoding. Also update the list variables. 2) If the corresponding stop flag is set, then trigger the associated L2P channel to take the next code block from the list for decoding. 3) If any path is triggered, associated stop flag is cleared. 4) If any path is triggered; the state is changed to RUNNING.
PING START	1) No checking done inside. Application must determine there are more code blocks in the list. 2) If the pingStop flag is set, triggers the ping L2P channel to take the next code block from the ping list for decoding. 3) If triggered, clears the pingStop flag.

	4) If triggered, the state is changed to RUNNING.
PONG START	1) No checking done inside. Application must determine there are more code blocks in the list. 2) If the pongStop flag is set, triggers the pong L2P channel to take the next code block from the pong list for decoding. 3) If triggered, clears the pongStop flag. 4) If triggered, the state is changed to RUNNING.

Table 5 - Start Modes and their operations

Note: It is required that the application must call this function first time to set the startFlag which enables the start function to be called from the enqueue function as needed.

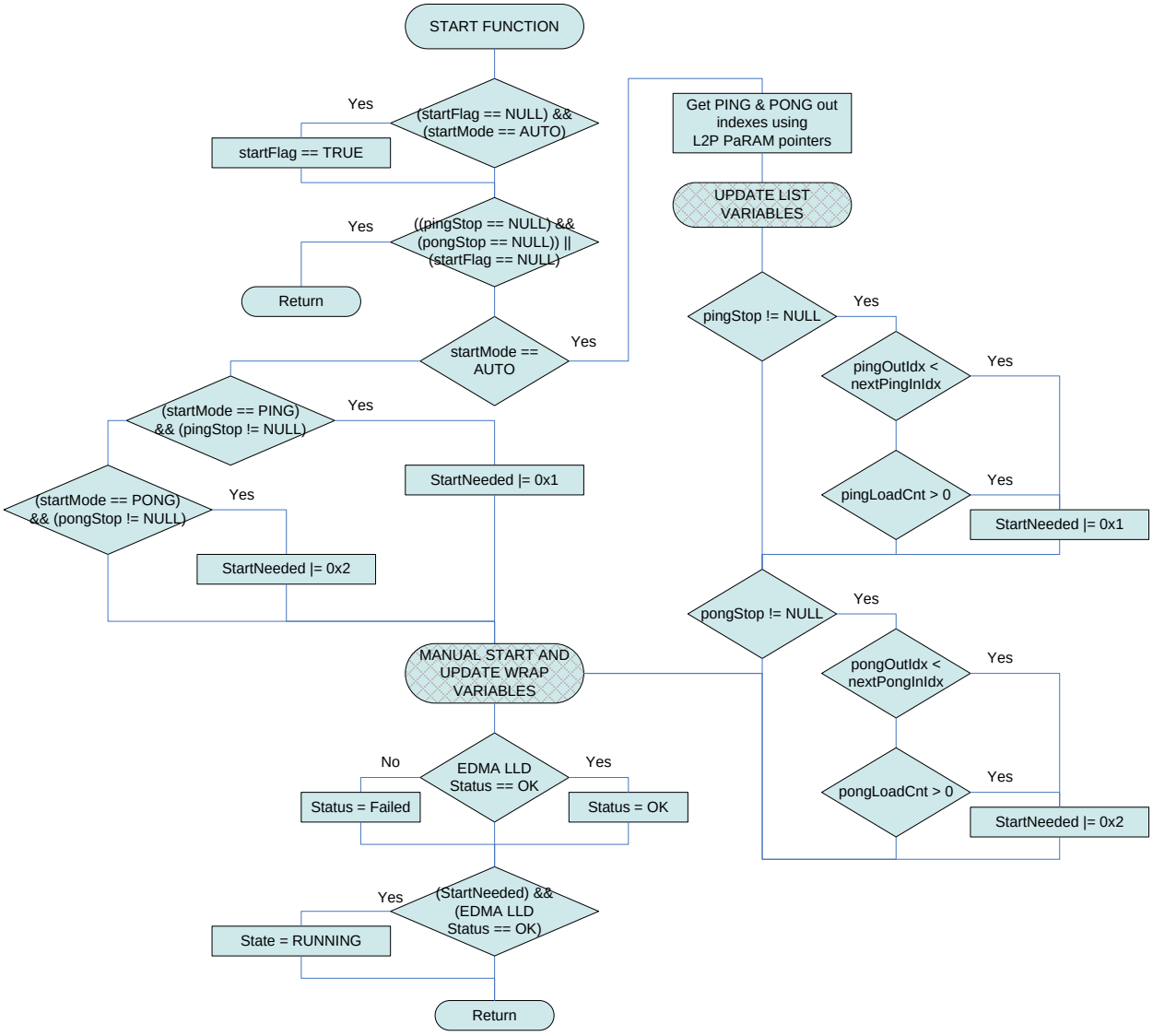

Figure 14 - TCP3D driver start function

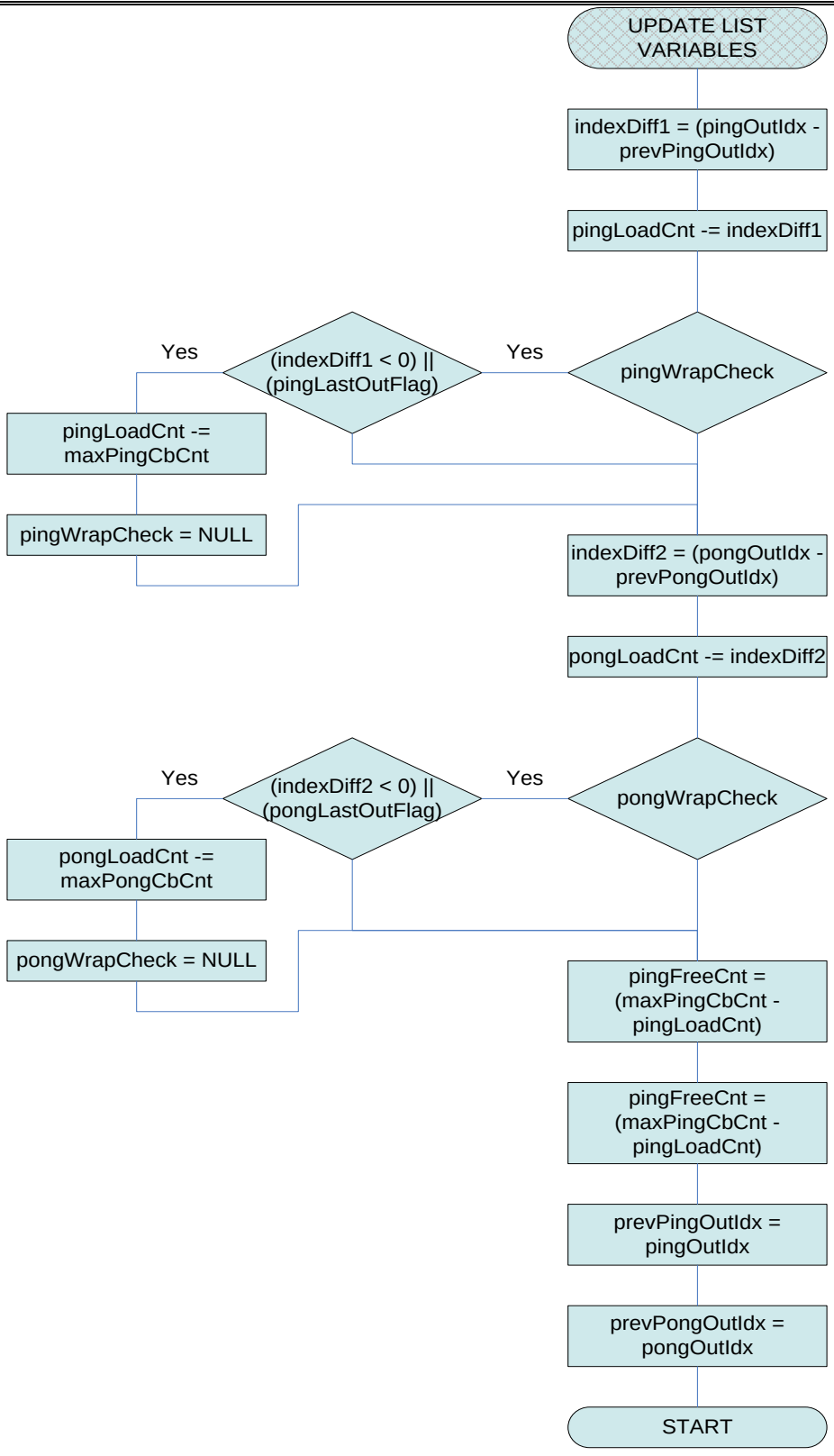

Figure 15 - TCP3D driver start function – Update list variables

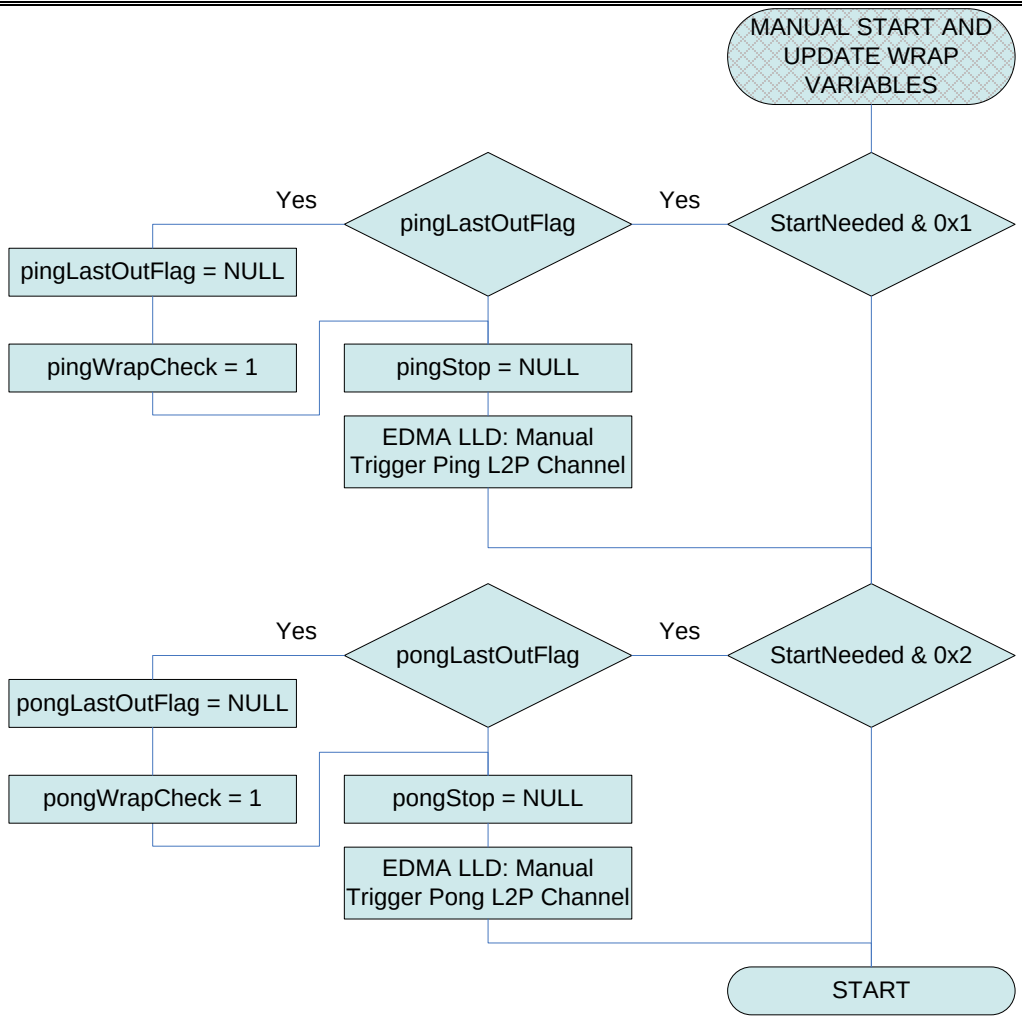

Figure 16 - TCP3D driver start function – Manual Start and update wrap variables

5.3.4 Status & Control Functions

These functions are used for making changes to the driver operations at run-time. Both of the functions have some commands defined for use with the current design and could be extended in future. Refer to the doxygen documentation for the list of supported commands and the associated data structure descriptions.

In the current implementation, the control function is used mainly to control the EDMA completion interrupts during the driver execution. Refer to section 5.2.4.2 to know which PaRAMs are changed in run-time.

5.4 OSAL

The OSAL is the operating system abstraction layer which is used to port the TCP3D driver to a specific OS. The OSAL callouts are implemented in the "`tcp3d_osal.h`" header file and need to be ported by the application developers to their specific operating system.

5.4.1 Logging API

Internally the TCP3D driver uses the `Tcp3d_osalLog` macro to perform all logging operations. The OSAL adaptation layer ports this macro to the following API prototype:

```
void Osal_tcp3dLog( String fmt, ... )
```

The parameter '`fmt`' is a `printf` style formatted string. This should only be defined and used for debugging purposes.

6 Testing

TCP3D driver testing is done with pre-generated input and reference outputs. Various test cases are defined for LTE, WIMAX & WCDMA (3GPP) modes with different combination of parameters and block sizes.

The following table gives the description of various test cases used for regression test for the driver. In all tests, the optional outputs (STS & SD) are randomly selected otherwise noted.

Mode	Test Folder Name	# of tests	Description
LTE & Dual Map	Test1_lte	6	Mid range block sizes, smallest and largest block sizes.
	Test2_lte	4	Mid range block sizes, no optional outputs (STS & SD).
	Test3_lte	6	All Max Block sizes.
	Test4_lte	6	Mid range block sizes.
	Sim_config\LTE	20	All Tests taken from SIMULATOR test bench.
	LTE	100	Test cases generated using matlab script with random parameters used. Aimed for testing the wrap-around mode.
WiMax & Dual Map	Test1_wimax	5	Lower range block sizes and largest block size.
	Test2_wimax	4	Mid range block sizes, no optional outputs (STS & SD).
	Test3_wimax	5	Mid range block sizes.
	Sim_config\WIMAX	13	All Tests taken from SIMULATOR test bench.
	WIMAX	100	Test cases generated using matlab script with random parameters used. Aimed for testing the wrap-around mode.
WCDMA & Split Mode	Test1_wcdma	32	Block sizes around 100 & 800 and few larger block sizes around 2000. Repeated tests to cover liner increase or decrease of sizes.
	Test2_wcdma	6	Lower blocks in steps of 100 from 100 & Upper range block sizes around 5000.
	Test3_wcdma	9	Lower range block sizes around 50 and larger block sizes around 1000 & 2000.
	Sim_config\WCDMA	28	All Tests taken from SIMULATOR test bench.
	WCDMA	100	Test cases generated using matlab script with random parameters used. Aimed for testing the wrap-around mode.

Table 6 – Test Cases Used

Note: There are no test cases defined for Single Map decoder mode since the driver does not support this mode.

Testing of this driver is done using BIOS and all OSAL APIs including those of the dependent packages are mapped for it. The Tester task is the main task used for running in a while loop for all test cases after setting some global test parameters such as *testMaxCodeBlocks* and *instNum* (for selecting the TCP3D peripheral instance) that are used for all test cases. Also, the *instNum* used for running the tests is chosen based on which core the test is loaded. Figure 17 shows the

tasks and the drivers with their execution sequence – test #1 is normal case and test #2 is LIST_FULL case.

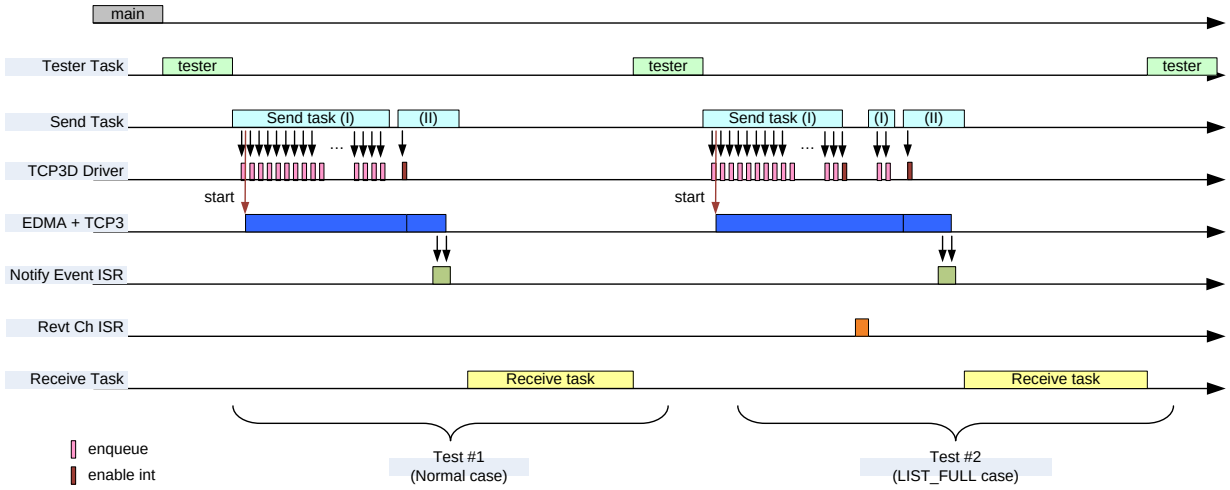

Figure 17 - TCP3D driver test sequence diagram

As you can see from the Figure 18, Tester task initiates the test by calling appropriate initialization functions and then posts a Semaphore to wake the Send Task and wait for Receive task completion. Send task has two parts and in the first part all of the code blocks to be decoded are sent to the input list using the *Tcp3d_enqueueCodeBlock()* driver API and also the driver is started using the *Tcp3d_start()* API. When the driver is used in wrap-around mode, the LIST_FULL return error is handled in the first part of the send task. In the LIST_FULL case, the test bench is setup such that the enqueue function is called again and again until successful. See Figure 19, Figure 20, Figure 21, Figure 22 and Figure 23 for details on various tasks used in the test framework.

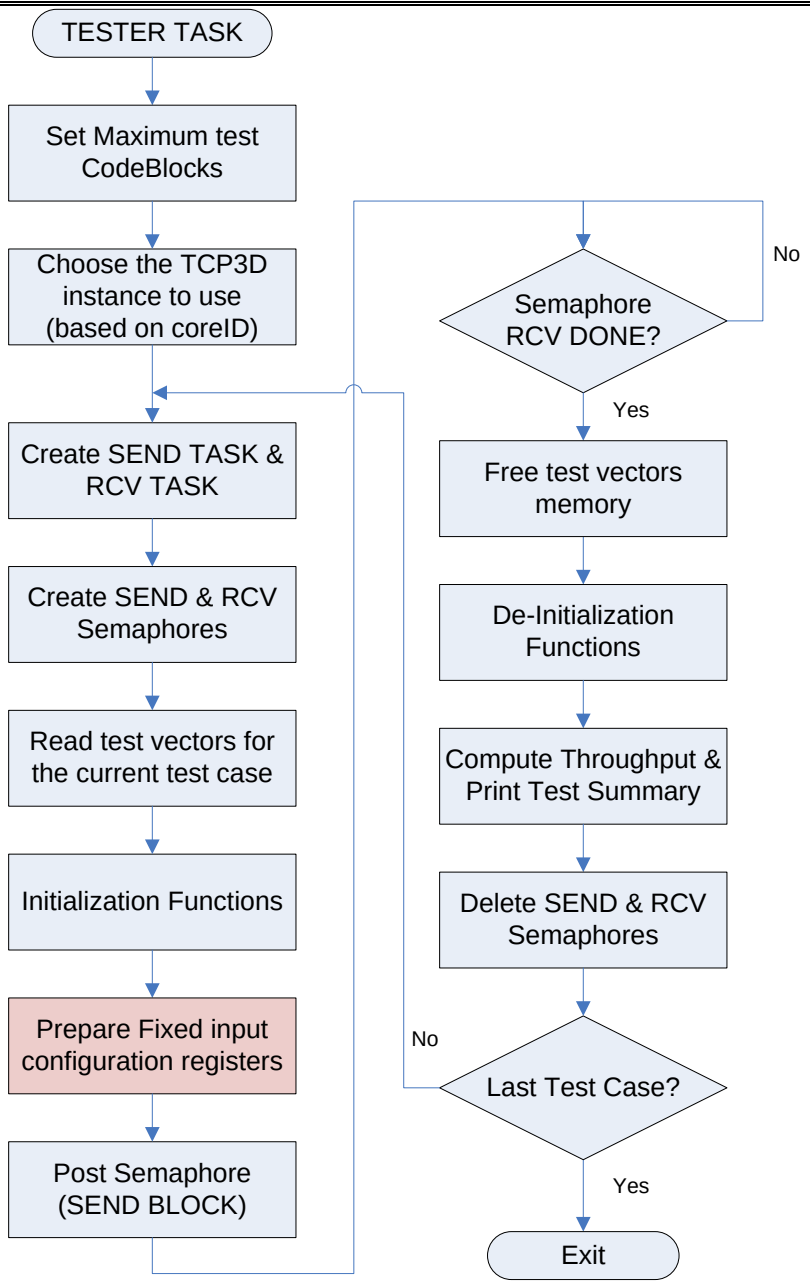

Figure 18 - TCP3D driver test – Tester Task flow diagram

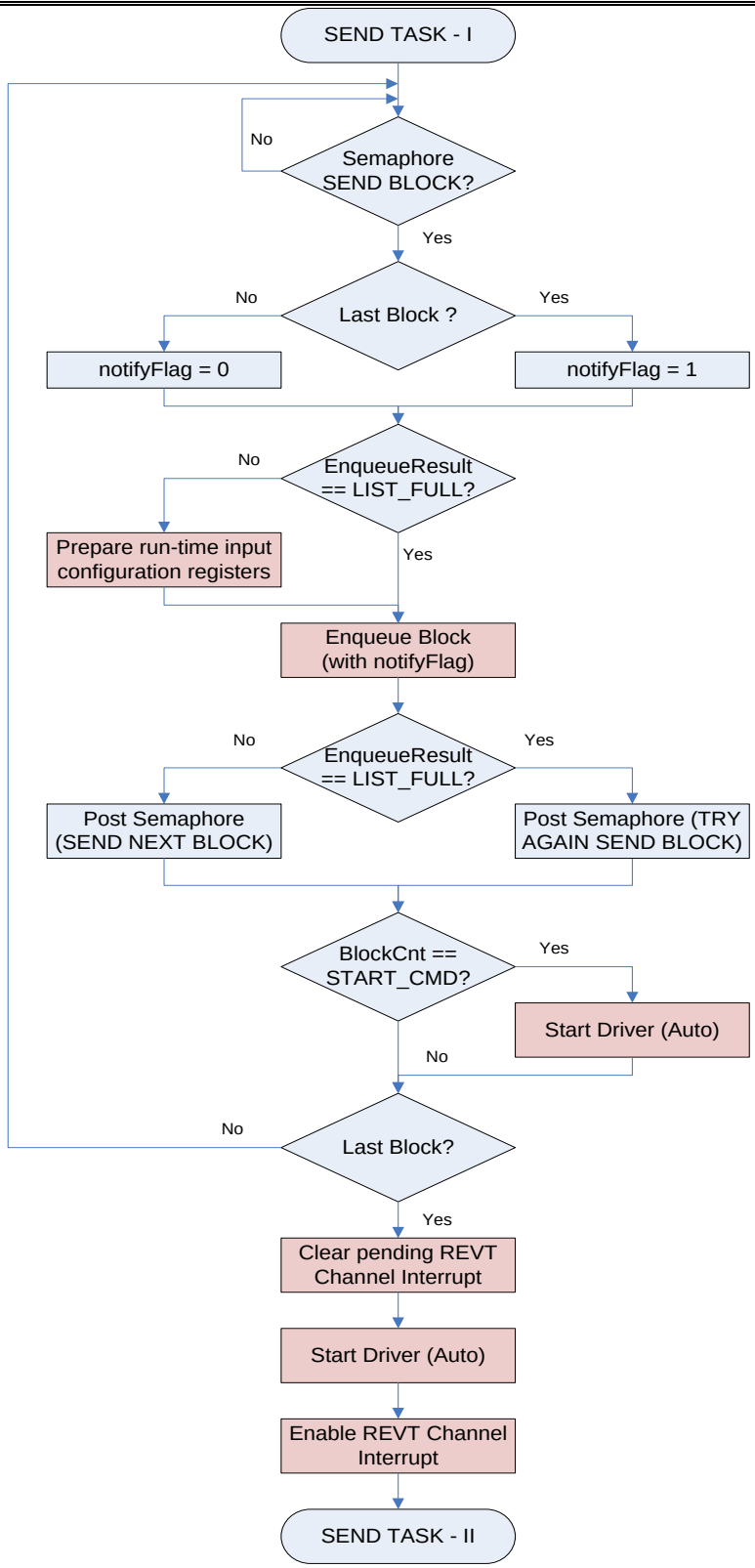

Figure 19 - TCP3D driver test – Send Task (part – I) flow diagram

The second part of the Send Task does the following:

1) Wait until the required notifications are received following the completion of both the input lists (PING & PONG) and post semaphore to start the Receive Task.
2) Restart the driver, if it reaches PAUSE state before completing all decoding.
3) In case of split mode, there is a possibility of race condition in the notification which might result to missing one notification. So, whenever a single notification is received we check the last two block outputs to declare all decoding is complete.

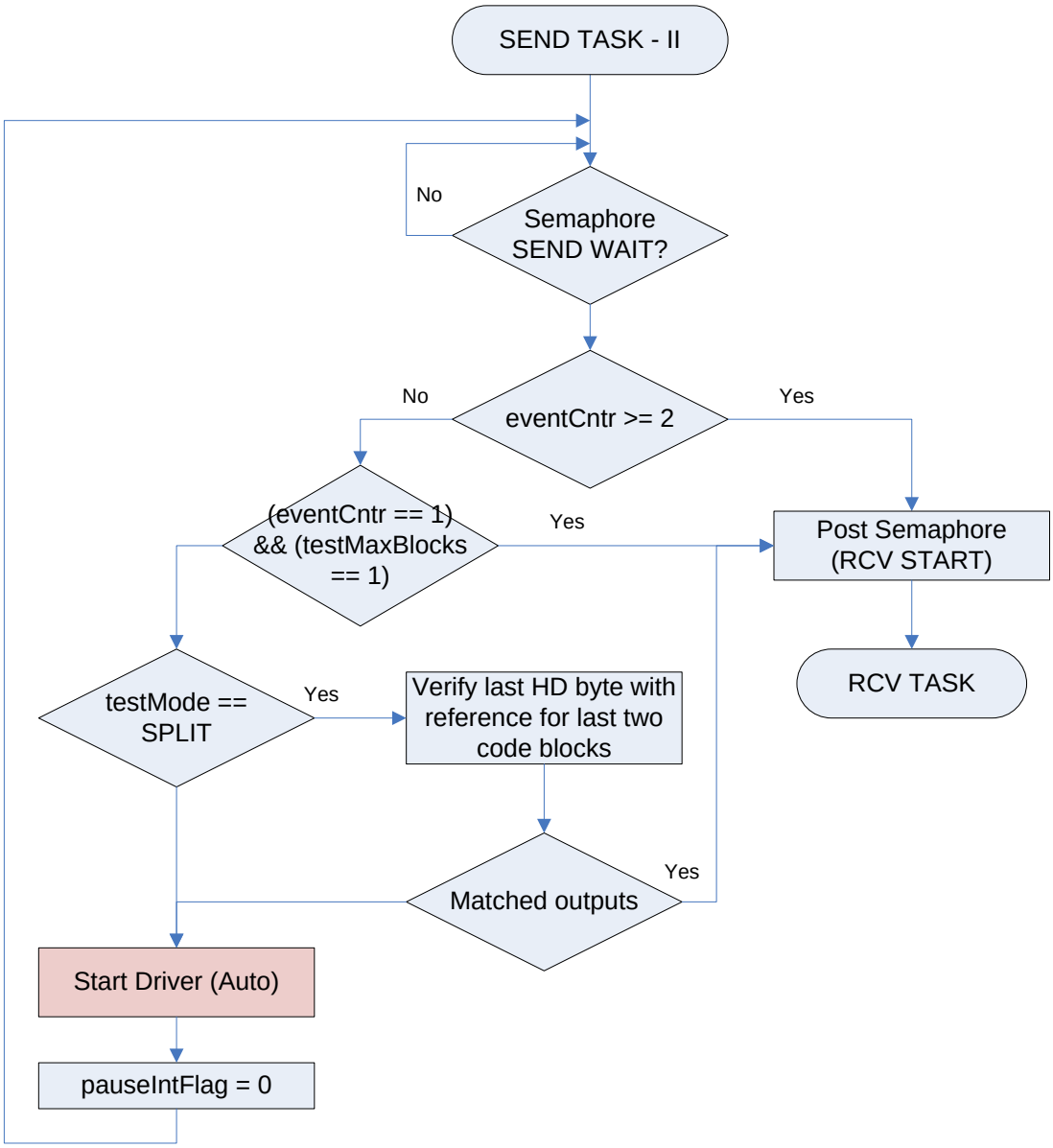

Figure 20 - TCP3D driver test – Send Task (part – II) flow diagram

The Receive Task compares all the selected outputs with the expected values for each code block in the test case and reports any errors with appropriate error messages. When the comparisons of all code blocks are done, it will post Semaphore to pass the control to Tester Task which does the de-initialization for this test and goes to the next test if any.

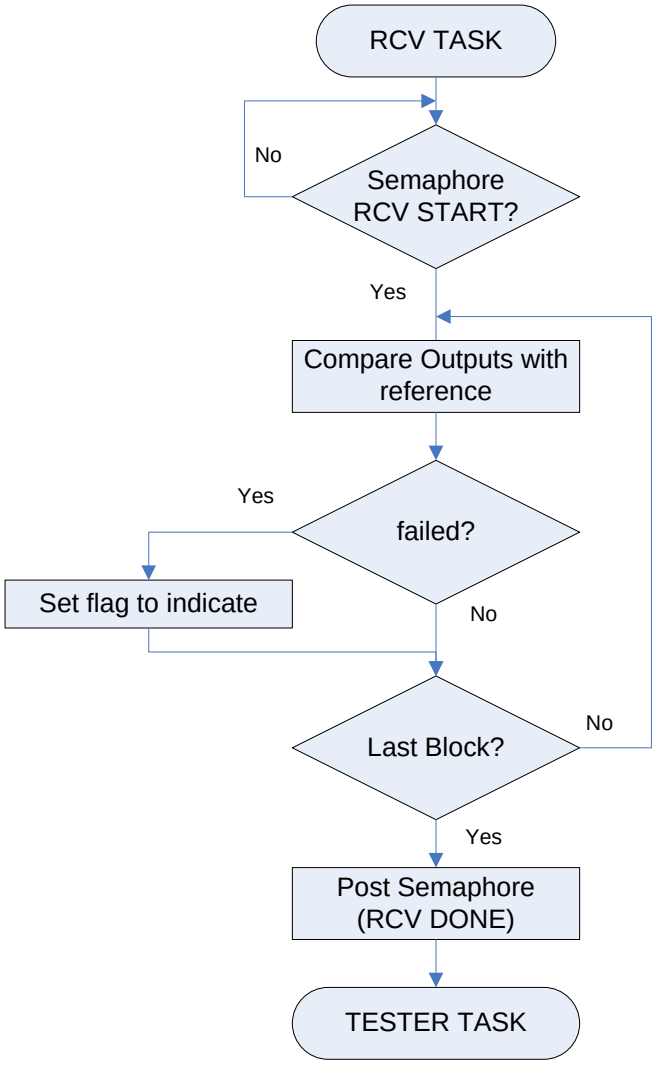

Figure 21 - TCP3D driver test – Receive Task flow diagram

The REVT channel call back function, shown in Figure 22, is called when the interrupts are enabled either for the REVT or L2P channels. Because, the TCC value for both the channels point to the REVT channel number at any given time. In any case, this function transfers the control to the Send Task to take appropriate action based on the flags.

This function usage is shown in the second test in the test sequence diagram in Figure 17 in the illustration of the LIST_FULL error case.

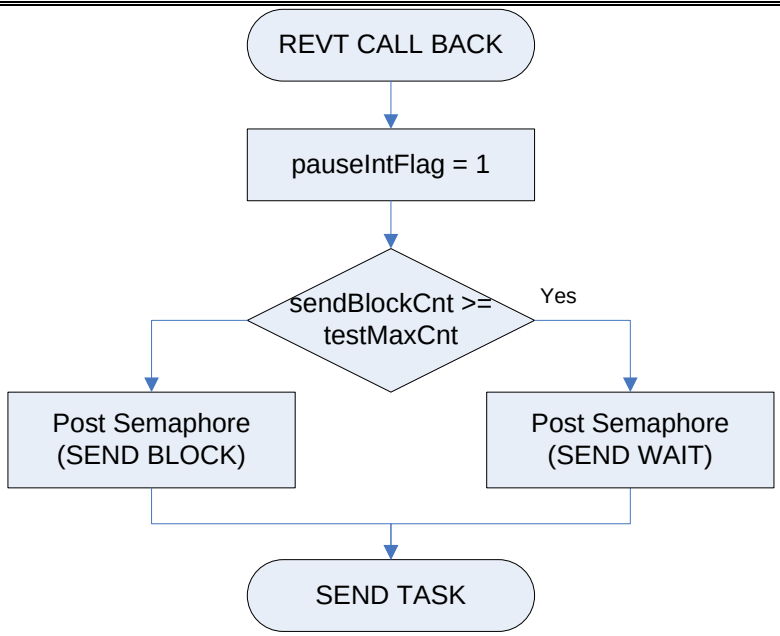

Figure 22 - TCP3D driver test – REVT Channel Call Back Function

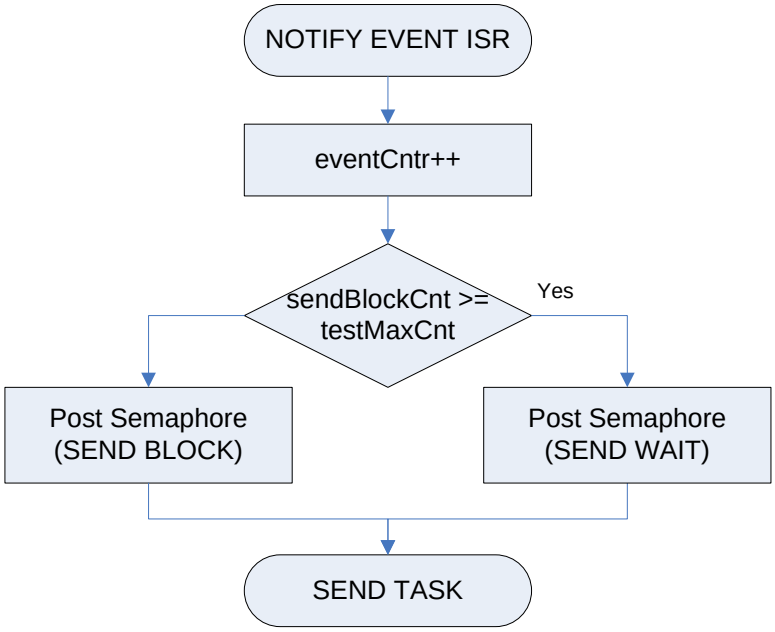

Figure 23 - TCP3D driver test – Notification ISR Function

7 Integration

The TCP3D driver depends on the following components:

a. EDMA3 LLD
b. CSL

These components need to be installed before the TCP3D driver can be integrated. The TCP3D driver is released in source code and in pre-built library. Applications can decide how to use the TCP3D driver.

The TCP3D driver release notes indicate the version of the above components which that release is dependent upon. The next steps use the version numbers for illustrative purpose only.

7.1 Pre-built approach

In this approach the application developers decide to use the TCP3D driver pre-built libraries as is. The following steps need to be done:

a. The application developers modify their application configuration file to use the TCP3D package.

```
var Tcp3d = xdc.loadPackage('ti.drv.tcp3d');
```

b. Ensure that the XDCPATH is configured to have the path to the TCP3D package
c. This implies that XDC Configuration scripts will link the application using the TCP3D Driver libraries (`Module.xs`)
d. The application authors need to provide an OSAL implementation file for TCP3D and ensure that this linked with the application; failure to do so will results in linking errors. The OSAL source file should have the following TCP3D OSAL functions implemented:

```
Void  Osal_tcp3dLog( String fmt, ... )
```

Note: Since TCP3D depends upon EDMA3, the OSAL implementation should also have their implementations. Please refer to the EDMA3 OSAL definitions for more information.

If the application is not using XDC then replace steps (a) and (b) above with the following steps instead:

a. Append the include path to the top level TCP3D package directory
b. Make sure the TCP3D pre-built libraries are added to the application project and the library search path is configured correctly.

This approach is highlighted in the TCP3D "`example`" projects.

7.2 Rebuild library

In this approach the application developers decide to use the TCP3D driver source code and add these files to the application project to rebuild the TCP3D driver code base. The following steps need to be redone:

a. Application developers should port the file "`tcp3d_osal.h`" to their operating system environment. *Developers are recommended to create a copy of this file and place it in*

their application directory. They should use the file which is provided in the TCP3D installation only as a template. The goal here should be to map the `Tcp3d_osalXXX` macros to the OS calls directly thus reducing the overhead of an API callout. For example:

```
#define Tcp3d_osalLog    System_printf
```

b. Application developers should port the file "`tcp3d_drv_types.h`" to the application environment. *Developers are recommended to create a copy of this file and place it in their application directory.*

c. Append the include path to the top level TCP3D package directory

d. Add the TCP3D driver files listed below from the `src` directory to the application build files.

```
tcp3d_drv.c
tcp3d_reg.c
```

The approach above is highlighted in the TCP3D "`test`" projects.

8 Future Extensions

a. Need to add support for Single Map Decoder, if required by customers.