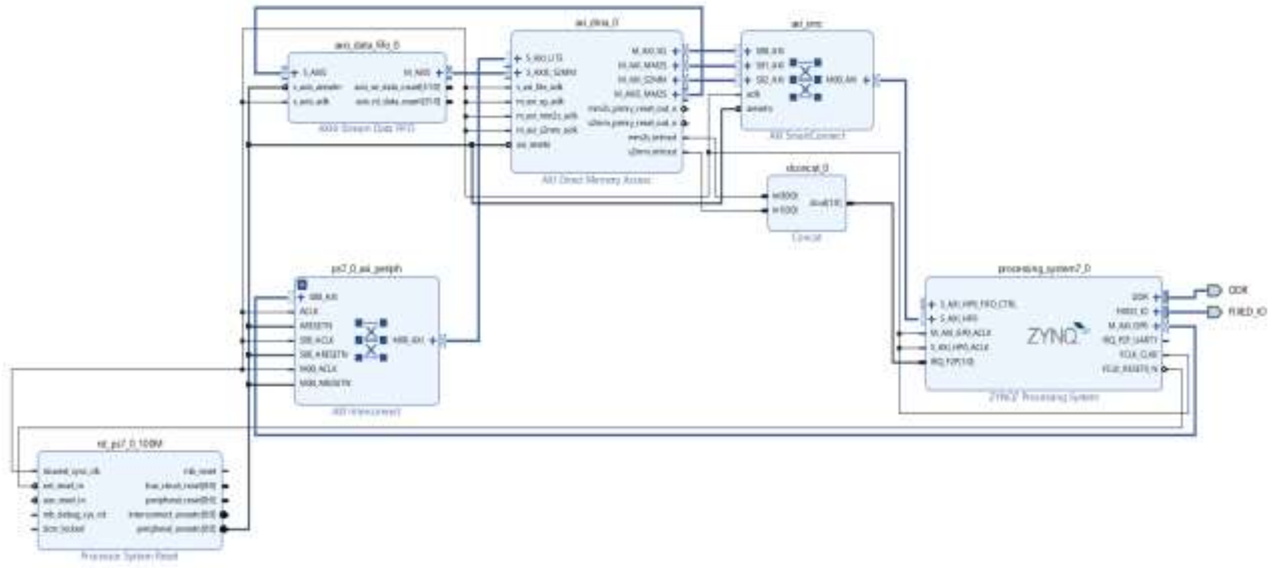
DOCUMENTATION OF DMA IN SCATTER GATHER MODE (Example):-

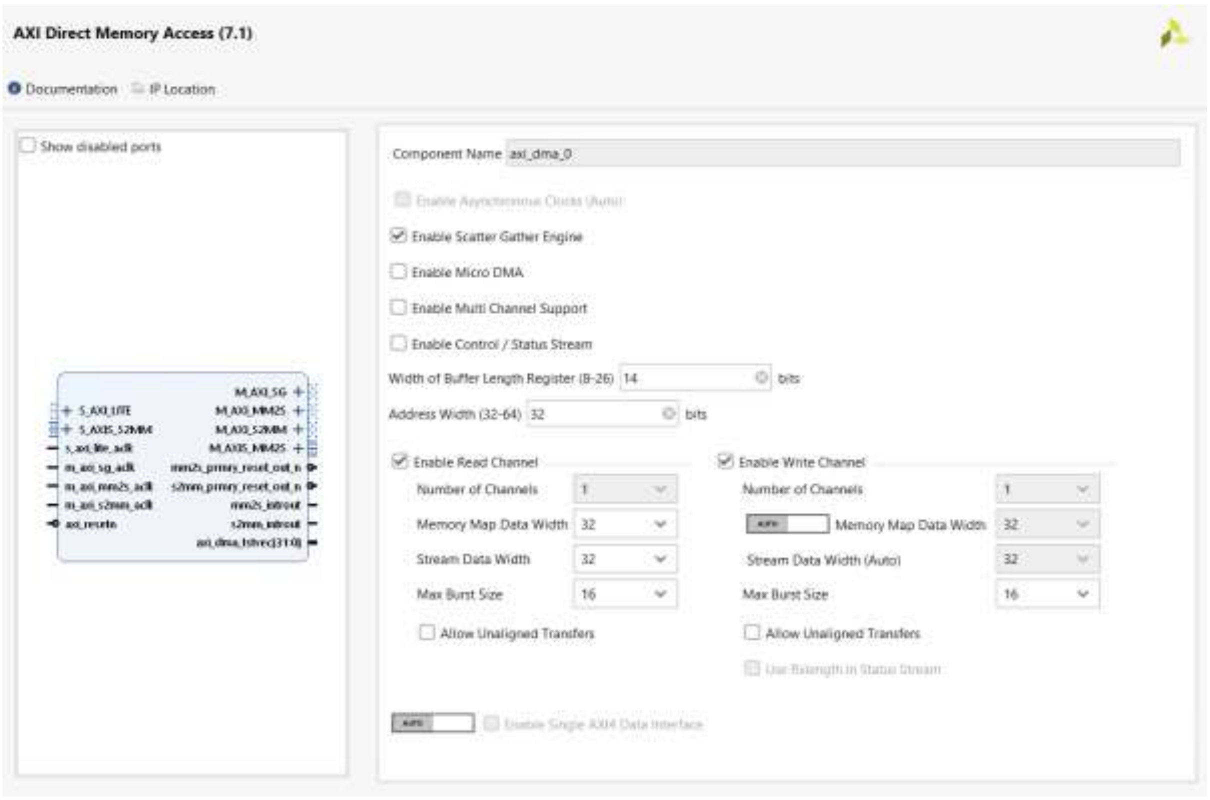
The Vivado project we will discuss focuses on a basic Simple Poll DMA example in scatter-gather mode. The HDL design incorporates a Zynq Processing System (PS) to configure the DMA controller, a DMA configured for scatter-gather (SG) mode, and an AXI-Stream Data FIFO for loopback functionality.

Ensure that the "Enable Scatter-Gather Engine" option is checked to activate SG mode. Additionally, the "Width of Buffer Length Register" can be set to any value up to 26, depending on the size of the data transfer for each Block Descriptor, as this affects the maximum length for the data transfer parameter. The parameters for the AXI4-Stream Data FIFO should be configured based on the type of data transfer being performed.

The HDL design used is :



The current DMA configuration in the above block design is:



AXI Direct Memory Access (7.1)

Component Name axi_dma_0

[ ] Enable Asynchronous Clocks (Auto)

[✓] Enable Scatter Gather Engine

[ ] Enable Micro DMA

[ ] Enable Multi Channel Support

[ ] Enable Control / Status Stream

Width of Buffer Length Register (8-26) 14 bits

Address Width (32-64) 32 bits

[✓] Enable Read Channel		[✓] Enable Write Channel	
Number of Channels	1	Number of Channels	1
Memory Map Data Width	32	Memory Map Data Width	32
Stream Data Width	32	Stream Data Width (Auto)	32
Max Burst Size	16	Max Burst Size	16
[ ] Allow Unaligned Transfers		[ ] Allow Unaligned Transfers	
		[ ] Use Rxlength in Status Stream	

[ ] Enable Single AXI4 Data Interface

Upon reviewing the bare-metal C code, some of the user-specified memory assignments are as follows:-

- `MEM_BASE_ADDR -> 0x01000000` (All memory assignments will be relative to this base address)
- `TX_BD_SPACE_BASE -> MEM_BASE_ADDR`
- `TX_BD_SPACE_HIGH -> MEM_BASE_ADDR + 0x00000FFF` (A total of 4096 bytes, starting from the base address 0x01000000, will be allocated for storing TX Buffer Descriptor (BD) information)
- `RX_BD_SPACE_BASE -> MEM_BASE_ADDR + 0x00001000` (starting from address 0x01001000)
- `RX_BD_SPACE_HIGH -> MEM_BASE_ADDR + 0x00001FFF` (A total of 4096 bytes, starting from the RX base address 0x01001000 and extending to 0x01001FFF, will be used for storing RX Buffer Descriptor (BD) information)
- `TX_BUFFER_BASE -> MEM_BASE_ADDR + 0X00100000` (From this buffer address, we can store any data to be transferred via DMA, starting from 0x01100000)
- `RX_BUFFER_BASE -> MEM_BASE_ADDR + 0X00300000` (From this buffer address, we can store any data received via DMA, starting from 0x01300000)
- `RX_BUFFER_HIGH -> MEM_BASE_ADDR + 0x004FFFFF` (This is the maximum address up to which we can store received data)

The functions used are:

- TxSetup – This sets up the TX channel of the DMA engine, preparing it for transmission. The data transfer will be handled in the next function. In this section, we will perform the basic allocation of free Buffer Descriptors (BDs) and prepare them for hardware use during data transfer. While allocating, we will assign all available BDs, although not all may be utilized for the transfer.
- SendPacket – Data transfer is handled in this section. We can specify how many Buffer Descriptors (BDs) will be used for the transfer.
- RxSetup - This sets up the RX channel of the DMA engine, preparing it for reception. In this section, we will prepare the entire RX Buffer Descriptor (BD) ring, which includes creating and initializing BDs, allocating all available BDs, updating the BD registers, and starting the channel.
- CheckDmaResult – Waits until DMA transaction is over, checks data.

Let’s take a look into the main code of the design:

1. The pointer to the DMA configuration structure is stored in the variable ‘Config’. The structure is ‘XAxiDma_Config’, which holds the hardware configuration information for the application driver, including all parameters set in the HDL DMA block. The structure is defined as follows:

```
typedef struct {
	u32 DeviceId;
	UINTPTR BaseAddr;
	int HasStsCntrlStrm;
	int HasMm2S;
	int HasMm2SDRE;
	int Mm2SDataWidth;
	int HasS2Mm;
	int HasS2MmDRE;
	int S2MmDataWidth;
	int HasSg;
	int Mm2sNumChannels;
	int S2MmNumChannels;
	int Mm2SBurstSize;
	int S2MmBurstSize;
```

```
        int MicroDmaMode;
        int AddrWidth;              /**< Address Width */
        int SgLengthWidth;

} XAxiDma_Config;
```

----

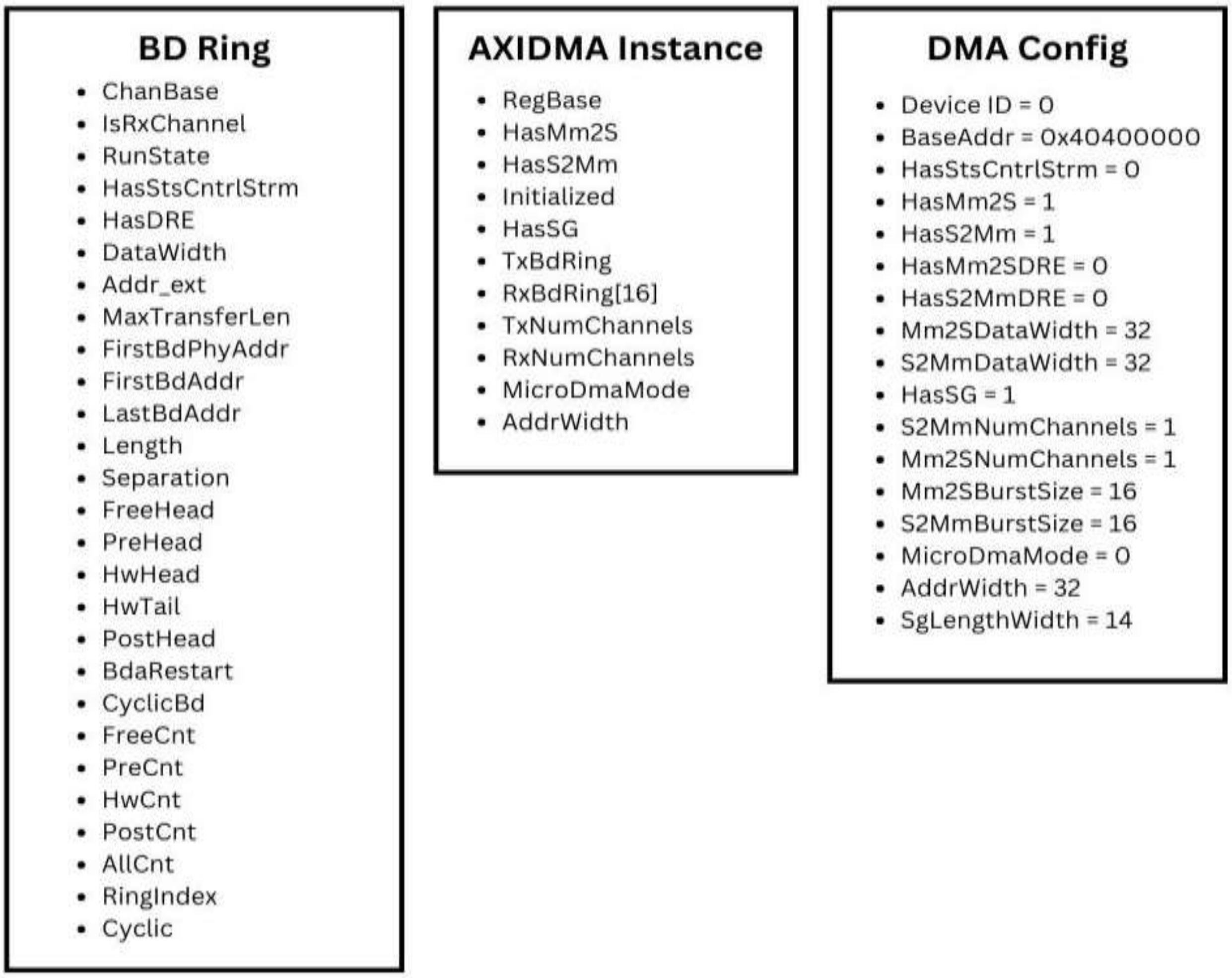
2. We now examine the hardware configuration for a specific DMA instance. Since we are using only one DMA in our case, the corresponding DMA ID will be passed to the function 'XAxiDma_LookupConfig'. This function checks the provided DMA ID against the configuration entries. If a match is found, it returns a pointer to the configuration structure for that DMA ID. Essentially, the function iterates through multiple DMA instances, looking for a matching ID, and returns the pointer for the specified DMA configuration when found. The function is as follows –

```
#define XPAR_XAXIDMA_NUM_INSTANCES 1

→Config = XAxiDma_LookupConfig(DMA_DEV_ID);

XAxiDma_Config * XAxiDma_LookupConfig (u32 DeviceId)
{
        extern XAxiDma_Config XAxiDma_ConfigTable[];
        XAxiDma_Config *CfgPtr;
        u32 Index;
        CfgPtr = NULL;

        for (Index = 0; Index < XPAR_XAXIDMA_NUM_INSTANCES; Index++) {
                if (XAxiDma_ConfigTable[Index].DeviceId == DeviceId) {
                        CfgPtr = &XAxiDma_ConfigTable[Index];
                        break;
                }
        }
        return CfgPtr;
}
```

In this case, DMA_DEV_ID corresponds to that of 1st DMA instance, which is 0. Additionally, the parameter `XPAR_XAXIDMA_NUM_INSTANCES` has a value of 1, meaning the **for** loop will iterate only once, checking the ID and returning configuration pointer.

3. **INITIALIZE DMA ENGINE**
Once the configuration for the given DMA ID is found, we need to initialize the DMA engine. To do this, we first define a device instance called AxiDma, which is of type XAxiDma. Each DMA in the design should have its own instance of the XAxiDma structure, as their attributes may vary. Since our design includes only one DMA, we use the variable name AxiDma.
    - The structure XAxiDma looks like-

```
        XAxiDma AxiDma;

        typedef struct XAxiDma {
                UINTPTR RegBase; /* Virtual base address of DMA engine */
                int HasMm2S;        /* Has transmit channel or not*/
                int HasS2Mm;        /* Has receive channel or not*/
                int Initialized;    /* Driver has been initialized */
                int HasSg;      /* if scatter gather is supported */
                XAxiDma_BdRing TxBdRing; /*BD container mngmnt for TX chnl */
                XAxiDma_BdRing RxBdRing[16];/*BD container mngmnt for RX chn*/
                int TxNumChannels; /* no of TX channels available */
                int RxNumChannels; /* no of RX channels available */
        int MicroDmaMode;    /* has Micro DMA mode selected */
        int AddrWidth;                /**< Address Width */
    } XAxiDma;
```



BD Ring

- ChanBase
- IsRxChannel
- RunState
- HasStsCntrlStrm
- HasDRE
- DataWidth
- Addr_ext
- MaxTransferLen
- FirstBdPhyAddr
- FirstBdAddr
- LastBdAddr
- Length
- Separation
- FreeHead
- PreHead
- HwHead
- HwTail
- PostHead
- BdaRestart
- CyclicBd
- FreeCnt
- PreCnt
- HwCnt
- PostCnt
- AllCnt
- RingIndex
- Cyclic

AXIDMA Instance

- RegBase
- HasMm2S
- HasS2Mm
- Initialized
- HasSG
- TxBdRing
- RxBdRing[16]
- TxNumChannels
- RxNumChannels
- MicroDmaMode
- AddrWidth

DMA Config

- Device ID = 0
- BaseAddr = 0x40400000
- HasStsCntrlStrm = 0
- HasMm2S = 1
- HasS2Mm = 1
- HasMm2SDRE = 0
- HasS2MmDRE = 0
- Mm2SDataWidth = 32
- S2MmDataWidth = 32
- HasSG = 1
- S2MmNumChannels = 1
- Mm2SNumChannels = 1
- Mm2SBurstSize = 16
- S2MmBurstSize = 16
- MicroDmaMode = 0
- AddrWidth = 32
- SgLengthWidth = 14

These are some of the structures we use in this code. DMA config is updated based on hardware setting of DMA, which you can see from Fig 2 of DMA provided above.

- The XAxiDma_CfgInitialize function is used to initialize the DMA engine instance with the configuration information found in step 2. This function has 2 parameters, a pointer to DMA instance to be initialized and pointer to configuration structure containing data for updation.
    a. First attribute Initialized is set to 0 indicating the DMA is not initialized yet.
    b. Then check if Config pointer is valid or not.
    c. Clear all values of the DMA to 0 using memset function.
    d. We then assign RegBase of DMA with that of BaseAddr of the config.(Base address here is 0x40400000)
    e. Similarly we copy the attributes HasMm2s, HasS2Mm, HasSg, MicroDmaMode, AddrWidth, TxNumChannels, RxNumChannels from config to DMA instance.
    f. Based on number of channels, it sets the maximum transfer length. If more than 1 channel is available, it sets this to a predefined maximum value.
    g. Initialize the Ring structures in DMA structure. For TX BD Ring, set the RunState attr to HALTED, the IsRxChannel to 0, RingIndex attribute to 0 and MaxTransferLen attribute according to whether MicroDmaMode is activated or not. It is to be noted that RingIndex is always 0 for TX BD Ring.
    h. For RX, number of BD Rings depend on number of RX channels(or S2MM channels). So using a for loop, we update the RunState to HALTED for each of these BD Rings, set the IsRxChannel to 1 and RingIndex according to the

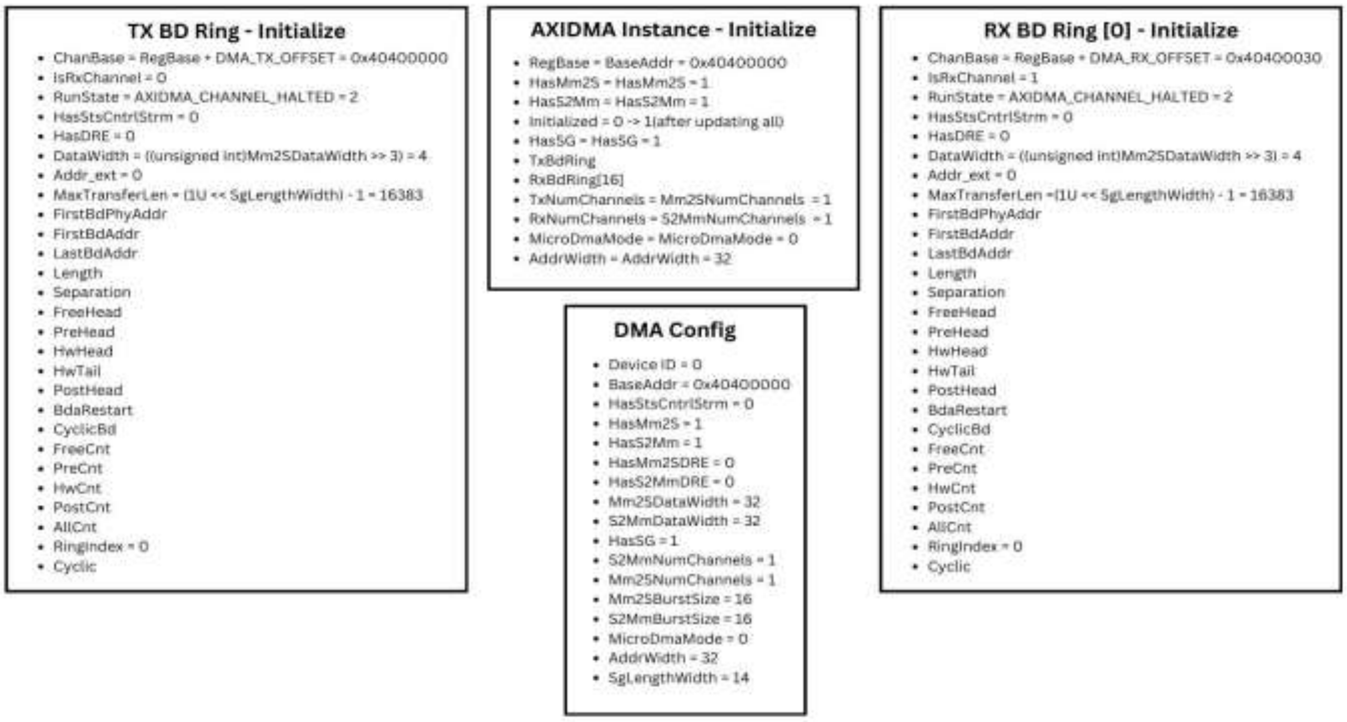
channel number(for loop index). In our case, we use only 1 channel for S2MM so only 1 RX BD Ring.

i. We set the channel base address for both TX and RX BD Ring, based on whether S2MM or MM2S has been set. Alongside, we update the HasStsCntrlStrm attribute, Addr_ext based on whether AddrWidth > 32, HasDRE attribute, DataWidth for both TX and RX BD Rings.
j. Reset the DMA so hardware can start from a known state.

The code is provided below:

```
 int XAxiDma_CfgInitialize(XAxiDma * InstancePtr, XAxiDma_Config
 *Config)
{
	UINTPTR BaseAddr;
	int TimeOut;
	int Index;
	u32 MaxTransferLen;
	InstancePtr->Initialized = 0;
	if(!Config) {
		return XST_INVALID_PARAM;
	}
	BaseAddr = Config->BaseAddr;
	/* Setup the instance */
	memset(InstancePtr, 0, sizeof(XAxiDma));
	InstancePtr->RegBase = BaseAddr;
	/* Get hardware setting information from the configuration
	structure */
	InstancePtr->HasMm2S = Config->HasMm2S;
	InstancePtr->HasS2Mm = Config->HasS2Mm;
	InstancePtr->HasSg = Config->HasSg;
	InstancePtr->MicroDmaMode = Config->MicroDmaMode;
	InstancePtr->AddrWidth = Config->AddrWidth;
	/* Get the number of channels */
	InstancePtr->TxNumChannels = Config->Mm2sNumChannels;
	InstancePtr->RxNumChannels = Config->S2MmNumChannels;
	/* This condition is for IP version < 6.00a */
	if (!InstancePtr->TxNumChannels)
		InstancePtr->TxNumChannels = 1;
	if (!InstancePtr->RxNumChannels)
		InstancePtr->RxNumChannels = 1;
	if ((InstancePtr->RxNumChannels > 1) ||
		(InstancePtr->TxNumChannels > 1)) {
		MaxTransferLen =XAXIDMA_MCHAN_MAX_TRANSFER_LEN;
	}
	else {
		MaxTransferLen = (1U << Config->SgLengthWidth) - 1;
	}
	/* Initialize the ring structures */
	InstancePtr->TxBdRing.RunState = AXIDMA_CHANNEL_HALTED;
	InstancePtr->TxBdRing.IsRxChannel = 0;
	if (!InstancePtr->MicroDmaMode) {
		InstancePtr->TxBdRing.MaxTransferLen = MaxTransferLen;
	}
	else {
		/* In MicroDMA mode, Maximum length that can be
		transferred
		* is '(Memory Data Width / 4) * Burst Size'*/
		InstancePtr->TxBdRing.MaxTransferLen = ((Config-
		>Mm2SDataWidth / 4) * Config->Mm2SBurstSize);
	}
	InstancePtr->TxBdRing.RingIndex = 0;
	for (Index = 0; Index < InstancePtr->RxNumChannels; Index++) {
		InstancePtr->RxBdRing[Index].RunState =
		AXIDMA_CHANNEL_HALTED;
		InstancePtr->RxBdRing[Index].IsRxChannel = 1;
		InstancePtr->RxBdRing[Index].RingIndex = Index;
	}
	if (InstancePtr->HasMm2S) {
		InstancePtr->TxBdRing.ChanBase = BaseAddr +
		XAXIDMA_TX_OFFSET;
		InstancePtr->TxBdRing.HasStsCntrlStrm = Config-
		>HasStsCntrlStrm;
		if (InstancePtr->AddrWidth > 32)
			InstancePtr->TxBdRing.Addr_ext = 1;
		else
			InstancePtr->TxBdRing.Addr_ext = 0;

		InstancePtr->TxBdRing.HasDRE = Config->HasMm2SDRE;
		InstancePtr->TxBdRing.DataWidth = ((unsigned
		int)Config->Mm2SDataWidth >> 3);
	}
	if (InstancePtr->HasS2Mm) {
		for (Index = 0;
			Index < InstancePtr->RxNumChannels; Index++) {
			InstancePtr->RxBdRing[Index].ChanBase =
					BaseAddr + XAXIDMA_RX_OFFSET;
			InstancePtr->RxBdRing[Index].HasStsCntrlStrm =
					Config->HasStsCntrlStrm;
			InstancePtr->RxBdRing[Index].HasDRE =
					Config->HasS2MmDRE;
			InstancePtr->RxBdRing[Index].DataWidth =
			((unsigned int)Config->S2MmDataWidth >> 3);
			if (!InstancePtr->MicroDmaMode) {
				InstancePtr-
				>RxBdRing[Index].MaxTransferLen =
				MaxTransferLen;
			}
			else {
			/* In MicroDMA mode, Maximum length that can be
			transferred
			* is '(Memory Data Width / 4) * Burst Size' */
				InstancePtr-
				>RxBdRing[Index].MaxTransferLen =
				((Config->S2MmDataWidth / 4) *
				Config->S2MmBurstSize);
			}
			if (InstancePtr->AddrWidth > 32)
				InstancePtr->RxBdRing[Index].Addr_ext = 1;
			else
				InstancePtr->RxBdRing[Index].Addr_ext = 0;
		}
	}

	/* Reset the engine so the hardware starts from a known st */
	XAxiDma_Reset(InstancePtr);
	/* At the initialization time, hardware should finish reset
	quickly*/
	TimeOut = XAXIDMA_RESET_TIMEOUT;
	while (TimeOut) {
		if(XAxiDma_ResetIsDone(InstancePtr)) {
			break;
		}
		TimeOut -= 1;
	}
	if (!TimeOut) {
		xdbg_printf(XDBG_DEBUG_ERROR, "Failed reset in"
					"initialize\r\n");
		/* Need system hard reset to recover*/
		InstancePtr->Initialized = 0;
		return XST_DMA_ERROR;
	}
	/* Initialization is successful*/
```

```
        InstancePtr->Initialized = 1;
        return XST_SUCCESS;
    }
```



TX BD Ring - Initialize

- ChanBase = RegBase + DMA_TX_OFFSET = 0x40400000
- IsRxChannel = 0
- RunState = AXIDMA_CHANNEL_HALTED = 2
- HasStsCntrlStrm = 0
- HasDRE = 0
- DataWidth = ((unsigned int)Mm2SDataWidth >> 3) = 4
- Addr_ext = 0
- MaxTransferLen = (1U << SgLengthWidth) - 1 = 16383
- FirstBdPhyAddr
- FirstBdAddr
- LastBdAddr
- Length
- Separation
- FreeHead
- PreHead
- HwHead
- HwTail
- PostHead
- BdaRestart
- CyclicBd
- FreeCnt
- PreCnt
- HwCnt
- PostCnt
- AllCnt
- RingIndex = 0
- Cyclic

AXIDMA Instance - Initialize

- RegBase = BaseAddr = 0x40400000
- HasMm2S = HasMm2S = 1
- HasS2Mm = HasS2Mm = 1
- Initialized = 0 -> 1(after updating all)
- HasSG = HasSG = 1
- TxBdRing
- RxBdRing[16]
- TxNumChannels = Mm2SNumChannels = 1
- RxNumChannels = S2MmNumChannels = 1
- MicroDmaMode = MicroDmaMode = 0
- AddrWidth = AddrWidth = 32

DMA Config

- Device ID = 0
- BaseAddr = 0x40400000
- HasStsCntrlStrm = 0
- HasMm2S = 1
- HasS2Mm = 1
- HasMm2SDRE = 0
- HasS2MmDRE = 0
- Mm2SDataWidth = 32
- S2MmDataWidth = 32
- HasSG = 1
- S2MmNumChannels = 1
- Mm2SNumChannels = 1
- Mm2SBurstSize = 16
- S2MmBurstSize = 16
- MicroDmaMode = 0
- AddrWidth = 32
- SgLengthWidth = 14

RX BD Ring [0] - Initialize

- ChanBase = RegBase + DMA_RX_OFFSET = 0x40400030
- IsRxChannel = 1
- RunState = AXIDMA_CHANNEL_HALTED = 2
- HasStsCntrlStrm = 0
- HasDRE = 0
- DataWidth = ((unsigned int)Mm2SDataWidth >> 3) = 4
- Addr_ext = 0
- MaxTransferLen =(1U << SgLengthWidth) - 1 = 16383
- FirstBdPhyAddr
- FirstBdAddr
- LastBdAddr
- Length
- Separation
- FreeHead
- PreHead
- HwHead
- HwTail
- PostHead
- BdaRestart
- CyclicBd
- FreeCnt
- PreCnt
- HwCnt
- PostCnt
- AllCnt
- RingIndex = 0
- Cyclic

After initializing the DMA, these are the updated structures based on values from DMA config.

4. **DMA IN SG MODE**
   Once the DMA is initialized, we need to make sure that the DMA is configured in SG mode. It is evident from the above picture that HasSG attribute for the given DMA is set to 1

5. **TX CHANNEL SETUP FOR DMA**
   Now we move onto the TX Setup. This function sets up the TX channel of the DMA for packet transmission. The parameter for the function is the pointer to the AxiDma engine.

   - First, we take the pointer to TX BD Ring from the DMA engine and store the value in TxRingPtr. The function returns the address for TxBdRing.

     ```
     ➔ TxRingPtr = XAxiDma_GetRing(&AxiDma);

     #define XAxiDma_GetTxRing(InstancePtr)\ (&((InstancePtr)->TxBdRing))
     ```

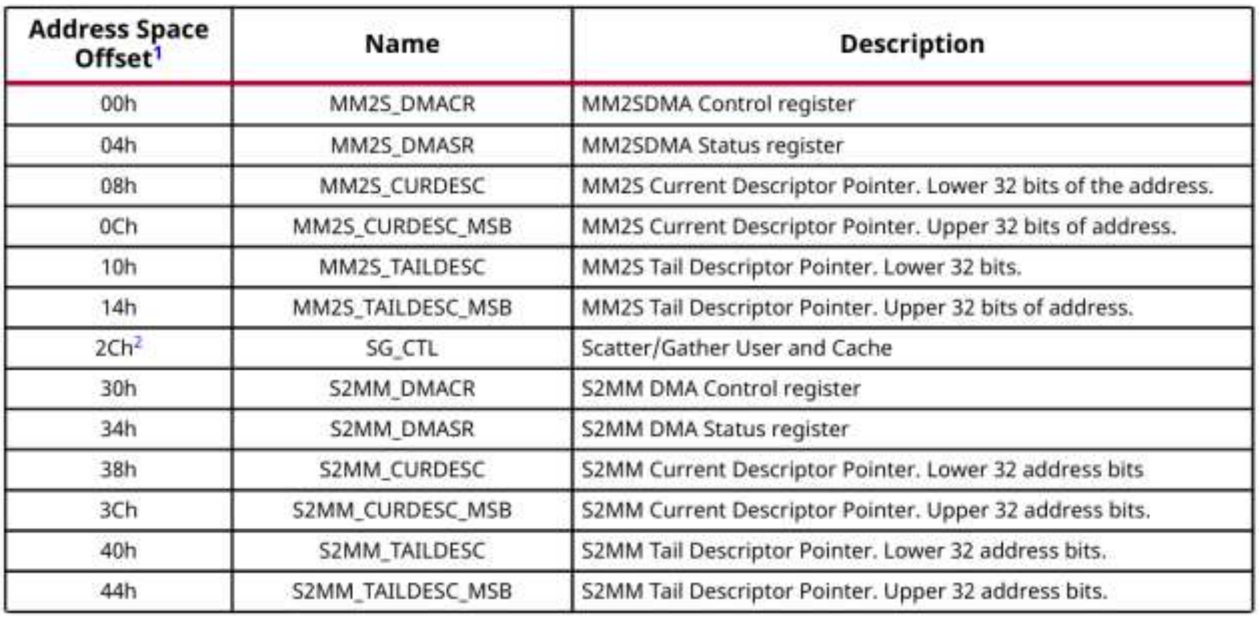
   - Before setting up the TX Buffer Descriptor (BD) space, it is important to disable all interrupt bits for the channel as specified by the mask for the TX BD ring. The mask used for this purpose is XAXIDMA_IRQ_MASK, which has a value of 0x00007000. This mask disables the following interrupts: the Interrupt on Completion (IOC) interrupt, the delay interrupt, and the error interrupt.

     The function reads the current value from the control register of the DMA register address map for the transmit (MM2S) channel. It then clears the relevant interrupt bits by performing a bitwise AND operation with the negation of the mask (~Mask). Finally, it writes the modified value back to the control register.

     ```
     #define XAXIDMA_CR_OFFSET   0x00000000   /**< Channel control */

     ➔ XAxiDma_BdRingIntDisable(TxRingPtr, XAXIDMA_IRQ_ALL_MASK);
     ```

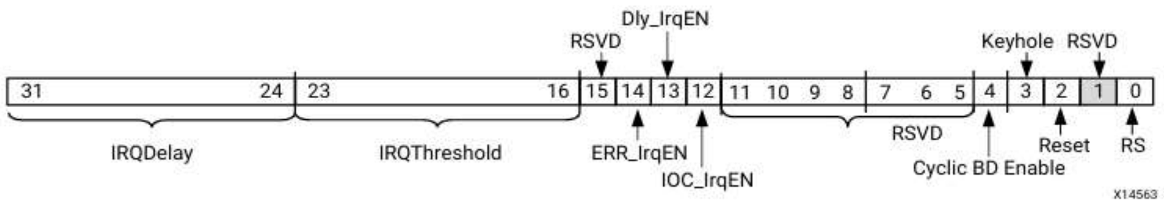
```
#define XAxiDma_BdRingIntDisable(RingPtr, Mask)                \
      (XAxiDma_WriteReg((RingPtr)->ChanBase, XAXIDMA_CR_OFFSET, \
      XAxiDma_ReadReg((RingPtr)->ChanBase, XAXIDMA_CR_OFFSET) & \
      ~ ((Mask) & XAXIDMA_IRQ_ALL_MASK)))
```

DMA REGISTER ADDRESS SPACE



Address Space Offset[1]	Name	Description
00h	MM2S_DMACR	MM2SDMA Control register
04h	MM2S_DMASR	MM2SDMA Status register
08h	MM2S_CURDESC	MM2S Current Descriptor Pointer. Lower 32 bits of the address.
0Ch	MM2S_CURDESC_MSB	MM2S Current Descriptor Pointer. Upper 32 bits of address.
10h	MM2S_TAILDESC	MM2S Tail Descriptor Pointer. Lower 32 bits.
14h	MM2S_TAILDESC_MSB	MM2S Tail Descriptor Pointer. Upper 32 bits of address.
2Ch[2]	SG_CTL	Scatter/Gather User and Cache
30h	S2MM_DMACR	S2MM DMA Control register
34h	S2MM_DMASR	S2MM DMA Status register
38h	S2MM_CURDESC	S2MM Current Descriptor Pointer. Lower 32 address bits
3Ch	S2MM_CURDESC_MSB	S2MM Current Descriptor Pointer. Upper 32 address bits.
40h	S2MM_TAILDESC	S2MM Tail Descriptor Pointer. Lower 32 address bits.
44h	S2MM_TAILDESC_MSB	S2MM Tail Descriptor Pointer. Upper 32 address bits.

We updated the Offset 00h register of DMA, for disabling the interrupts.



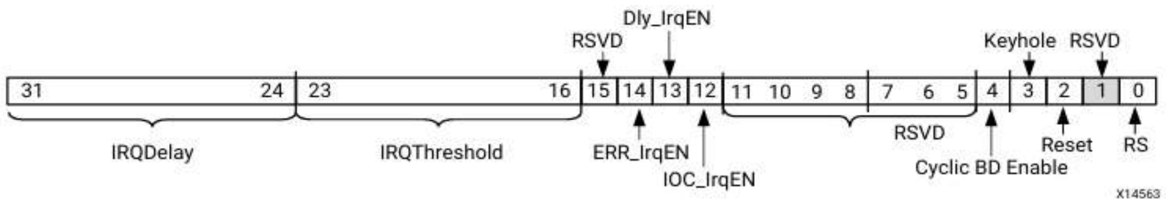
The interrupt fields are bits 12, 13, 14 of the control register MM2S_DMACR.

- We need to set the TX timer and counter for the DMA instance. The Timer value is set to 0, and the counter is set to 1. These values correspond to fields in the control register for both the TX and RX channels of the DMA instance. Since these are 8-bit fields, they can take values from 0 to 255.
  The counter value (coalesce) determines the interrupt threshold, which is used to generate an interrupt after an Interrupt on Completion (IOC) event occurs. Similarly, the delay value is used to set the interrupt delay, which plays a role in the interrupt timeout mechanism, generating an interrupt after the specified delay period has expired, starting from the end of the packet. Since we are not utilizing this mode, we can set the delay value to 0.
  Thus, updating the control register for IRQ Delay and Threshold for the TX BD ring is accomplished by this function.

```
int Delay = 0;
int Coalesce = 1;
➔ XAxiDma_BdRingSetCoalesce(TxRingPtr, Coalesce, Delay);

int XAxiDma_BdRingSetCoalesce(XAxiDma_BdRing *RingPtr, u32 Counter,
u32 Timer)
{
      u32 Cr;
      Cr = XAxiDma_ReadReg(RingPtr->ChanBase, XAXIDMA_CR_OFFSET);
```

```
        if (Counter != XAXIDMA_NO_CHANGE) {
                if ((Counter == 0) || (Counter > 0xFF)) {
                        xdbg_printf(XDBG_DEBUG_ERROR,"BdRingSetCoalesce:”
                        "invalid  coalescing threshold %d",
                        (int)Counter);
                        return XST_FAILURE;
                }
                Cr = (Cr & ~XAXIDMA_COALESCE_MASK) |
                        (Counter << XAXIDMA_COALESCE_SHIFT);
        }

        if (Timer != XAXIDMA_NO_CHANGE) {
                if (Timer > 0xFF) {
                        xdbg_printf(XDBG_DEBUG_ERROR,"BdRingSetCoalesce:"
                        "invalid  delay counter %d", (int)Timer);
                        return XST_FAILURE;
                }

                Cr = (Cr & ~XAXIDMA_DELAY_MASK) |
                (Timer << XAXIDMA_DELAY_SHIFT);
        }
        XAxiDma_WriteReg(RingPtr->ChanBase, XAXIDMA_CR_OFFSET, Cr);
        return XST_SUCCESS;
}
```

ChanBase attribute of the DMA BD Ring stores the base address for that channel, whether it is TX or RX. With respect to this address, we write to various registers of the DMA like control register, status register, current descriptor pointer, tail descriptor pointer etc. `XAXIDMA_CR_OFFSET` stores a value of 0x00000000 since from the above DMA register address map table, we saw that control register has an offset of 00h from base address (MM2S_DMACR). So in the code, we read from the control register for specific channel, here TX, and updated the IRQ threshold and Delay part with user specified values and rewritten back to the control register.



Bits 16 to 23 is updated with counter and bits 24 to 31 updated with Delay value.

- Now we need to calculate the total number of Block Descriptors (BDs) that can be stored in the allocated memory for the TX BDs. In other words, we want to determine how many BDs will fit within the given memory constraints. To achieve this, we use a BD count calculator function called XAxiDma_BdRingCntCalc.
  This function takes two parameters: **Alignment** and **Total Available Bytes**. The Alignment parameter defines the byte alignment that each BD will follow, while the Total Available Bytes parameter specifies the total amount of memory allocated for storing BD information. Essentially, this function calculates how many BDs can be created from the total memory allocated for the TX BD space.

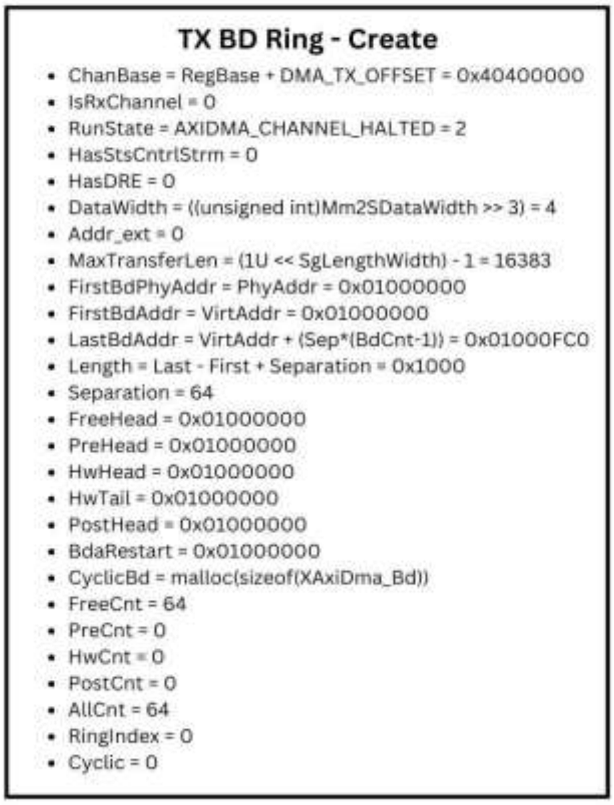
```
#define TX_BD_SPACE_BASE   (MEM_BASE_ADDR)
#define TX_BD_SPACE_HIGH   (MEM_BASE_ADDR + 0x00000FFF)

➔  BdCount = XAxiDma_BdRingCntCalc(XAXIDMA_BD_MINIMUM_ALIGNMENT,
                          TX_BD_SPACE_HIGH - TX_BD_SPACE_BASE + 1);
#define XAxiDma_BdRingCntCalc(Alignment, Bytes)              \
        (uint32_t)((Bytes)/((sizeof(XAxiDma_Bd)+((Alignment)-
        1))&~((Alignment)-1)))
```

The memory space for TX BD is 0xFFF + 1 which is 4096 bytes. The parameter alignment is passed with value XAXIDMA_BD_MINIMUM_ALIGNMENT, which has a value of 0x40 or 64. Size of XAxiDma_Bd must be 16 as it comprises of 32 bit 16 words, thus occupying a total of 64 bytes. This function will return value 64 as 64 BDs each occupying 64 bytes can be made from 4096 bytes.

- Now, we create the TX BD Ring. To create the TX BD (Buffer Descriptor) Ring, we use the XAxiDma_BdRingCreate function. This function takes several parameters: the BD Ring instance (a pointer to either the TX or RX BD Ring), the physical address, the virtual address of the application memory region, the alignment of the BD, and the count of BDs (BdCount). The function essentially initializes the BD ring structure by resetting all count values, clearing the BD space, updating the next descriptor field for each BD, and configuring hardware attributes such as the status control stream and DRE registers for all BDs in the ring.
  The XAxiDma_BdRingCreate function performs the following key steps:
  I. Resets count values like AllCnt, FreeCnt, HwCnt, PreCnt, and PostCnt in the XAxiDma_BdRing structure to ensure proper initialization.
  II. Checks if the alignment meets the minimum requirement and ensures that it is a power of two. Also, the physical and virtual addresses must align correctly.
  III. The function links each BD to the next one using physical addresses, with the last BD linking back to the first, forming a circular ring.
  IV. Writes hardware-specific settings into each BD, such as status control and DRE settings.
  V. Updates all attributes of the XAxiDma_BdRing structure, including the start and end addresses of the BD ring, the total size, and the counters for different BD groups (free, pre-work, hardware work, post-work).

```
➔ Status = XAxiDma_BdRingCreate(TxRingPtr, TX_BD_SPACE_BASE,
       TX_BD_SPACE_BASE,XAXIDMA_BD_MINIMUM_ALIGNMENT, BdCount);

u32 XAxiDma_BdRingCreate(XAxiDma_BdRing *RingPtr, UINTPTR PhysAddr,
        UINTPTR VirtAddr, u32 Alignment, int BdCount)
{
        int i;
        UINTPTR BdVirtAddr;
        UINTPTR BdPhysAddr;
        if (BdCount <= 0) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingCreate: non-
                positive BD number %d\r\n", BdCount);
                return XST_INVALID_PARAM;
        }
        RingPtr->AllCnt = 0;
        RingPtr->FreeCnt = 0;
        RingPtr->HwCnt = 0;
        RingPtr->PreCnt = 0;
        RingPtr->PostCnt = 0;
        RingPtr->Cyclic = 0;
        if (Alignment < XAXIDMA_BD_MINIMUM_ALIGNMENT) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingCreate: alignment
                too small %d, need to be at least %d\r\n",
                (int)Alignment,XAXIDMA_BD_MINIMUM_ALIGNMENT);
                return XST_INVALID_PARAM;
        }
        if ((Alignment - 1) & Alignment) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingCreate: alignment
                not valid %d\r\n", (int)Alignment);
                return XST_INVALID_PARAM;
        }
        if ((PhysAddr % Alignment) || (VirtAddr % Alignment)) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingCreate: Physical
                address %x and virtual address %x have different
                alignment\r\n",(unsigned int)PhysAddr, (unsigned
                int)VirtAddr);
                return XST_INVALID_PARAM;
        }
        RingPtr->Separation =
                (sizeof(XAxiDma_Bd) +(Alignment - 1))& ~(Alignment -1);
        if (VirtAddr >(VirtAddr + (RingPtr->Separation * BdCount)-1)){
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingCreate: BD space
                cross 0x0\r\n");
                return XST_DMA_SG_LIST_ERROR;
        }
        memset((void *) VirtAddr, 0, (RingPtr->Separation * BdCount));
        BdVirtAddr = VirtAddr;
        BdPhysAddr = PhysAddr + RingPtr->Separation;
        for (i = 1; i < BdCount; i++) {
                XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_NDESC_OFFSET,
                                (BdPhysAddr & XAXIDMA_DESC_LSB_MASK));
                XAxiDma_BdWrite(BdVirtAddr,
                XAXIDMA_BD_NDESC_MSB_OFFSET,UPPER_32_BITS(BdPhysAddr));
                /* Put hardware information in the BDs
                */
                XAxiDma_BdWrite(BdVirtAddr,
                XAXIDMA_BD_HAS_STSCNTRL_OFFSET,(u32)RingPtr>HasStsCntrl
                Strm);

                XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_HAS_DRE_OFFSET,
                (((u32)(RingPtr->HasDRE)) << XAXIDMA_BD_HAS_DRE_SHIFT)|
                RingPtr->DataWidth);

                XAXIDMA_CACHE_FLUSH(BdVirtAddr);
                BdVirtAddr += RingPtr->Separation;
                BdPhysAddr += RingPtr->Separation;
        }
        XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_NDESC_OFFSET,
                        (PhysAddr & XAXIDMA_DESC_LSB_MASK));
        XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_NDESC_MSB_OFFSET,
                        UPPER_32_BITS(PhysAddr));
        XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_HAS_STSCNTRL_OFFSET,
                (u32)RingPtr->HasStsCntrlStrm);

        XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_HAS_DRE_OFFSET,
                (((u32)(RingPtr->HasDRE)) << XAXIDMA_BD_HAS_DRE_SHIFT)|
                RingPtr->DataWidth);
        RingPtr->RunState = AXIDMA_CHANNEL_HALTED;
        RingPtr->FirstBdAddr = VirtAddr;
        RingPtr->FirstBdPhysAddr = PhysAddr;
        RingPtr->LastBdAddr = BdVirtAddr;
        RingPtr->Length = RingPtr->LastBdAddr - RingPtr->FirstBdAddr +
                RingPtr->Separation;
        RingPtr->AllCnt = BdCount;
        RingPtr->FreeCnt = BdCount;
        RingPtr->FreeHead = (XAxiDma_Bd *) VirtAddr;
        RingPtr->PreHead = (XAxiDma_Bd *) VirtAddr;
        RingPtr->HwHead = (XAxiDma_Bd *) VirtAddr;
        RingPtr->HwTail = (XAxiDma_Bd *) VirtAddr;
        RingPtr->PostHead = (XAxiDma_Bd *) VirtAddr;
        RingPtr->BdaRestart = (XAxiDma_Bd *) VirtAddr;
        RingPtr->CyclicBd = (XAxiDma_Bd *) malloc(sizeof(XAxiDma_Bd));
        return XST_SUCCESS;
}
-------------------------------------------------------------------
```

The TX BD Ring after creating will look like the figure given below. We can see that for this example, a total of 64 BDs are created and all are in Free group. Also we modified the Header pointers for all groups with the address of first BD.


TX BD Ring - Create

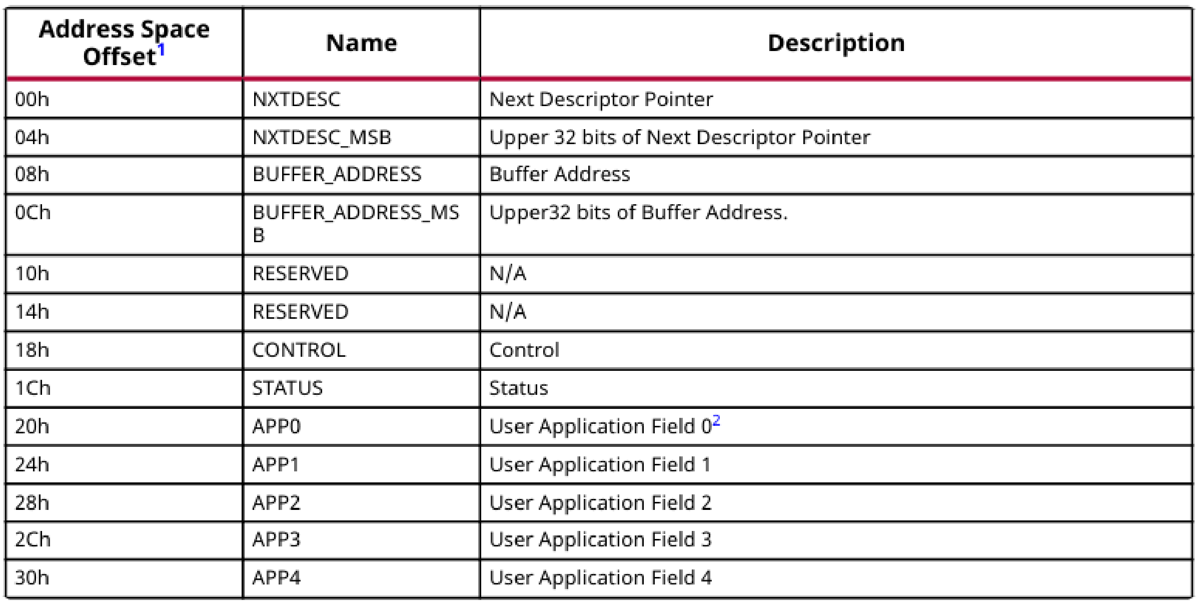
- ChanBase = RegBase + DMA_TX_OFFSET = 0x40400000
- IsRxChannel = 0
- RunState = AXIDMA_CHANNEL_HALTED = 2
- HasStsCntrlStrm = 0
- HasDRE = 0
- DataWidth = ((unsigned int)Mm2SDataWidth >> 3) = 4
- Addr_ext = 0
- MaxTransferLen = (1U << SgLengthWidth) - 1 = 16383
- FirstBdPhyAddr = PhyAddr = 0x01000000
- FirstBdAddr = VirtAddr = 0x01000000
- LastBdAddr = VirtAddr + (Sep*(BdCnt-1)) = 0x01000FC0
- Length = Last - First + Separation = 0x1000
- Separation = 64
- FreeHead = 0x01000000
- PreHead = 0x01000000
- HwHead = 0x01000000
- HwTail = 0x01000000
- PostHead = 0x01000000
- BdaRestart = 0x01000000
- CyclicBd = malloc(sizeof(XAxiDma_Bd))
- FreeCnt = 64
- PreCnt = 0
- HwCnt = 0
- PostCnt = 0
- AllCnt = 64
- RingIndex = 0
- Cyclic = 0

Explanation for the above code is:

- Since the address translation is not used here, we keep the virtual address parameter same as physical address, which is set to TX_BD_SPACE_BASE.
- The alignment field is set to 64, which is the required minimum value and BdCount set to the value found in previous function.
- First, we check the BdCount is a valid number, followed by alignment check on whether its greater than the required minimum.
- Also make sure that alignment is a power of 2 and both physical and virtual address follows same alignment.
- BD Ring pointer attribute, Separation, stores how many bytes are used by each BD in BD Ring which is 64.
- Using memset function, we set all memory locations for storing TX BDs to 0.
- Now update the next descriptor registers (LSB and MSB) for each BD with physical address of next BD(actually 1st BD phy addr + separation = 2nd BD phy addr).
- Along with that, we update the status control stream and DRE registers of each BDs.
- This is accomplished using a for loop and it also made sure that the last BD next descriptor field is updated with physical address of 1st BD to keep the BD Ring in cyclic mode.
- Finally, we set the counter and pointers to BD Ring instance Structure. The structure and its attributes are –

  ```
  typedef struct {
  	UINTPTR ChanBase;		/**< physical base address*/
  	int IsRxChannel;	/**< Is this a receive channel */
  	volatile int RunState;/**< Whether chnl is running */
  	int HasStsCntrlStrm;	/**<has stscntrl stream */
  	int HasDRE;
  	int DataWidth;
  	int Addr_ext;
  	u32 MaxTransferLen;
  	UINTPTR FirstBdPhysAddr;/*Phy addr of 1st BD in list */
  	UINTPTR FirstBdAddr;/*Vir addr of 1st BD in list */
  	UINTPTR LastBdAddr; /*Vir addr of last BD in the list */
  ```

```
       u32 Length;         /*Total size of ring in bytes */
       UINTPTR Separation; /*No of bytes between the starting
                            address of adjacent BDs */
       XAxiDma_Bd *FreeHead;/*First BD in the free group */
       XAxiDma_Bd *PreHead;/*First BD in the pre-work group */
       XAxiDma_Bd *HwHead; /**< First BD in the work group */
       XAxiDma_Bd *HwTail; /**< Last BD in the work group */
       XAxiDma_Bd *PostHead;/*First BD in post-work group */
       XAxiDma_Bd *BdaRestart;/*BD to load when chnl started*/
       XAxiDma_Bd *CyclicBd;/**< Useful for Cyclic DMA op*/
       int FreeCnt;/**< No of allocatable BDs in free group */
       int PreCnt;/**< No of BDs in pre-work group */
       int HwCnt;/**< Number of BDs in work group */
       int PostCnt;/**< Number of BDs in post-work group */
       int AllCnt;/**< Total Number of BDs for channel */
       int RingIndex;/**< Ring Index */
       int Cyclic;/**< Check for cyclic DMA Mode */
} XAxiDma_BdRing;
--------------------------------------------------------------
```



Address Space Offset[1]	Name	Description
00h	NXTDESC	Next Descriptor Pointer
04h	NXTDESC_MSB	Upper 32 bits of Next Descriptor Pointer
08h	BUFFER_ADDRESS	Buffer Address
0Ch	BUFFER_ADDRESS_MSB	Upper32 bits of Buffer Address.
10h	RESERVED	N/A
14h	RESERVED	N/A
18h	CONTROL	Control
1Ch	STATUS	Status
20h	APP0	User Application Field 0[2]
24h	APP1	User Application Field 1
28h	APP2	User Application Field 2
2Ch	APP3	User Application Field 3
30h	APP4	User Application Field 4

We updated registers 00h, 04h, 38h, and 3Ch in the above part. Last 3 words are not shown in the above table, which is basically ID register (34h), Has_stscntrl register (38h) and Has_dre register (3Ch).

- Finally, we update the BDRing structure attributes. Since we haven't started the channel yet, we keep the attribute RunState as HALTED. Also we update the 1st BD address, physical BD address with that of 1st BD – TX_BD_SPACE_BASE. The last BD address is that of 64th BD. Attribute Length is equivalent to 4096 bytes and we set the allcnt and freecnt to total BDs available. Update rest of the pointers with address to 1st BD.

- After creating the BD (Buffer Descriptor) Ring, the next step is to initialize the memory assigned to the BD Ring, except for the fields that were already updated during the ring creation. To accomplish this, we create a template BD with all fields set to zero (except for the ones that were updated in the BD Ring creation). This template BD will be cloned into the entire BD list for consistency.
  We use the function XAxiDma_BdClear() to zero out the BD template. The template BD is of type u32 and consists of 16 words. During the BD ring creation, certain fields (specifically words 1, 2, 15, and 16) were already initialized, so the remaining words (3 to 14) need to be set to zero in the template BD.

```
------------------------------------------------------------------------------------------------
XAxiDma_Bd BdTemplate;

➔ XAxiDma_BdClear(&BdTemplate);
```

```
#define XAxiDma_BdClear(BdPtr)                          \
        memset((void *)(((UINTPTR)(BdPtr))+XAXIDMA_BD_START_CLEAR),0,\
        XAXIDMA_BD_BYTES_TO_CLEAR)
```

XAxiDma_Bd is a structure type representing a BD, consisting of 16 32-bit words. We create a variable BdTemplate of type XAxiDma_Bd and pass the **address** of this template BD to the XAxiDma_BdClear() function. In the function, the macro uses memset() to zero out memory starting from XAXIDMA_BD_START_CLEAR (which is word 8) to the sum of XAXIDMA_BD_START_CLEAR and XAXIDMA_BD_BYTES_TO_CLEAR (i.e., 8 + 48 = 56 bytes). This effectively clears all words between 3 and 14, leaving words 1, 2, 15, and 16 untouched. This ensures the BDs in the ring are properly initialized with zeros where needed, while preserving the essential fields already configured during the BD ring creation.

- Once the template BD has been created and initialized, the next step is to clone it to all the BDs in the BD Ring. This is done using the XAxiDma_BdRingClone() function, which takes two parameters which are pointer to BD Ring and template BD that will be cloned to all BDs in the BD Ring.

NOTE: It is crucial to call XAxiDma_BdRingClone() only when all BDs are in the free group, meaning that none of the BDs have been allocated yet for hardware (HW) operations. This ensures that the cloning process does not modify any BDs currently in use by hardware, which could otherwise lead to data corruption or inconsistencies during data transfers.

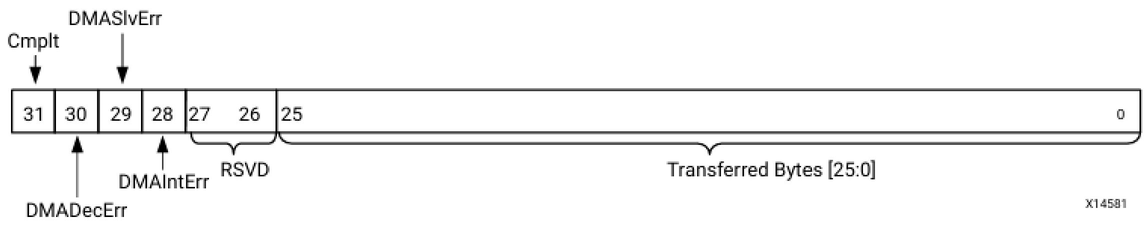
```
Status = XAxiDma_BdRingClone(TxRingPtr, &BdTemplate);
int XAxiDma_BdRingClone(XAxiDma_BdRing * RingPtr, XAxiDma_Bd *
        SrcBdPtr)
{
        int i;
        UINTPTR CurBd;
        u32 Save;
        XAxiDma_Bd TmpBd;
        /* Can't do this function if there isn't a ring */
        if (RingPtr->AllCnt == 0) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingClone: no
                bds\r\n");
                return XST_DMA_SG_NO_LIST;
        }
        /* Can't do this function with the channel running */
        if (RingPtr->RunState == AXIDMA_CHANNEL_NOT_HALTED) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingClone: bd ring
                started already, cannot do\r\n");
                return XST_DEVICE_IS_STARTED;
        }
        /* Can't do this function with some of the BDs in use */
        if (RingPtr->FreeCnt != RingPtr->AllCnt) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingClone: some bds
                already in use %d/%d\r\n",RingPtr->FreeCnt, RingPtr-
                >AllCnt);
                return XST_DMA_SG_LIST_ERROR;
        }
        /* Make a copy of the template then modify it by clearing
         * the complete bit in status/control field
         */
        memcpy(&TmpBd, SrcBdPtr, sizeof(XAxiDma_Bd));

        Save = XAxiDma_BdRead(&TmpBd, XAXIDMA_BD_STS_OFFSET);
        Save &= ~XAXIDMA_BD_STS_COMPLETE_MASK;
        XAxiDma_BdWrite(&TmpBd, XAXIDMA_BD_STS_OFFSET, Save);

        for (i = 0, CurBd = RingPtr->FirstBdAddr;
                i < RingPtr->AllCnt; i++, CurBd += RingPtr->Separation) {
                  memcpy((void *)((UINTPTR)CurBd+XAXIDMA_BD_START_CLEAR),
                         (void *)((UINTPTR)(&TmpBd) +
                         XAXIDMA_BD_START_CLEAR),XAXIDMA_BD_BYTES_TO_CLEAR
                         );
                  XAXIDMA_CACHE_FLUSH(CurBd);
        }
        return XST_SUCCESS;
}
```

Before cloning the template BD across the entire BD Ring, the code first ensures that a valid BD Ring is being passed by checking the number of BDs in the ring. If the BD count is incorrect or invalid, the cloning operation is terminated to avoid any errors. Additionally, the code verifies that the BD Ring is in the **HALTED** state, meaning it is not currently in the **RUNNING** state. This is critical because modifying a BD Ring while it is active could interfere with on-going hardware operations, potentially causing data corruption or malfunction.

Next, the code checks if all available BDs are in the **Free group**. This is necessary because if any of the BDs have already been allocated to the **Pre-Work group**, indicating they are prepared for hardware operations, they should not be modified. Making changes to BDs that are already in use by the hardware can compromise data integrity, so the cloning process must only proceed when all BDs are unallocated and available in the Free group.

After passing these checks, a copy of the template BD is created, and its **status register** is modified to set the **complete bit** to 0 using a mask. This ensures that the cloned BDs are reset and ready for new transactions. Finally, the registers of each BD in the BD Ring are updated by copying the corresponding register values from the template BD, ensuring consistent initialization across the entire ring.

In the status register, bit 31 corresponds to complete flag for BD:



- The final step in setting up the TX channel is to start the BD Ring. This involves updating the current descriptor in the DMA register address map for the given channel and initiating the channel's operation. To achieve this, we pass the pointer to the channel instance (in this case, the TX channel) into the function responsible for starting the BD Ring.

```
➔ Status = XAxiDma_BdRingStart(TxRingPtr);

int XAxiDma_BdRingStart(XAxiDma_BdRing * RingPtr)
{
        int Status;
        Status = XAxiDma_UpdateBdRingCDesc(RingPtr);
        if (Status != XST_SUCCESS) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingStart: "
               "Updating Current Descriptor Failed\n\r");
               return Status;
        }
        Status = XAxiDma_StartBdRingHw(RingPtr);
        if (Status != XST_SUCCESS) {
              xdbg_printf(XDBG_DEBUG_ERROR, "BdRingStart: "
              "Starting Hardware Failed\n\r");
              return Status;
        }
        return XST_SUCCESS;
}
-----------------------------------------------------------------
```

Two key functions are called as part of the process:

I. XAxiDma_UpdateBdRingCDesc(): This function updates the current descriptor in the DMA register address map, which points to the start of the BD Ring. It ensures that the DMA engine is aware of the memory address of the first BD in the ring, setting the context for upcoming data transactions.

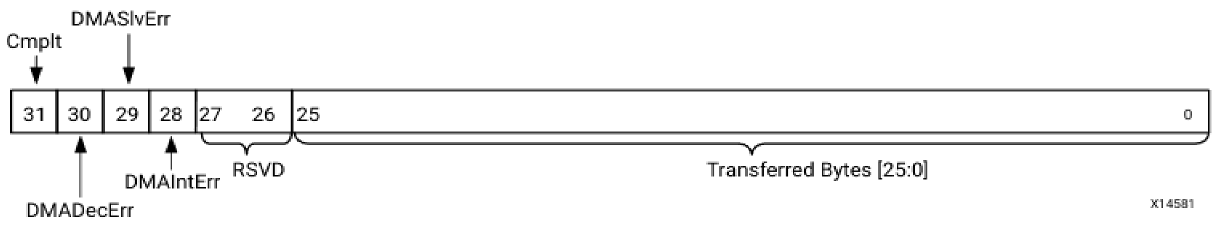
II. XAxiDma_StartBdRingHw():This function starts the hardware execution of the BD Ring by enabling the channel. However, at this stage, the channel is not yet ready to transfer data.

NOTE: It is to be noted that DMA will not immediately start sending data, even though we update the current descriptor and start channel. Transmission will happen only after writing to Tail Descriptor register in DMA.

Let's look into more detail, on each of these functions:

I. XAxiDma_UpdateBdRingCDesc() function takes in the (TX) BD Ring structure as parameter and will set the current descriptor of that particular channel.

General flow looks like, 1st we check if the TX BD Ring is valid by looking into AllCnt attribute, which indicates how many total BDs are allocated for this TX BD Ring. We will write into current descriptor only if the channel is not started. It is checked by looking into the RunState attribute of the structure. If the TX BD Ring is already running, then we need not update the current descriptor for that corresponding channel (TX) or MM2S channel.
If it is not started yet (which is out current case), then we store the Channel Base address into a register called RegBase and the BdaRestart attribute value into BdPtr register. BdaRestart is a pointer to the BD which should be loaded when the channel is started. In our case, it points to first BD address. Now we have 2 possibilities, either the current BD (BdPtr) has completed its implementation or not. In our case, it's a fresh BD and its complete bit In status reg will be 0 which is bit 31. This is BD register and not DMA register.

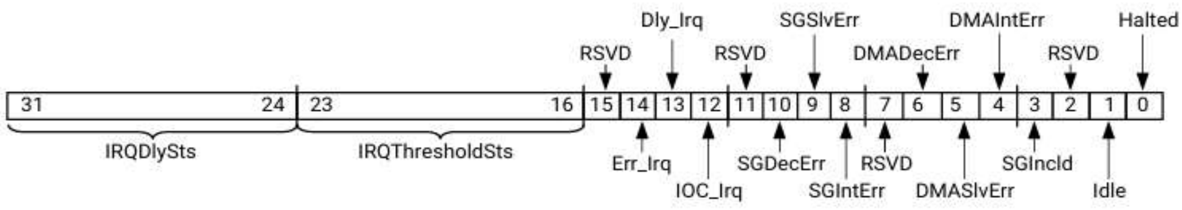


We now check if it's RX or TX channel, on which it will pass **else** condition for **if else** condition checked with attribute IsRxChannel. Then with respect to the channel base address of DMA TX channel (MM2S_DMACR), we update the current descriptor register (BASE + 08h) and (BASE + 0Ch) registers [MM2S_CURDESC and MM2S_CURDESC_MSB]. The MSB address is updated based on whether Addr_Ext attribute has been set or not, depending on address width set from HDL DMA design.

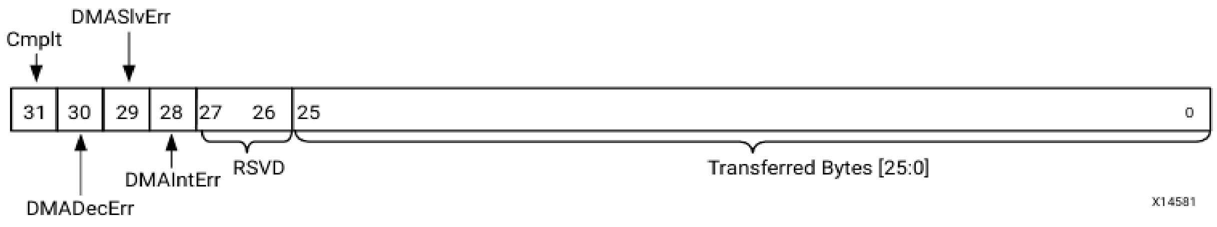
```
#define XAxiDma_BdRingHwIsStarted(RingPtr)                        \
        ((XAxiDma_ReadReg((RingPtr)->ChanBase, XAXIDMA_SR_OFFSET) \
        & XAXIDMA_HALTED_MASK) ? FALSE : TRUE)
#define XAxiDma_BdHwCompleted(BdPtr)                         \
        (XAxiDma_BdRead((BdPtr), XAXIDMA_BD_STS_OFFSET) &
        XAXIDMA_BD_STS_COMPLETE_MASK)

int XAxiDma_UpdateBdRingCDesc(XAxiDma_BdRing* RingPtr)
{
        UINTPTR RegBase;
        UINTPTR BdPtr;
        int RingIndex = RingPtr->RingIndex;
        /* BD list has yet to be created for this channel */
        if (RingPtr->AllCnt == 0) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingStart: no ds\r\n");
                return XST_DMA_SG_NO_LIST;
        }
        /* Do nothing if already started */
        if (RingPtr->RunState == AXIDMA_CHANNEL_NOT_HALTED) {
                /* Need to update tail pointer if needed (Engine is not
                 * transferring)
                */
                return XST_SUCCESS;
        }
        if (!XAxiDma_BdRingHwIsStarted(RingPtr)) {
                /* If hardware is not running, then we need to put a valid
                current BD pointer to the current BD register before start the
                hardware
                */
                RegBase = RingPtr->ChanBase;
                /* Put a valid BD pointer in the current BD pointer register.
                So, the hardware is ready to go when tail BD pointer is
                updated */
                BdPtr = (UINTPTR)(void *)(RingPtr->BdaRestart);
                if (!XAxiDma_BdHwCompleted(BdPtr)) {
                        if (RingPtr->IsRxChannel) {
                                if (!RingIndex) {
                                        XAxiDma_WriteReg(RegBase,
                                        XAXIDMA_CDESC_OFFSET,
                                        (XAXIDMA_VIRT_TO_PHYS(BdPtr) &
                                        XAXIDMA_DESC_LSB_MASK));
                                        if (RingPtr->Addr_ext)
                                                XAxiDma_WriteReg(RegBase,
                                                XAXIDMA_CDESC_MSB_OFFSET,
                                               UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
                                                (BdPtr)));
                                }
                                else {
                                        XAxiDma_WriteReg(RegBase,
                                        (XAXIDMA_RX_CDESC0_OFFSET +
                                        (RingIndex - 1) *
                                        XAXIDMA_RX_NDESC_OFFSET),
                                        (XAXIDMA_VIRT_TO_PHYS(BdPtr) &
                                        XAXIDMA_DESC_LSB_MASK));
                                        if (RingPtr->Addr_ext)
                                                XAxiDma_WriteReg(RegBase,
                                                (XAXIDMA_RX_CDESC0_MSB_OFFSET+
                                               (RingIndex - 1)*
                                               XAXIDMA_RX_NDESC_OFFSET),
                                                UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
                                                (BdPtr)));
                                }
                        }
                        else {
```

```
                                                XAxiDma_WriteReg(RegBase,
                                                XAXIDMA_CDESC_OFFSET,
                                                (XAXIDMA_VIRT_TO_PHYS(BdPtr) &
                                                XAXIDMA_DESC_LSB_MASK));
                                                if (RingPtr->Addr_ext)
                                                        XAxiDma_WriteReg(RegBase,XAXIDMA_CDESC_MSB
                                                        _OFFSET,UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
                                                        (BdPtr)));
                                        }
                                }
                                else {
                                        /* Look for an uncompleted BD
                                        */
                                        while (XAxiDma_BdHwCompleted(BdPtr)) {
                                                BdPtr = XAxiDma_BdRingNext(RingPtr, BdPtr);

                                                if ((UINTPTR)BdPtr == (UINTPTR) RingPtr-
                                                >BdaRestart) {
                                                        xdbg_printf(XDBG_DEBUG_ERROR,
                                                        "StartBdRingHw: Cannot find valid
                                                        cdesc\r\n");
                                                        return XST_DMA_ERROR;
                                                }
                                                if (!XAxiDma_BdHwCompleted(BdPtr)) {
                                                        if (RingPtr->IsRxChannel) {
                                                                if (!RingIndex) {
                                                                        XAxiDma_WriteReg(RegBase,
                                                                        XAXIDMA_CDESC_OFFSET,
                                                                        (XAXIDMA_VIRT_TO_PHYS(BdPtr)&
                                                                        XAXIDMA_DESC_LSB_MASK));
                                                                        if (RingPtr->Addr_ext)
                                                                                XAxiDma_WriteReg(RegBase,
                                                                                XAXIDMA_CDESC_MSB_OFFSET,
                                                                                UPPER_32_BITS(
                                                                                XAXIDMA_VIRT_TO_PHYS
                                                                                (BdPtr)));
                                                                }
                                                                else {
                                                                        XAxiDma_WriteReg(RegBase,
                                                                        (XAXIDMA_RX_CDESC0_OFFSET +
                                                                        (RingIndex - 1) *
                                                                        XAXIDMA_RX_NDESC_OFFSET),
                                                                        (XAXIDMA_VIRT_TO_PHYS(BdPtr)
                                                                        & XAXIDMA_DESC_LSB_MASK));
                                                                        if (RingPtr->Addr_ext)
                                                                                XAxiDma_WriteReg(RegBase,
                                                                               (XAXIDMA_RX_CDESC0_MSB_OFFSET
                                                                                 +(RingIndex - 1) *
                                                                                XAXIDMA_RX_NDESC_OFFSE),
                                                                                UPPER_32_BITS(
                                                                                XAXIDMA_VIRT_TO_PHYS
                                                                                (BdPtr)));
                                                                }
                                                        }
                                                        else {
                                                                XAxiDma_WriteReg(RegBase,
                                                                        XAXIDMA_CDESC_OFFSET,
                                                                        (XAXIDMA_VIRT_TO_PHYS(BdPtr)&
                                                                        XAXIDMA_DESC_LSB_MASK));
                                                                        if (RingPtr->Addr_ext)
                                                                                XAxiDma_WriteReg
                                                                                (RegBase,
                                                                                 XAXIDMA_CDESC_MSB_OFFSET,
                                                                                UPPER_32_BITS(
                                                                                XAXIDMA_VIRT_TO_PHYS
                                                                                (BdPtr)));
                                                        }
                                        break;
                                }
                        }
                }
        }
        return XST_SUCCESS;
}
```

- Since this function can be used by RX as well as TX channel, first, we store the attribute RingIndex from the BD Ring structure passed into this function and is kept in variable RingIndex. The ringindex will tell us about which RX channel we are using.
- Check if the BD Ring provided is in running state by looking into RunState attribute of the BD Ring structure. If it is already started, no need to update the current descriptor register. Thus we leave the function with SUCCESS returned.
- UPDATION of Current Descriptor will happen only if channel is not running.
- We use a function to make sure that the BD Ring is not in RUNNING state. It is done by looking into the status register of DMA with respect to that channel. We check for Bit 0 of status register. If 0 its running, else halted.



- Once we make sure channel is not running, we store the channel base address (depends on whether its TX or RX) into RegBase register. Also we store the current descriptor pointer value taken from BdaRestart attribute. This attribute stores the pointer to the BD in the Ring that should be loaded when the channel starts.
- Check if this BD taken from BdaRestart is completed or not using a function. This function looks into the status register of that BD, especially bit 31 corresponding to complete bit. If 0 BD is not completed by HW.



- We have different situations from here onwards:
    1. Current BD is not completed yet
        a. It is RX channel
            i. RingIndex value is 0
               Update the CDESC of RX channel (BASE + 38h,3Ch) using physical address found from virtual address of that BD multiplied with DESC_OFFSET. It is obtained by adding BD address with (BdPhyAddr-BdAddr).
            ii. RingIndex is non-zero
               According to the RingIndex, corresponding channel current descriptor will be updated in similar fashion.

b. Is TX channel
   We directly update the CDESC of TX channel(BASE + 08h,0Ch) using the physical address found from virtual address of that BD.

2. Current BD is completed.
   We look for uncompleted BD. We update the BdPtr value with the address of either next BD or 1st BD if it was last BD. It is done by adding the address with separation between BDs. Then we make sure that new BD is not completed yet. If completed, we will exit this function with SUCCESS.
   a. It is RX channel
      i. RingIndex value is 0
         Update the CDESC of RX channel (BASE + 38h,3Ch) using physical address found from virtual address of that BD multiplied with DESC_OFFSET. It is obtained by adding BD address with (BdPhyAddr-BdAddr).
      ii. RingIndex is non-zero
         According to the RingIndex, corresponding channel current descriptor will be updated in similar fashion.
   b. Is TX channel
      We directly update the CDESC of TX channel(BASE + 08h,0Ch) using the physical address found from virtual address of that BD.

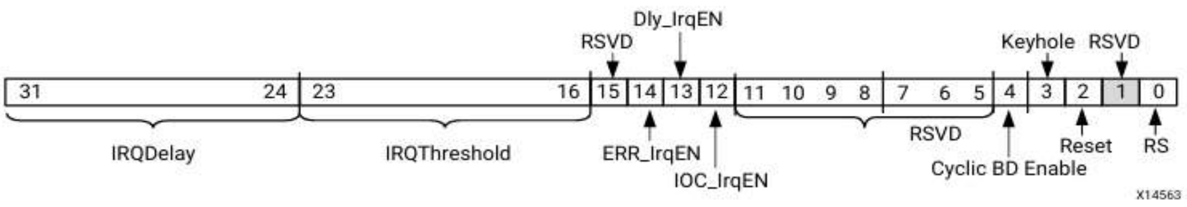
II. XAxiDma_StartBdRingHw() function takes in the (TX) BD Ring structure as parameter and is used to start the channel. Once a channel is started, the channel is not HALTED, means we will set the last bit of status register of DMA to 0 (RUNNING). The channel will be idle even though channel is started, means there will be no active transfers.

    General flow looks like, first we check if the channel is running or not. If it is not running, which is our case, we start the channel by setting the last bit, run bit, of control register of BD Ring for the TX channel to 1. Next, this function will check if there are any BDs in the Work group, under direct control of HW. Since all BDs are in Free group, it will exit the function with SUCCESS. If there were any unprocessed BDs as indicated by attribute HwCnt of that BD Ring Channel, we write to tail descriptor here itself.

    We will use this function in the later part of this example after allocating and preparing the BDs for transmission. In that case, we will be taken into the part where we will write the tail descriptor for the DMA Channel.

```
#define XAxiDma_BdRingHwIsStarted(RingPtr)                        \
	((XAxiDma_ReadReg((RingPtr)->ChanBase, XAXIDMA_SR_OFFSET) \
	& XAXIDMA_HALTED_MASK) ? FALSE : TRUE)

int XAxiDma_StartBdRingHw(XAxiDma_BdRing * RingPtr)
{
	UINTPTR RegBase;
	int RingIndex = RingPtr->RingIndex;
	if (!XAxiDma_BdRingHwIsStarted(RingPtr)) {
		/* Start the hardware
		*/
		RegBase = RingPtr->ChanBase;
		XAxiDma_WriteReg(RegBase, XAXIDMA_CR_OFFSET,
			XAxiDma_ReadReg(RegBase, XAXIDMA_CR_OFFSET)
			| XAXIDMA_CR_RUNSTOP_MASK);
	}

	if (XAxiDma_BdRingHwIsStarted(RingPtr)) {
		/* Note as started */
		RingPtr->RunState = AXIDMA_CHANNEL_NOT_HALTED;
		/* If there are unprocessed BDs then we want the channel to
		 begin processing right away */
		if (RingPtr->HwCnt > 0) {
			XAXIDMA_CACHE_INVALIDATE(RingPtr->HwTail);
			if (RingPtr->Cyclic) {
				XAxiDma_WriteReg(RingPtr->ChanBase,
					 XAXIDMA_TDESC_OFFSET,
					(u32)XAXIDMA_VIRT_TO_PHYS(RingPtr-
					>CyclicBd));
				if (RingPtr->Addr_ext)
					XAxiDma_WriteReg(RingPtr->ChanBase,
						XAXIDMA_TDESC_MSB_OFFSET,
						UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
						(RingPtr->CyclicBd)));
				return XST_SUCCESS;
			}
			if ((XAxiDma_BdRead(RingPtr->HwTail,
				XAXIDMA_BD_STS_OFFSET) &
				XAXIDMA_BD_STS_COMPLETE_MASK) == 0) {
				if (RingPtr->IsRxChannel) {
					if (!RingIndex) {
						XAxiDma_WriteReg(RingPtr->ChanBase,
							XAXIDMA_TDESC_OFFSET,
							(XAXIDMA_VIRT_TO_PHYS(RingPtr
							->HwTail) &
							XAXIDMA_DESC_LSB_MASK));
						if (RingPtr->Addr_ext)
							XAxiDma_WriteReg(RingPtr-
							>ChanBase,
							XAXIDMA_TDESC_MSB_OFFSET,
							UPPER_32_BITS
							(XAXIDMA_VIRT_TO_PHYS(RingPtr
							->HwTail)));
					}
					else {
						XAxiDma_WriteReg(RingPtr->ChanBase,
							(XAXIDMA_RX_TDESC0_OFFSET +
							(RingIndex - 1) *
							XAXIDMA_RX_NDESC_OFFSET),
							(XAXIDMA_VIRT_TO_PHYS(RingPtr-
							>HwTail) &
							XAXIDMA_DESC_LSB_MASK ));
						if (RingPtr->Addr_ext)
							XAxiDma_WriteReg(RingPtr-
							>ChanBase,
							(XAXIDMA_RX_TDESC0_MSB_OFFSET +
							(RingIndex - 1) *
							XAXIDMA_RX_NDESC_OFFSET),
							UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
							(RingPtr->HwTail)));
					}
				}
				else {
					XAxiDma_WriteReg(RingPtr->ChanBase,
                                        XAXIDMA_TDESC_OFFSET,
                                        (XAXIDMA_VIRT_TO_PHYS(RingPtr-
                                        >HwTail) & XAXIDMA_DESC_LSB_MASK));
                                if (RingPtr->Addr_ext)
                                        XAxiDma_WriteReg(RingPtr->ChanBase,
                                        XAXIDMA_TDESC_MSB_OFFSET,
                                        UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
                                        (RingPtr->HwTail)));
                        }
                }
        }
        return XST_SUCCESS;
    }
    return XST_DMA_ERROR;
}
```

- This function can be used by both RX and TX channels, so initially we store the RingIndex attribute used by RX channel into a register RingIndex.
- The first step is to check if the channel is started or not using the function. It is checked by reading from channel status register and check if last bit(halted) is set or not. If it is not started, we start the HW by writing 1 to last bit (bit 0) of control register of specified channel in DMA register address map.



- Again we confirm that the channel has been started by reading the status register of the channel. If it failed here, we return from this function with an ERROR.
- So now the channel or BD Ring is running, we update the BD Ring structure attribute Runstate as NOT_HALTED (1). On checking again, it should be working fine. If not it will return with error message.
- Check if there are any BDs in the Work group as indicated by attribute HwCnt of that DMA channel. If there are no BDs in the Work group, it will return from function with SUCCESS and not writing into tail descriptor.
- If HwCnt > 0, we have to start processing those unprocessed BDs by updating the tail descriptor.
- Check if the BD Ring for the given channel is cyclic. If so, then we update the tail descriptor of that channel with the physical address found from virtual address of the BD pointed by CyclicBD attribute. Additionally, if Addr_ext is enabled, then write to MSB tail descriptor register for that channel. It returns from the function after this with SUCCESS.
- If noncyclic, we check if the last DMA in Work group is completed or not. It is done by checking the complete bit of status register of this last BD in work group. If it is not completed, based on whether it is TX or RX channel, we update the tail descriptor.
- If it is RX channel, based on RingIndex value, we will update the corresponding Tail descriptor with physical address of HwTail BD.

- Similarly for TX channel, we update tail descriptor with physical address of HwTail BD.

Basic things understood from this function are, it will start the channel by writing into control register of that channel in DMA. Since the channel is already started, Tail descriptors will be updated only if there are any BDs in the Work group **(that is, if HwCnt > 0)**. If so, first priority is given for attribute cyclic, then update the tail descriptor with physical address of cyclic BD. If non cyclic BD Ring is used, then we check if the last BD in the work group is finished or not. If finished, no need to write to tail descriptor. Else, based on TX or RX channel, we will update the tail descriptor with the physical address of last BD in work group (HwTail).
In case of TX Setup, since no BDs are in Work group, it will not update the tail descriptor here, only starting the channel will happen here.

This ends the TX Setup part.

6. **RX SETUP**

With the TX channel setup complete, the next step is to configure the RX channel to make it ready to receive incoming data. Unlike the TX setup, where we prepare the descriptors for transmission, the RX setup involves allocating RX BDs (Buffer Descriptors) and configuring the hardware such that it is ready to receive data samples via the RX path immediately after the function completes.
The parameter passed is the pointer to AxiDma engine, which has attributes like TXnumchannels, RXnumchannels, RegBase, MicroDMA etc.

- First step in RX setup is to take store the pointer to RX BD Ring using function XAxiDma_GetRxRing() and passing the pointer to DMA instance. This function will return pointer to 0th RX BD Ring. We already know that, based on the number of RX channels, we can have multiple RX BD Rings, each indicated by value RingIndex attribute.

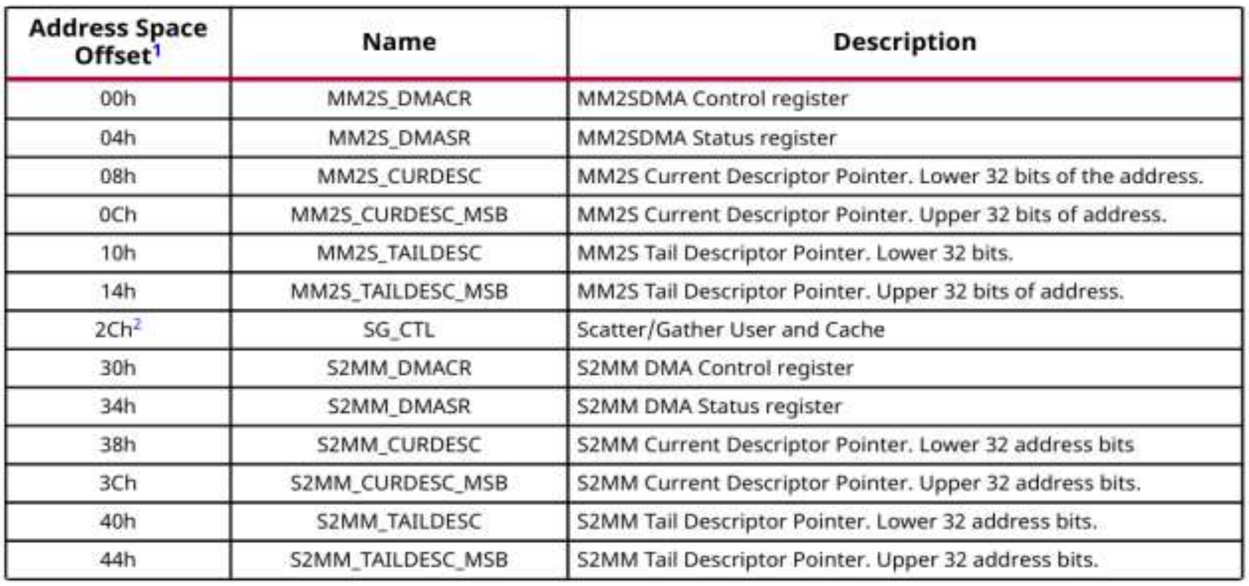
```
➔ RxRingPtr = XAxiDma_GetRxRing(&AxiDma);

#define XAxiDma_GetRxRing(InstancePtr) \
            (&((InstancePtr)->RxBdRing[0]))
```

- Before setting up the RX BD Ring, we need to disable all interrupts for the channel. It is done by setting bits 12, 13, 14 of control register in DMA Register Address Map for this channel (S2MM_DMACR – 30h). this function reads the control register of the channel, from DMA channel Base address and modify these 3 bits using a MASK and written back to same register.

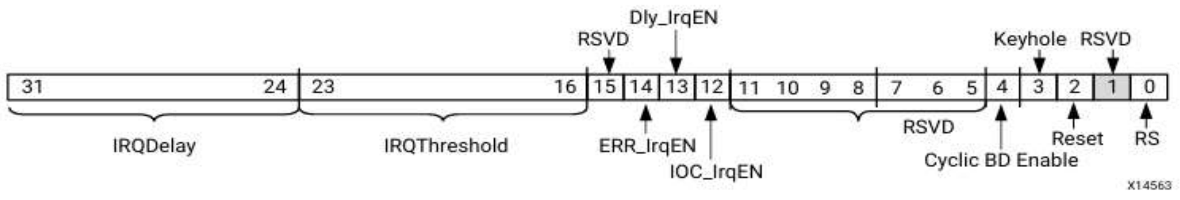
```
#define XAXIDMA_IRQ_ALL_MASK     0x00007000 /**< All interrupts */

➔ XAxiDma_BdRingIntDisable(RxRingPtr, XAXIDMA_IRQ_ALL_MASK);

#define XAxiDma_BdRingIntDisable(RingPtr, Mask)                 \
        (XAxiDma_WriteReg((RingPtr)->ChanBase, XAXIDMA_CR_OFFSET, \
        XAxiDma_ReadReg((RingPtr)->ChanBase, XAXIDMA_CR_OFFSET) & \
        ~((Mask) & XAXIDMA_IRQ_ALL_MASK)))
```

- We need to setup the IRQ threshold and Delay in the control register of this RX BD Ring. The procedure is same as TX channel. What we do is read from the control register of parameter channel (RX – 30h), update the Threshold and Delay field (bits

16 to 23 for Threshold and bits 24 to 31 for Delay) with the user provided values respectively. Finally we write back this value into the control register of the channel.



Address Space Offset[1]	Name	Description
00h	MM2S_DMACR	MM2SDMA Control register
04h	MM2S_DMASR	MM2SDMA Status register
08h	MM2S_CURDESC	MM2S Current Descriptor Pointer. Lower 32 bits of the address.
0Ch	MM2S_CURDESC_MSB	MM2S Current Descriptor Pointer. Upper 32 bits of address.
10h	MM2S_TAILDESC	MM2S Tail Descriptor Pointer. Lower 32 bits.
14h	MM2S_TAILDESC_MSB	MM2S Tail Descriptor Pointer. Upper 32 bits of address.
2Ch[2]	SG_CTL	Scatter/Gather User and Cache
30h	S2MM_DMACR	S2MM DMA Control register
34h	S2MM_DMASR	S2MM DMA Status register
38h	S2MM_CURDESC	S2MM Current Descriptor Pointer. Lower 32 address bits
3Ch	S2MM_CURDESC_MSB	S2MM Current Descriptor Pointer. Upper 32 address bits.
40h	S2MM_TAILDESC	S2MM Tail Descriptor Pointer. Lower 32 address bits.
44h	S2MM_TAILDESC_MSB	S2MM Tail Descriptor Pointer. Upper 32 address bits.



```
int Delay = 0;
int Coalesce = 1;
➔  XAxiDma_BdRingSetCoalesce(RxRingPtr, Coalesce, Delay);

int XAxiDma_BdRingSetCoalesce(XAxiDma_BdRing *RingPtr, u32 Counter,
u32 Timer)
{
        u32 Cr;
        Cr = XAxiDma_ReadReg(RingPtr->ChanBase, XAXIDMA_CR_OFFSET);
        if (Counter != XAXIDMA_NO_CHANGE) {
                if ((Counter == 0) || (Counter > 0xFF)) {
                        xdbg_printf(XDBG_DEBUG_ERROR,"BdRingSetCoalesce:”
                        "invalid  coalescing threshold %d",
                        (int)Counter);
                        return XST_FAILURE;
                }
                Cr = (Cr & ~XAXIDMA_COALESCE_MASK) |
                        (Counter << XAXIDMA_COALESCE_SHIFT);
        }

        if (Timer != XAXIDMA_NO_CHANGE) {
                if (Timer > 0xFF) {
                        xdbg_printf(XDBG_DEBUG_ERROR,"BdRingSetCoalesce:"
                        "invalid  delay counter %d", (int)Timer);
                        return XST_FAILURE;
                }

                Cr = (Cr & ~XAXIDMA_DELAY_MASK) |
                (Timer << XAXIDMA_DELAY_SHIFT);
        }
        XAxiDma_WriteReg(RingPtr->ChanBase, XAXIDMA_CR_OFFSET, Cr);
        return XST_SUCCESS;
}
```

- Before creating the BD Ring for the RX channel, it is essential to determine how many BDs (Buffer Descriptors) can be created from the memory allocated for RX BDs. This involves calculating the total number of BDs based on the available memory and the size of each BD.
  We use the function XAxiDma_BdRingCntCalc() which takes in parameter the BD alignment and Total bytes for storing RX BD words. Divide the Bytes with alignment of each BD and return the count.

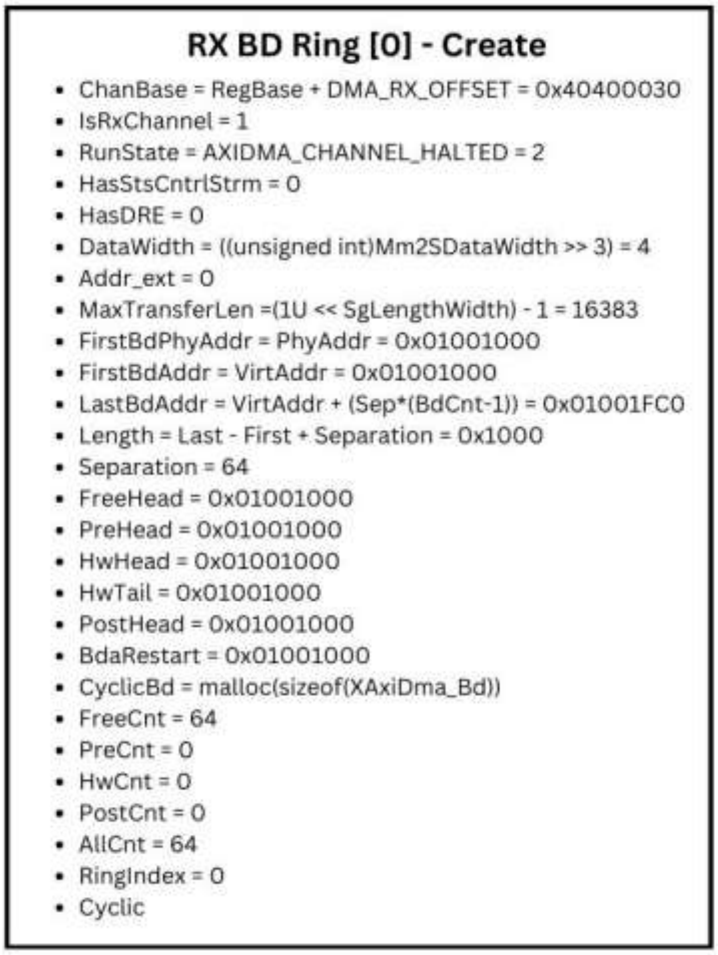
```
#define RX_BD_SPACE_BASE   (MEM_BASE_ADDR + 0x00001000)
#define RX_BD_SPACE_HIGH   (MEM_BASE_ADDR + 0x00001FFF)

➔ BdCount = XAxiDma_BdRingCntCalc(XAXIDMA_BD_MINIMUM_ALIGNMENT,
                          RX_BD_SPACE_HIGH - RX_BD_SPACE_BASE + 1);
#define XAxiDma_BdRingCntCalc(Alignment, Bytes)             \
      (uint32_t)((Bytes)/((sizeof(XAxiDma_Bd)+((Alignment)-
      1))&~((Alignment)-1)))
```

- Create the RX BD Ring. We set the pointer to RX BD ring, keep the physical and virtual address to RX_BD_SPACE_BASE, set the alignment to minimum possible value, which is 64 bytes, and pass the total BD count to the function.
  Procedure is same as in the TX BD Ring. The only difference is the function is now pointing to RX BD ring instead of TX and the physical address will be now updated with respect to RX space as specified by user. The flow of the code is:
  - Check if the Bdcount is a valid number and alignment is meeting certain conditions.
  - Reset all counters used in RX BD Ring instance.
  - Update the separation field of the Ring instance using the size of BD instance and alignment chosen.
  - Clear the entire Ring space using function memset.
  - Update the Next Descriptor register, Has DRE register and Has status control stream register of each of the BDs in the BD ring. It is to be noted that only the next descriptor field will change between each BDs as it points to adjacent BD address. Rest all registers will have same value as is specified from RX BD Ring instance.
  - Last BD will point to first BD.
  - Finally, update all attributes of the RX BD Ring Instance, including counters and pointers.

```
➔ Status = XAxiDma_BdRingCreate(RxRingPtr, RX_BD_SPACE_BASE,
       RX_BD_SPACE_BASE,XAXIDMA_BD_MINIMUM_ALIGNMENT, BdCount);

u32 XAxiDma_BdRingCreate(XAxiDma_BdRing *RingPtr, UINTPTR PhysAddr,
      UINTPTR VirtAddr, u32 Alignment, int BdCount)
{
      int i;
      UINTPTR BdVirtAddr;
      UINTPTR BdPhysAddr;
      if (BdCount <= 0) {
            xdbg_printf(XDBG_DEBUG_ERROR, "BdRingCreate: non-
            positive BD number %d\r\n", BdCount);
            return XST_INVALID_PARAM;
      }
      RingPtr->AllCnt = 0;
      RingPtr->FreeCnt = 0;
      RingPtr->HwCnt = 0;
      RingPtr->PreCnt = 0;
      RingPtr->PostCnt = 0;
      RingPtr->Cyclic = 0;
      if (Alignment < XAXIDMA_BD_MINIMUM_ALIGNMENT) {
            xdbg_printf(XDBG_DEBUG_ERROR, "BdRingCreate: alignment
            too small %d, need to be at least %d\r\n",
            (int)Alignment,XAXIDMA_BD_MINIMUM_ALIGNMENT);
```

```
        return XST_INVALID_PARAM;
}
if ((Alignment - 1) & Alignment) {
        xdbg_printf(XDBG_DEBUG_ERROR, "BdRingCreate: alignment
        not valid %d\r\n", (int)Alignment);
        return XST_INVALID_PARAM;
}
if ((PhysAddr % Alignment) || (VirtAddr % Alignment)) {
        xdbg_printf(XDBG_DEBUG_ERROR, "BdRingCreate: Physical
        address %x and virtual address %x have different
        alignment\r\n",(unsigned int)PhysAddr, (unsigned
        int)VirtAddr);
        return XST_INVALID_PARAM;
}
RingPtr->Separation =
        (sizeof(XAxiDma_Bd) +(Alignment - 1))& ~(Alignment -1);
if (VirtAddr >(VirtAddr + (RingPtr->Separation * BdCount)-1)){
        xdbg_printf(XDBG_DEBUG_ERROR, "BdRingCreate: BD space
        cross 0x0\r\n");
        return XST_DMA_SG_LIST_ERROR;
}
memset((void *) VirtAddr, 0, (RingPtr->Separation * BdCount));
BdVirtAddr = VirtAddr;
BdPhysAddr = PhysAddr + RingPtr->Separation;
for (i = 1; i < BdCount; i++) {
        XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_NDESC_OFFSET,
                     (BdPhysAddr & XAXIDMA_DESC_LSB_MASK));
        XAxiDma_BdWrite(BdVirtAddr,
        XAXIDMA_BD_NDESC_MSB_OFFSET,UPPER_32_BITS(BdPhysAddr));
        /* Put hardware information in the BDs
        */
        XAxiDma_BdWrite(BdVirtAddr,
        XAXIDMA_BD_HAS_STSCNTRL_OFFSET,(u32)RingPtr>HasStsCntrl
        Strm);

        XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_HAS_DRE_OFFSET,
        (((u32)(RingPtr->HasDRE)) << XAXIDMA_BD_HAS_DRE_SHIFT)|
        RingPtr->DataWidth);

        XAXIDMA_CACHE_FLUSH(BdVirtAddr);
        BdVirtAddr += RingPtr->Separation;
        BdPhysAddr += RingPtr->Separation;
}
XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_NDESC_OFFSET,
             (PhysAddr & XAXIDMA_DESC_LSB_MASK));
XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_NDESC_MSB_OFFSET,
             UPPER_32_BITS(PhysAddr));
XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_HAS_STSCNTRL_OFFSET,
        (u32)RingPtr->HasStsCntrlStrm);

XAxiDma_BdWrite(BdVirtAddr, XAXIDMA_BD_HAS_DRE_OFFSET,
        (((u32)(RingPtr->HasDRE)) << XAXIDMA_BD_HAS_DRE_SHIFT)|
        RingPtr->DataWidth);
RingPtr->RunState = AXIDMA_CHANNEL_HALTED;
RingPtr->FirstBdAddr = VirtAddr;
RingPtr->FirstBdPhysAddr = PhysAddr;
RingPtr->LastBdAddr = BdVirtAddr;
RingPtr->Length = RingPtr->LastBdAddr - RingPtr->FirstBdAddr +
        RingPtr->Separation;
RingPtr->AllCnt = BdCount;
RingPtr->FreeCnt = BdCount;
RingPtr->FreeHead = (XAxiDma_Bd *) VirtAddr;
RingPtr->PreHead = (XAxiDma_Bd *) VirtAddr;
RingPtr->HwHead = (XAxiDma_Bd *) VirtAddr;
RingPtr->HwTail = (XAxiDma_Bd *) VirtAddr;
RingPtr->PostHead = (XAxiDma_Bd *) VirtAddr;
RingPtr->BdaRestart = (XAxiDma_Bd *) VirtAddr;
```

```
        RingPtr->CyclicBd = (XAxiDma_Bd *) malloc(sizeof(XAxiDma_Bd));
        return XST_SUCCESS;
}
---------------------------------------------------------------------
```



RX BD Ring [0] - Create

- ChanBase = RegBase + DMA_RX_OFFSET = 0x40400030
- IsRxChannel = 1
- RunState = AXIDMA_CHANNEL_HALTED = 2
- HasStsCntrlStrm = 0
- HasDRE = 0
- DataWidth = ((unsigned int)Mm2SDataWidth >> 3) = 4
- Addr_ext = 0
- MaxTransferLen =(1U << SgLengthWidth) - 1 = 16383
- FirstBdPhyAddr = PhyAddr = 0x01001000
- FirstBdAddr = VirtAddr = 0x01001000
- LastBdAddr = VirtAddr + (Sep*(BdCnt-1)) = 0x01001FC0
- Length = Last - First + Separation = 0x1000
- Separation = 64
- FreeHead = 0x01001000
- PreHead = 0x01001000
- HwHead = 0x01001000
- HwTail = 0x01001000
- PostHead = 0x01001000
- BdaRestart = 0x01001000
- CyclicBd = malloc(sizeof(XAxiDma_Bd))
- FreeCnt = 64
- PreCnt = 0
- HwCnt = 0
- PostCnt = 0
- AllCnt = 64
- RingIndex = 0
- Cyclic

RX BD Ring 0 after being created will be updated with these values. We can see that the Base address for RX BD Ring is 0x1000 away from that of TX BD Ring. Rest all parameters are ideal.

- For resetting the BDs in the BD Ring, we need to first create an all-zero BD except the ones which we have already written at time of creating BD Ring. So we create a template BD which is later passed onto all the BDs in the Ring. Template BD is similar to individual BDs in the Ring, that is, it form an array of 16 32-bit words and set all words from 3 to 14 as 0.

```
---------------------------------------------------------------------
XAxiDma_Bd BdTemplate;

➔ XAxiDma_BdClear(&BdTemplate);

#define XAxiDma_BdClear(BdPtr)                              \
        memset((void *)(((UINTPTR)(BdPtr))+XAXIDMA_BD_START_CLEAR),0,\
        XAXIDMA_BD_BYTES_TO_CLEAR)

---------------------------------------------------------------------
```

- Clone this template BD into all BDs in the BD Ring. Ensure that BD Ring is not running and all BDs are in the Free group. Otherwise, no BD should be modified. While updating BDs, ensure that all BDs' status register complete bit should be set a 0.

```
---------------------------------------------------------------------
➔ Status = XAxiDma_BdRingClone(TxRingPtr, &BdTemplate);

int XAxiDma_BdRingClone(XAxiDma_BdRing * RingPtr, XAxiDma_Bd *
        SrcBdPtr)
{
        int i;
        UINTPTR CurBd;
        u32 Save;
        XAxiDma_Bd TmpBd;
        /* Can't do this function if there isn't a ring */
        if (RingPtr->AllCnt == 0) {
              xdbg_printf(XDBG_DEBUG_ERROR, "BdRingClone: no
              bds\r\n");
              return XST_DMA_SG_NO_LIST;
        }
        /* Can't do this function with the channel running */
        if (RingPtr->RunState == AXIDMA_CHANNEL_NOT_HALTED) {
              xdbg_printf(XDBG_DEBUG_ERROR, "BdRingClone: bd ring
              started already, cannot do\r\n");
              return XST_DEVICE_IS_STARTED;
        }
        /* Can't do this function with some of the BDs in use */
        if (RingPtr->FreeCnt != RingPtr->AllCnt) {
              xdbg_printf(XDBG_DEBUG_ERROR, "BdRingClone: some bds
              already in use %d/%d\r\n",RingPtr->FreeCnt, RingPtr-
              >AllCnt);
              return XST_DMA_SG_LIST_ERROR;
        }
        /* Make a copy of the template then modify it by clearing
         * the complete bit in status/control field
         */
        memcpy(&TmpBd, SrcBdPtr, sizeof(XAxiDma_Bd));

        Save = XAxiDma_BdRead(&TmpBd, XAXIDMA_BD_STS_OFFSET);
        Save &= ~XAXIDMA_BD_STS_COMPLETE_MASK;
        XAxiDma_BdWrite(&TmpBd, XAXIDMA_BD_STS_OFFSET, Save);

        for (i = 0, CurBd = RingPtr->FirstBdAddr;
             i < RingPtr->AllCnt; i++, CurBd += RingPtr->Separation) {
              memcpy((void *)((UINTPTR)CurBd+XAXIDMA_BD_START_CLEAR),
                     (void *)((UINTPTR)(&TmpBd) +
                     XAXIDMA_BD_START_CLEAR),XAXIDMA_BD_BYTES_TO_CLEAR
                     );
              XAXIDMA_CACHE_FLUSH(CurBd);
        }
        return XST_SUCCESS;
}
```

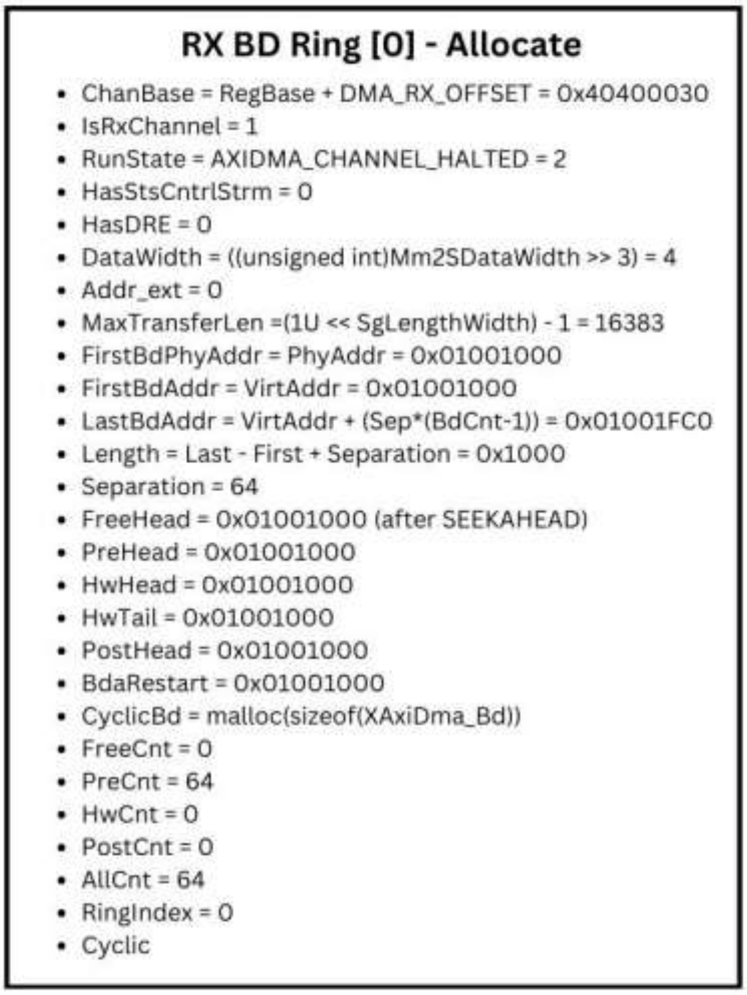
From this point onwards, it is different from that of TX Setup.

- We calculate the total number of free BDs in the RX BD Ring and store them into variable FreeBdCount. Initially, all the BDs are in the Free group. So, we get the BdCount value into FreeBdCount.
  The use of this function is to allocate all the available BDs in the Ring for reception.

  ```
  FreeBdCount = XAxiDma_BdRingGetFreeCnt(RxRingPtr);

  #define XAxiDma_BdRingGetFreeCnt(RingPtr)  ((RingPtr)->FreeCnt)
  ```

- This step involves allocating the BDs (Buffer Descriptors) that will be sent to the hardware. The function primarily ensures that the requested number of BDs for assignment to the channel is within the range of available BDs in the ring. This is important to prevent exceeding the number of BDs that can be allocated based on the current free BDs in the RX BD Ring.
  This function plays a crucial role because, before these BDs can be sent to the hardware, they need to be prepared. First, it checks the availability of the requested BDs and updates the attributes of the RX BD Ring accordingly. It reduces the

FreeCount by the number of requested BDs and increases the PreCount, as these BDs are now moved from the Free group to the Pre-Work group. Additionally, the function updates the FreeHead attribute, which is a pointer to the next available BD in the Free group after the current allocation, ensuring that the RX BD Ring is accurately updated. This is critical for keeping track of which BDs remain free and which have been assigned. Once the BDs are allocated, they can be prepared for hardware use. The function also returns a pointer to the first allocated BD in the Pre-Work group, allowing further preparation before these BDs are sent to the hardware for data reception.



RX BD Ring [0] - Allocate

- ChanBase = RegBase + DMA_RX_OFFSET = 0x40400030
- IsRxChannel = 1
- RunState = AXIDMA_CHANNEL_HALTED = 2
- HasStsCntrlStrm = 0
- HasDRE = 0
- DataWidth = ((unsigned int)Mm2SDataWidth >> 3) = 4
- Addr_ext = 0
- MaxTransferLen =(1U << SgLengthWidth) - 1 = 16383
- FirstBdPhyAddr = PhyAddr = 0x01001000
- FirstBdAddr = VirtAddr = 0x01001000
- LastBdAddr = VirtAddr + (Sep*(BdCnt-1)) = 0x01001FC0
- Length = Last - First + Separation = 0x1000
- Separation = 64
- FreeHead = 0x01001000 (after SEEKAHEAD)
- PreHead = 0x01001000
- HwHead = 0x01001000
- HwTail = 0x01001000
- PostHead = 0x01001000
- BdaRestart = 0x01001000
- CyclicBd = malloc(sizeof(XAxiDma_Bd))
- FreeCnt = 0
- PreCnt = 64
- HwCnt = 0
- PostCnt = 0
- AllCnt = 64
- RingIndex = 0
- Cyclic

After creating BD Ring, after allocation, we can see that all BDs from free group is now in pre-work group. Also, after modifying the FreeHead attribute, we still get pointer to first BD, because on looking into function XAXIDMA_RING_SEEKAHEAD(), Addr after updation will exceed the LastAddr attribute (0x01002000 > 0x01001FC0), thus we will subtract with 0x1000 and we get 0x01001000.

```
------------------------------------------------------------------------------------------------------------
#define XAXIDMA_RING_SEEKAHEAD(RingPtr, BdPtr, NumBd)          \
{                                                               \
        UINTPTR Addr = (UINTPTR)(void *)(BdPtr);               \
        Addr += ((RingPtr)->Separation * (NumBd));                \
        if ((Addr > (RingPtr)->LastBdAddr)||((UINTPTR)(BdPtr)>Addr)) \
        {                                                          \
                Addr -= (RingPtr)->Length;                        \
        }                                                          \
        (BdPtr) = (XAxiDma_Bd*)(void *)Addr;                      \
}

XAxiDma_Bd *BdPtr;

➔  Status = XAxiDma_BdRingAlloc(RxRingPtr, FreeBdCount, &BdPtr);

int XAxiDma_BdRingAlloc(XAxiDma_BdRing * RingPtr, int NumBd,
        XAxiDma_Bd ** BdSetPtr)
{
        if (NumBd <= 0) {
              xdbg_printf(XDBG_DEBUG_ERROR, "BdRingAlloc: negative BD
                    number %d\r\n", NumBd);
              return XST_INVALID_PARAM;
        }
        /* Enough free BDs available for the request? */
        if (RingPtr->FreeCnt < NumBd) {
              xdbg_printf(XDBG_DEBUG_ERROR,"Not enough BDs
                    to alloc %d/%d\r\n", NumBd, RingPtr->FreeCnt);
              return XST_FAILURE;
        }
        /* Set the return argument and move FreeHead forward */
        *BdSetPtr = RingPtr->FreeHead;

        XAXIDMA_RING_SEEKAHEAD(RingPtr, RingPtr->FreeHead, NumBd);
        RingPtr->FreeCnt -= NumBd;
        RingPtr->PreCnt += NumBd;
        return XST_SUCCESS;
}

--------------------------------------------------------------------
```

Once this function is run, the BD pointer BdPtr will have the pointer to first BD in pre-work group. This pointer is returned by this function so that they can be used for further actions.

- Store the returned BD pointer, BdPtr into BdCurPtr variable and store the RX_BUFFER_BASE value into RxBufferPtr. It is from this address, RX_BUFFER_BASE, onwards, we will store the received data. This address starts from 0x01300000.

- After allocating some or all of the BDs (in our case, all) into the Pre-Work group, the next step is to prepare them before they are handed over to the hardware. Preparing the BDs involves several key updates: setting the buffer address, which specifies where the data should be stored or retrieved from, depending on whether it’s a TX or RX operation; updating the length of the data to be transferred; updating the control register and assigning an ID to each BD. These updates ensure that the BDs are fully configured and ready for the DMA engine to perform data transfers, whether sending (TX) or receiving (RX) data.

```
---------------------------------------------------------------------------------------------------------------
for (Index = 0; Index < FreeBdCount; Index++)
{
        Status = XAxiDma_BdSetBufAddr(BdCurPtr, RxBufferPtr);
        if (Status != XST_SUCCESS) {
              xil_printf("Set buffer addr %x on BD %x failed %d\r\n",
             (unsigned int)RxBufferPtr,(UINTPTR)BdCurPtr, Status);
              return XST_FAILURE;
        }

        Status = XAxiDma_BdSetLength(BdCurPtr, MAX_PKT_LEN,
              RxRingPtr->MaxTransferLen);
        if (Status != XST_SUCCESS) {
              xil_printf("Rx set length %d on BD %x failed %d\r\n",
              MAX_PKT_LEN, (UINTPTR)BdCurPtr, Status);
              return XST_FAILURE;
        }
        /* Receive BDs do not need to set anything for the control
         * The hardware will set the SOF/EOF bits per stream status
         */
        XAxiDma_BdSetCtrl(BdCurPtr, 0);
        XAxiDma_BdSetId(BdCurPtr, RxBufferPtr);
        RxBufferPtr += MAX_PKT_LEN;
        BdCurPtr = (XAxiDma_Bd *)XAxiDma_BdRingNext(RxRingPtr,
              BdCurPtr);
```

}

Let’s look more detail into each of these functions:

- XAxiDma_BdSetBufAddr() function is used to set the buffer address for a specific BD (Buffer Descriptor). It takes two parameters: a pointer to the current BD that needs to be updated and the address value that should be written into the BD. In the case of RX operations, this address is typically the RxBufferPtr, which is updated during each iteration to point to the appropriate buffer location. The function will return SUCCESS if the BD supports DRE (Data Realignment Engine) and the address is properly aligned. If these conditions are not met, the function will return an INVALID_PARAM error, indicating that the buffer address is invalid or improperly aligned.

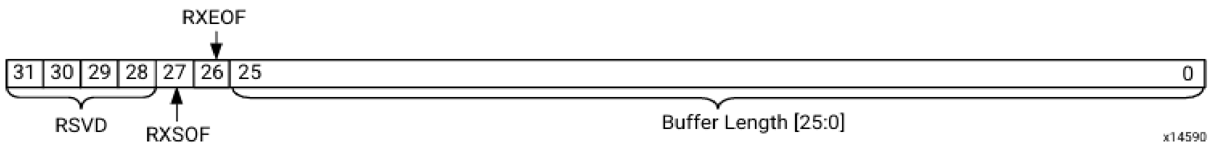
  This address provides location of the data to be transferred between stream and memory map.

31
0
Buffer Address [31:0]
X14570

```
-----------------------------------------------------------------------------------------------
u32 XAxiDma_BdSetBufAddr(XAxiDma_Bd* BdPtr, UINTPTR Addr)
{
	u32 HasDRE;
	u8 WordLen;
	HasDRE = XAxiDma_BdRead(BdPtr,
		XAXIDMA_BD_HAS_DRE_OFFSET);
	WordLen = HasDRE & XAXIDMA_BD_WORDLEN_MASK;

	if (Addr & (WordLen - 1)) {
		if ((HasDRE & XAXIDMA_BD_HAS_DRE_MASK) == 0) {
			xil_printf("Error set buf addr %x with %x
			      and %x, %x\r\n",Addr, HasDRE,
			      (WordLen - 1),Addr & (WordLen - 1));

			return XST_INVALID_PARAM;
		}
	}

#if defined(__aarch64__) || defined(__arch64__)
	XAxiDma_BdWrite64(BdPtr, XAXIDMA_BD_BUFA_OFFSET, Addr);
#else
	XAxiDma_BdWrite(BdPtr, XAXIDMA_BD_BUFA_OFFSET, Addr);
#endif

	return XST_SUCCESS;
}
--------------------------------------------------------------
```

- XAxiDma_BdSetLength() function is used to set the length field in the control register of each BD (Buffer Descriptor), which is 26 bits long. This limitation causes the DMA configuration in the HDL (as set through GUI parameters), where the maximum width of the buffer length register is restricted to 26 bits. This means that for each BD, only these 26 bits can be assigned to specify the data transfer length.
  The length specified must be non-zero and should not exceed the maximum transfer length allowed by the buffer length register. For the TX channel, the value passed corresponds to the number of bytes to transmit from the TX buffer associated with that BD. In the case of the RX channel, the length should be the size of the RX buffer associated with the BD in bytes.
  The function takes three parameters: a pointer to the BD that needs to be updated, the number of bytes to be transferred, and the maximum

allowable range for the transfer. The function checks if the length is within the valid range before updating the control register. If the length is valid, it is written into the BD, ensuring correct data transfer operations.



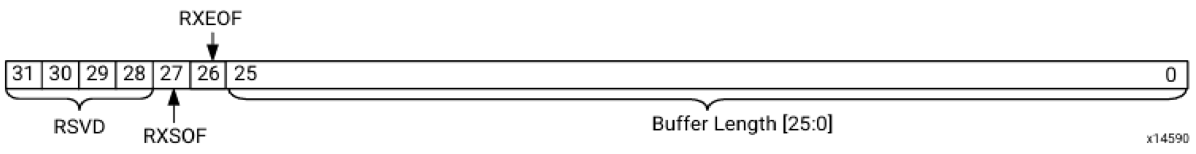
```
int XAxiDma_BdSetLength(XAxiDma_Bd *BdPtr, u32 LenBytes, u32
        LengthMask)
{
        if (LenBytes <= 0 || (LenBytes > LengthMask)) {
                xdbg_printf(XDBG_DEBUG_ERROR, "invalid length
                        %d\n",(int)LenBytes);
                return XST_INVALID_PARAM;
        }
        XAxiDma_BdWrite((BdPtr), XAXIDMA_BD_CTRL_LEN_OFFSET,
        ((XAxiDma_BdRead((BdPtr), XAXIDMA_BD_CTRL_LEN_OFFSET)&\
        ~LengthMask)) | LenBytes);

        return XST_SUCCESS;
}
```

Once the byte length is validated, it reads from the control register of the provided BD, clear the length part and add this data with user provided length value. Later written back to the control register.

- XAxiDma_BdSetCtrl() function is responsible for updating the Start of Frame (SOF) and End of Frame (EOF) bits within the control register of a specific BD (Buffer Descriptor). These bits are critical for marking the beginning and end of a frame, especially in the context of data transmission.
  In the case of a TX (transmit) channel, it is essential to explicitly set the SOF and EOF control flag bits in the control register. This ensures that the hardware correctly interprets where a frame starts and ends during the transmission of data. For example, if multiple BDs are being used to transmit a larger data buffer, the first BD would have the SOF bit set, while the last BD would have the EOF bit set. These flags help coordinate proper framing of the data during transmission.
  For an RX (receive) channel, however, these SOF and EOF bits are automatically set by the hardware when receiving data. Therefore, in RX operations, there is no need for the software to manually update these flags; the hardware handles it based on the incoming data.

```
void XAxiDma_BdSetCtrl(XAxiDma_Bd* BdPtr, u32 Data)
{
        u32 RegValue = XAxiDma_BdRead(BdPtr,
                XAXIDMA_BD_CTRL_LEN_OFFSET);
        RegValue &= ~XAXIDMA_BD_CTRL_ALL_MASK;
        RegValue |= (Data & XAXIDMA_BD_CTRL_ALL_MASK);
        XAxiDma_BdWrite((BdPtr), XAXIDMA_BD_CTRL_LEN_OFFSET,
                RegValue);

        return;
}
```

Using the parameter Data, we can set these bits as 1 or 0. First we read the control register of that BD, remove those bit values for EOF and SOF, add this data with that provided by user, write it back to the control register. Bits 26 and 27 correspond to EOF and SOF respectively.



- XAxiDma_BdSetId() function is used to assign an arbitrary value to each BD (Buffer Descriptor), allowing for a way to uniquely identify and associate each BD with the specific data it handles. This ID serves as a reference, making it easier to track which BD corresponds to which part of the data being transmitted or received.
  In our case, the ID value being set for each BD is the RxBufferPtr itself. This means that the pointer to the RX buffer is being used as the unique identifier, allowing us to associate each BD with the specific RX buffer it is managing.
  It is to be noted that this ID register for BD is not in documentation, but is in word address offset 34h.

  ```
  #define XAxiDma_BdSetId(BdPtr, Id)                    \
         (XAxiDma_BdWrite((BdPtr), XAXIDMA_BD_ID_OFFSET,
         (UINTPTR)(Id)))
  ```

- Preparation of BDs in the Pre-work group is done. Now clear the RX buffer space to prevent any junk data being present in this area. This is done using memset function.

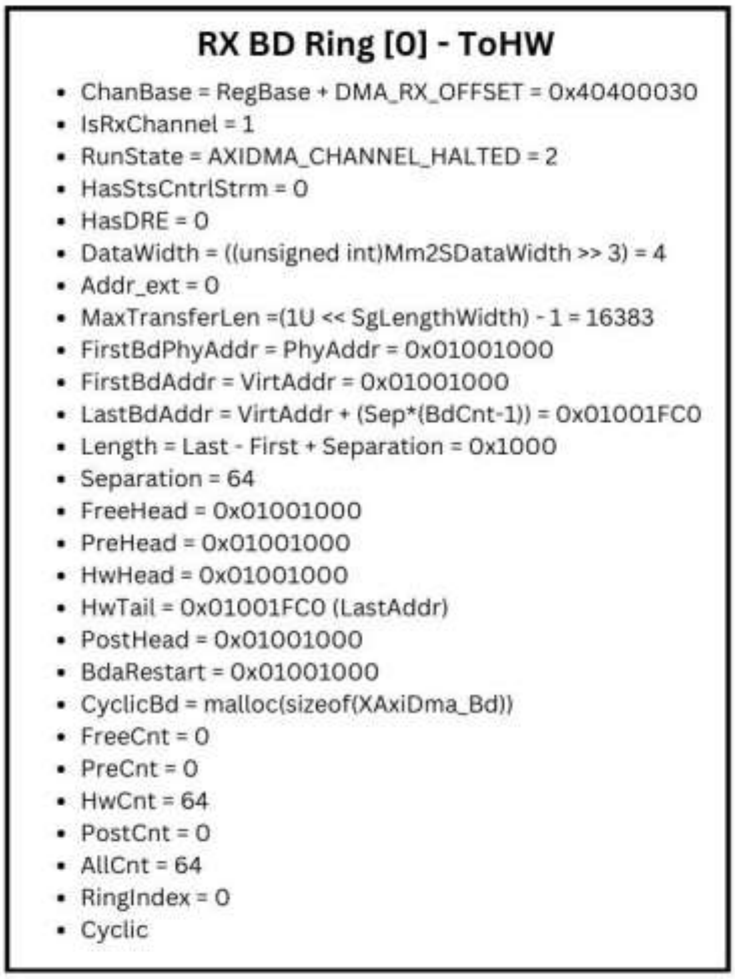
- After preparing the BDs, the next step is to pass them to the hardware for processing, which is done using the XAxiDma_BdRingToHw() function. This function is responsible for transferring control of a set of BDs from the software to the hardware, enabling the hardware to handle data transmission or reception.
  The function requires three parameters. First, it takes a pointer to the RX BD Ring, which represents the structure managing the BDs for the RX channel. Second, it requires the number of BDs in the set, specifying how many BDs are being committed to the hardware for processing. Finally, it takes a pointer to the first BD in the set, indicating the initial BD that will be handed off to the hardware.
  The hardware will now start processing the data associated with these BDs based on the DMA transfer settings.

  ```
  ➔ Status = XAxiDma_BdRingToHw(RxRingPtr, FreeBdCount, BdPtr);

  int XAxiDma_BdRingToHw(XAxiDma_BdRing * RingPtr, int NumBd,
         XAxiDma_Bd * BdSetPtr)
  {
         XAxiDma_Bd *CurBdPtr;
         int i;
         u32 BdCr;
         u32 BdSts;
         int RingIndex = RingPtr->RingIndex;
         if (NumBd < 0) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingToHw: negative BD
                       number %d\r\n", NumBd);
                return XST_INVALID_PARAM;
         }
         /* If the commit set is empty, do nothing */
         if (NumBd == 0) {
                return XST_SUCCESS;
         }
         /* Make sure we are in sync with XAxiDma_BdRingAlloc() */
         if ((RingPtr->PreCnt < NumBd) || (RingPtr->PreHead !=
                BdSetPtr))
         {
                xdbg_printf(XDBG_DEBUG_ERROR, "Bd ring has
  ```

```
                                problems\r\n");
                        return XST_DMA_SG_LIST_ERROR;
                }
                CurBdPtr = BdSetPtr;
                BdCr = XAxiDma_BdGetCtrl(CurBdPtr);
                BdSts = XAxiDma_BdGetSts(CurBdPtr);
                /* In case of Tx channel, the first BD should have been marked
                 * as start-of-frame
                 */
                if (!(RingPtr->IsRxChannel) && !(BdCr &
                        XAXIDMA_BD_CTRL_TXSOF_MASK))
                {
                        xdbg_printf(XDBG_DEBUG_ERROR, "Tx first BD does not
                                have SOF\r\n");
                        return XST_FAILURE;
                }
                /* Clear the completed status bit
                 */
                for (i = 0; i < NumBd - 1; i++) {
                        /* Make sure the length value in the BD is non-zero. */
                        if (XAxiDma_BdGetLength(CurBdPtr,
                                        RingPtr->MaxTransferLen) == 0) {
                                xdbg_printf(XDBG_DEBUG_ERROR, "0 length bd\r\n");
                                return XST_FAILURE;
                        }
                        BdSts &=  ~XAXIDMA_BD_STS_COMPLETE_MASK;
                        XAxiDma_BdWrite(CurBdPtr, XAXIDMA_BD_STS_OFFSET,
                                BdSts);

                        /* Flush the current BD so DMA core could see the
                                updates */
                        XAXIDMA_CACHE_FLUSH(CurBdPtr);

                        CurBdPtr = (XAxiDma_Bd *)((void *)
                                XAxiDma_BdRingNext(RingPtr, CurBdPtr));
                        BdCr = XAxiDma_BdRead(CurBdPtr,
                                XAXIDMA_BD_CTRL_LEN_OFFSET);
                        BdSts = XAxiDma_BdRead(CurBdPtr,
                                XAXIDMA_BD_STS_OFFSET);
                }
                /* In case of Tx channel,the last BD should have EOF bit set*/
                if (!(RingPtr->IsRxChannel) && !(BdCr &
                                XAXIDMA_BD_CTRL_TXEOF_MASK)) {
                        xdbg_printf(XDBG_DEBUG_ERROR, "Tx last BD does not have
                                EOF\r\n");
                        return XST_FAILURE;
                }
                /* Make sure the length value in the last BD is non-zero. */
                if (XAxiDma_BdGetLength(CurBdPtr,
                                RingPtr->MaxTransferLen) == 0) {
                        xdbg_printf(XDBG_DEBUG_ERROR, "0 length bd\r\n");
                        return XST_FAILURE;
                }
                /* The last BD should also have the completed status bit
                Cleared */
                BdSts &= ~XAXIDMA_BD_STS_COMPLETE_MASK;
                XAxiDma_BdWrite(CurBdPtr, XAXIDMA_BD_STS_OFFSET, BdSts);
                /* Flush the last BD so DMA core could see the updates */
                XAXIDMA_CACHE_FLUSH(CurBdPtr);
                DATA_SYNC;
                /* This set has completed pre-processing, adjust ring
                pointers and counters */
                XAXIDMA_RING_SEEKAHEAD(RingPtr, RingPtr->PreHead, NumBd);
                RingPtr->PreCnt -= NumBd;
                RingPtr->HwTail = CurBdPtr;
                RingPtr->HwCnt += NumBd;
```

```
		/* If it is running, signal the engine to begin processing */
		if (RingPtr->RunState == AXIDMA_CHANNEL_NOT_HALTED) {
			if (RingPtr->Cyclic) {
				XAxiDma_WriteReg(RingPtr->ChanBase,
					XAXIDMA_TDESC_OFFSET,(u32)
					XAXIDMA_VIRT_TO_PHYS(RingPtr->CyclicBd));
				if (RingPtr->Addr_ext)
					XAxiDma_WriteReg(RingPtr->ChanBase,
						 XAXIDMA_TDESC_MSB_OFFSET,
						UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
						(RingPtr->CyclicBd)));
					return XST_SUCCESS;
			}
			if (RingPtr->IsRxChannel) {
				if (!RingIndex) {
					XAxiDma_WriteReg(RingPtr->ChanBase,
						XAXIDMA_TDESC_OFFSET,
						(XAXIDMA_VIRT_TO_PHYS(RingPtr->
						HwTail) &
						XAXIDMA_DESC_LSB_MASK));
					if (RingPtr->Addr_ext)
						XAxiDma_WriteReg(RingPtr->ChanBase,
							XAXIDMA_TDESC_MSB_OFFSET,
							UPPER_32_BITS
							(XAXIDMA_VIRT_TO_PHYS
							(RingPtr->HwTail)));
				}
				else {
					XAxiDma_WriteReg(RingPtr->ChanBase,
						(XAXIDMA_RX_TDESC0_OFFSET +
						(RingIndex - 1) *
						XAXIDMA_RX_NDESC_OFFSET),
						(XAXIDMA_VIRT_TO_PHYS(RingPtr-
						>HwTail) & XAXIDMA_DESC_LSB_MASK ));
					if (RingPtr->Addr_ext)
						XAxiDma_WriteReg(RingPtr->ChanBase,
							(XAXIDMA_RX_TDESC0_MSB_OFFSET
							+(RingIndex - 1) *
							XAXIDMA_RX_NDESC_OFFSET),
							UPPER_32_BITS
							(XAXIDMA_VIRT_TO_PHYS
							(RingPtr->HwTail)));
				}
			}
			else {
				XAxiDma_WriteReg(RingPtr->ChanBase,
					XAXIDMA_TDESC_OFFSET,(XAXIDMA_VIRT_TO_PHYS
					(RingPtr->HwTail) &
					XAXIDMA_DESC_LSB_MASK));
				if (RingPtr->Addr_ext)
					XAxiDma_WriteReg(RingPtr->ChanBase,
						XAXIDMA_TDESC_MSB_OFFSET,
						UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
						(RingPtr->HwTail)));
			}
		}
		return XST_SUCCESS;
	}
	-------------------------------------------------------------------
```

After allocation, we send the BDs to Hardware. All the BDs from pre-work group is send to work group. We also update the tail BD pointer, HwTail, with pointer to last BD in the work group. Since the channel is halted, we will not write to tail descriptor for the RX channel.



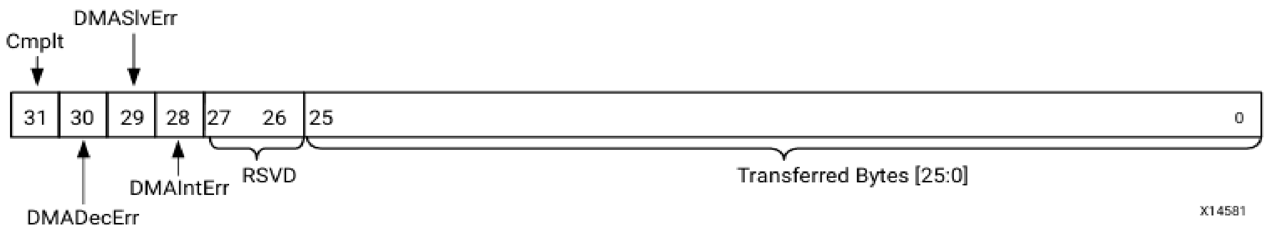
RX BD Ring [0] - ToHW

- ChanBase = RegBase + DMA_RX_OFFSET = 0x40400030
- IsRxChannel = 1
- RunState = AXIDMA_CHANNEL_HALTED = 2
- HasStsCntrlStrm = 0
- HasDRE = 0
- DataWidth = ((unsigned int)Mm2SDataWidth >> 3) = 4
- Addr_ext = 0
- MaxTransferLen =(1U << SgLengthWidth) - 1 = 16383
- FirstBdPhyAddr = PhyAddr = 0x01001000
- FirstBdAddr = VirtAddr = 0x01001000
- LastBdAddr = VirtAddr + (Sep*(BdCnt-1)) = 0x01001FC0
- Length = Last - First + Separation = 0x1000
- Separation = 64
- FreeHead = 0x01001000
- PreHead = 0x01001000
- HwHead = 0x01001000
- HwTail = 0x01001FC0 (LastAddr)
- PostHead = 0x01001000
- BdaRestart = 0x01001000
- CyclicBd = malloc(sizeof(XAxiDma_Bd))
- FreeCnt = 0
- PreCnt = 0
- HwCnt = 64
- PostCnt = 0
- AllCnt = 64
- RingIndex = 0
- Cyclic

- First of all, we store the RingIndex attribute value of the RX BD Ring into a register RingIndex. Note that this variable is used in the RX channel only.
- Check the number of BDs, if negative, return with ERROR, if it is 0, return with SUCCESS.
- Ensure that after allocation of BDs, it is not modified. This is done by checking if the PreCnt is less than number of BDs assigned. Or if the PreHead is not equal to starting pointer parameter. If any of these conditions is correct, then it will return with ERROR.
- We read the control flag bits (SOF, EOF) of the control register of 1st BD and store them in variable BdCr.

```
-----------------------------------------------------------------------------------------------
#define XAXIDMA_BD_CTRL_LEN_OFFSET      0x18
#define XAXIDMA_BD_CTRL_ALL_MASK 0x0C000000

#define XAxiDma_BdGetCtrl(BdPtr)                        \
        (XAxiDma_BdRead((BdPtr), XAXIDMA_BD_CTRL_LEN_OFFSET) \
        & XAXIDMA_BD_CTRL_ALL_MASK)
-------------------------------------------------------------
```

- We get the status bits, except the field of transferred bytes using a function.



```
-----------------------------------------------------------------------------------------------
#define XAXIDMA_BD_STS_OFFSET           0x1C   /**< Status */
#define XAXIDMA_BD_STS_ALL_MASK         0xFC000000
```

```
#define XAxiDma_BdGetSts(BdPtr)                 \
        (XAxiDma_BdRead((BdPtr), XAXIDMA_BD_STS_OFFSET) & \
        XAXIDMA_BD_STS_ALL_MASK)
--------------------------------------------------------------
```

- If the channel is TX, see it the SOF bit is set for the first BD in the set. We are in the first BD pointer. It must be set, otherwise it will return ERROR.
- Clear the complete bit which is bit 31 of status register for all the BDs. It is done using for loop, iterating over the entire number of BDs in allocated. Multiply the data in BdSts with not of COMPLETE mask which clears the complete status bit in the status register. Then write back to the status register. Update the BD pointer to next BD, update the variables BdCr, BdSts for the new BD. Wait until the entire iteration is complete, including updation for last BD.
- Now we are in the pointer to last BD in set. If the channel is TX, check if the EOF bit is set for the last BD.
- This marks the end of **Pre-Processing**. So we modify the attributes of the RX BD Ring instance. The attributes are – reduce PreCnt by NumBd, increase HwCnt by NumBd, HwTail pointer with value of last BD in the set, PreHead with new pointer to BD not in work group. Basically, all the BDs from pre-work group is now transferred into work group.
- Check if the channel is already running by looking into the Runstate attribute of RX BD Ring. Since in our case, it has not started yet, it will return with SUCCESS.
- If in case the BD Ring is in running state, that may be after Start Channel (as mentioned in the TX Setup last), then we need to write to tail descriptor with pointer to last BD, that is HwTail, as now the HwCnt is greater than 0.
- Priority for updating the tail descriptor is, first check if it is cyclic. If so, update the Tail Descriptor with physical address of CyclicBd.
- If it is not cyclic and channel is running, check if it is RX channel or TX channel. If RX, based on RingIndex value, update the Tail Descriptor with physical address of HwTail. If TX channel, update it with physical address of HwTail pointer.

So far we have seen allocation of all available BDs in free group to pre-working group. Then we updated the Buffer Address register for all BDs, Updated the control register with user defined byte length and cleared the control bits (EOF and SOF). ID register is also updated for every BDs in pre-work group.
The flow of this function in RX channel is to update the status register, especially clear the transfer byte field to 0 to remove any junk value, set the complete bit to 0 to show that no BD is completed yet. Finally we transfer all BDs from pre-work group to work group. No updation of tail descriptor is done here as channel is not running yet. Next we start the channel, by setting the current descriptor and starting the channel by setting last bit (run/stop) of control register of DMA instance.

- Last function is to start the RX BD Ring. We have already seen this in TX Setup part. So detailed explanation can be found there.

```
➔ Status = XAxiDma_BdRingStart(RxRingPtr);

❖ int XAxiDma_BdRingStart(XAxiDma_BdRing * RingPtr)
  {
          int Status;
          Status = XAxiDma_UpdateBdRingCDesc(RingPtr);
          if (Status != XST_SUCCESS) {
                  xdbg_printf(XDBG_DEBUG_ERROR, "BdRingStart: "
                 "Updating Current Descriptor Failed\n\r");
                 return Status;
          }
          Status = XAxiDma_StartBdRingHw(RingPtr);
          if (Status != XST_SUCCESS) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingStart: "
                "Starting Hardware Failed\n\r");
                return Status;
        }
        return XST_SUCCESS;
}
```

❖
```
#define XAxiDma_BdRingHwIsStarted(RingPtr)                      \
        ((XAxiDma_ReadReg((RingPtr)->ChanBase, XAXIDMA_SR_OFFSET) \
        & XAXIDMA_HALTED_MASK) ? FALSE : TRUE)
```

❖
```
#define XAxiDma_BdHwCompleted(BdPtr)                            \
        (XAxiDma_BdRead((BdPtr), XAXIDMA_BD_STS_OFFSET) &
        XAXIDMA_BD_STS_COMPLETE_MASK)
```

❖
```
int XAxiDma_UpdateBdRingCDesc(XAxiDma_BdRing* RingPtr)
{
        UINTPTR RegBase;
        UINTPTR BdPtr;
        int RingIndex = RingPtr->RingIndex;
        /* BD list has yet to be created for this channel */
        if (RingPtr->AllCnt == 0) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingStart: no ds\r\n");
                return XST_DMA_SG_NO_LIST;
        }
        /* Do nothing if already started */
        if (RingPtr->RunState == AXIDMA_CHANNEL_NOT_HALTED) {
                /* Need to update tail pointer if needed (Engine is not
                 * transferring)
                */
                return XST_SUCCESS;
        }
        if (!XAxiDma_BdRingHwIsStarted(RingPtr)) {
                /* If hardware is not running, then we need to put a valid
                current BD pointer to the current BD register before start the
                hardware
                */
                RegBase = RingPtr->ChanBase;
                /* Put a valid BD pointer in the current BD pointer register.
                So, the hardware is ready to go when tail BD pointer is
                updated */
                BdPtr = (UINTPTR)(void *)(RingPtr->BdaRestart);
                if (!XAxiDma_BdHwCompleted(BdPtr)) {
                        if (RingPtr->IsRxChannel) {
                                if (!RingIndex) {
                                        XAxiDma_WriteReg(RegBase,
                                        XAXIDMA_CDESC_OFFSET,
                                        (XAXIDMA_VIRT_TO_PHYS(BdPtr) &
                                        XAXIDMA_DESC_LSB_MASK));
                                        if (RingPtr->Addr_ext)
                                                XAxiDma_WriteReg(RegBase,
                                                XAXIDMA_CDESC_MSB_OFFSET,
                                               UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
                                                (BdPtr)));
                                }
                                else {
                                        XAxiDma_WriteReg(RegBase,
                                        (XAXIDMA_RX_CDESC0_OFFSET +
                                        (RingIndex - 1) *
                                        XAXIDMA_RX_NDESC_OFFSET),
                                        (XAXIDMA_VIRT_TO_PHYS(BdPtr) &
                                        XAXIDMA_DESC_LSB_MASK));
                                        if (RingPtr->Addr_ext)
                                                XAxiDma_WriteReg(RegBase,
                                                (XAXIDMA_RX_CDESC0_MSB_OFFSET+
                                               (RingIndex - 1)*
                                               XAXIDMA_RX_NDESC_OFFSET),
                                                UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
                          (BdPtr)));
                    }
            }
            else {
                    XAxiDma_WriteReg(RegBase,
                    XAXIDMA_CDESC_OFFSET,
                    (XAXIDMA_VIRT_TO_PHYS(BdPtr) &
                    XAXIDMA_DESC_LSB_MASK));
                    if (RingPtr->Addr_ext)
                            XAxiDma_WriteReg(RegBase,XAXIDMA_CDESC_MSB
                            _OFFSET,UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
                            (BdPtr)));
            }
    }
    else {
            /* Look for an uncompleted BD
            */
            while (XAxiDma_BdHwCompleted(BdPtr)) {
                    BdPtr = XAxiDma_BdRingNext(RingPtr, BdPtr);

                    if ((UINTPTR)BdPtr == (UINTPTR) RingPtr-
                    >BdaRestart) {
                            xdbg_printf(XDBG_DEBUG_ERROR,
                            "StartBdRingHw: Cannot find valid
                            cdesc\r\n");
                            return XST_DMA_ERROR;
                    }
                    if (!XAxiDma_BdHwCompleted(BdPtr)) {
                            if (RingPtr->IsRxChannel) {
                                    if (!RingIndex) {
                                            XAxiDma_WriteReg(RegBase,
                                            XAXIDMA_CDESC_OFFSET,
                                            (XAXIDMA_VIRT_TO_PHYS(BdPtr)&
                                            XAXIDMA_DESC_LSB_MASK));
                                            if (RingPtr->Addr_ext)
                                                    XAxiDma_WriteReg(RegBase,
                                                    XAXIDMA_CDESC_MSB_OFFSET,
                                                    UPPER_32_BITS(
                                                    XAXIDMA_VIRT_TO_PHYS
                                                    (BdPtr)));
                                    }
                                    else {
                                            XAxiDma_WriteReg(RegBase,
                                            (XAXIDMA_RX_CDESC0_OFFSET +
                                            (RingIndex - 1) *
                                            XAXIDMA_RX_NDESC_OFFSET),
                                            (XAXIDMA_VIRT_TO_PHYS(BdPtr)
                                            & XAXIDMA_DESC_LSB_MASK));
                                            if (RingPtr->Addr_ext)
                                                    XAxiDma_WriteReg(RegBase,
                                                   (XAXIDMA_RX_CDESC0_MSB_OFFSET
                                                     +(RingIndex - 1) *
                                                    XAXIDMA_RX_NDESC_OFFSE),
                                                    UPPER_32_BITS(
                                                    XAXIDMA_VIRT_TO_PHYS
                                                    (BdPtr)));
                                    }
                            }
                            else {
                                    XAxiDma_WriteReg(RegBase,
                                            XAXIDMA_CDESC_OFFSET,
                                            (XAXIDMA_VIRT_TO_PHYS(BdPtr)&
                                            XAXIDMA_DESC_LSB_MASK));
                                            if (RingPtr->Addr_ext)
                                                    XAxiDma_WriteReg
                                                    (RegBase,
                                                     XAXIDMA_CDESC_MSB_OFFSET,
                                                UPPER_32_BITS(
                                                XAXIDMA_VIRT_TO_PHYS
                                                (BdPtr)));
                                    }
                                    break;
                              }
                        }
                  }
            }
            return XST_SUCCESS;
    }
```

❖
```
#define XAxiDma_BdRingHwIsStarted(RingPtr)                          \
        ((XAxiDma_ReadReg((RingPtr)->ChanBase, XAXIDMA_SR_OFFSET) \
        & XAXIDMA_HALTED_MASK) ? FALSE : TRUE)
```

❖
```
int XAxiDma_StartBdRingHw(XAxiDma_BdRing * RingPtr)
{
        UINTPTR RegBase;
        int RingIndex = RingPtr->RingIndex;
        if (!XAxiDma_BdRingHwIsStarted(RingPtr)) {
                /* Start the hardware
                */
                RegBase = RingPtr->ChanBase;
                XAxiDma_WriteReg(RegBase, XAXIDMA_CR_OFFSET,
                        XAxiDma_ReadReg(RegBase, XAXIDMA_CR_OFFSET)
                        | XAXIDMA_CR_RUNSTOP_MASK);
        }

        if (XAxiDma_BdRingHwIsStarted(RingPtr)) {
                /* Note as started */
                RingPtr->RunState = AXIDMA_CHANNEL_NOT_HALTED;
                /* If there are unprocessed BDs then we want the channel to
                 begin processing right away */
                if (RingPtr->HwCnt > 0) {
                        XAXIDMA_CACHE_INVALIDATE(RingPtr->HwTail);
                        if (RingPtr->Cyclic) {
                                XAxiDma_WriteReg(RingPtr->ChanBase,
                                         XAXIDMA_TDESC_OFFSET,
                                        (u32)XAXIDMA_VIRT_TO_PHYS(RingPtr-
                                        >CyclicBd));
                                if (RingPtr->Addr_ext)
                                        XAxiDma_WriteReg(RingPtr->ChanBase,
                                                XAXIDMA_TDESC_MSB_OFFSET,
                                                UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
                                                (RingPtr->CyclicBd)));
                                return XST_SUCCESS;
                        }
                        if ((XAxiDma_BdRead(RingPtr->HwTail,
                                XAXIDMA_BD_STS_OFFSET) &
                                XAXIDMA_BD_STS_COMPLETE_MASK) == 0) {
                                if (RingPtr->IsRxChannel) {
                                        if (!RingIndex) {
                                                XAxiDma_WriteReg(RingPtr->ChanBase,
                                                        XAXIDMA_TDESC_OFFSET,
                                                        (XAXIDMA_VIRT_TO_PHYS(RingPtr
                                                        ->HwTail) &
                                                        XAXIDMA_DESC_LSB_MASK));
                                                if (RingPtr->Addr_ext)
                                                        XAxiDma_WriteReg(RingPtr-
                                                        >ChanBase,
                                                        XAXIDMA_TDESC_MSB_OFFSET,
                                                        UPPER_32_BITS
                                                        (XAXIDMA_VIRT_TO_PHYS(RingPtr
                                                        ->HwTail)));
                                        }
                                        else {
```

```
                                                            XAxiDma_WriteReg(RingPtr->ChanBase,
                                                                    (XAXIDMA_RX_TDESC0_OFFSET +
                                                                    (RingIndex - 1) *
                                                                    XAXIDMA_RX_NDESC_OFFSET),
                                                                    (XAXIDMA_VIRT_TO_PHYS(RingPtr-
                                                                    >HwTail) &
                                                                    XAXIDMA_DESC_LSB_MASK ));
                                                            if (RingPtr->Addr_ext)
                                                                    XAxiDma_WriteReg(RingPtr-
                                                                    >ChanBase,
                                                                    (XAXIDMA_RX_TDESC0_MSB_OFFSET +
                                                                    (RingIndex - 1) *
                                                                    XAXIDMA_RX_NDESC_OFFSET),
                                                                    UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
                                                                    (RingPtr->HwTail)));
                                                    }
                                            }
                                            else {
                                                    XAxiDma_WriteReg(RingPtr->ChanBase,
                                                            XAXIDMA_TDESC_OFFSET,
                                                            (XAXIDMA_VIRT_TO_PHYS(RingPtr-
                                                            >HwTail) & XAXIDMA_DESC_LSB_MASK));
                                                    if (RingPtr->Addr_ext)
                                                            XAxiDma_WriteReg(RingPtr->ChanBase,
                                                            XAXIDMA_TDESC_MSB_OFFSET,
                                                            UPPER_32_BITS(XAXIDMA_VIRT_TO_PHYS
                                                            (RingPtr->HwTail)));
                                            }
                                    }
                            }
                            return XST_SUCCESS;
                    }
                    return XST_DMA_ERROR;
            }

            ------------------------------------------------------------------------
```

In case of this RX Setup function, once all BDs are allocated, prepared, pre-processed and send to work group, we call the function XAxiDma_BdRingStart() to start the RX BD Ring. It has 2 functions, namely, XAxiDma_UpdateBdRingCDesc() and XAxiDma_StartBdRingHw().
The first function updates the Current descriptor of RX channel (S2MM) of DMA instance. Since the channel is not in running state now, we will check if the first available BD in the Ring, BdaRestart pointer, pointing to 1st BD in the Ring is completed or not by checking the complete bit of status register of BD. Since it is RX Channel and RingIndex is 0, we will update the current descriptor with physical address of first BD in the Ring.
The second function checks if the channel is started. Since its not started yet, we start the channel by writing into last bit of control register for the RX BD Ring. Since the HwCnt is non-zero, we will update the tail descriptor with physical address of HwTail pointer BD.

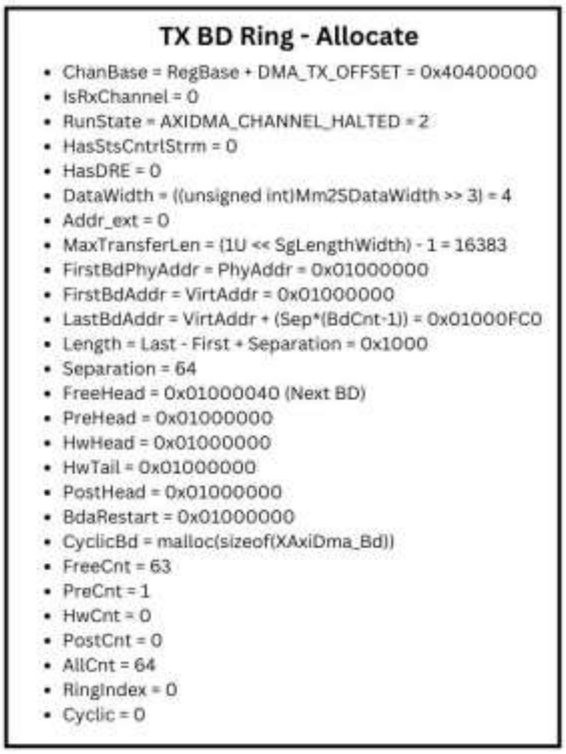
7. SEND PACKET – START TX

We have setup both TX and RX channels. Now we need to send data over the TX channel. It is handled by function SendPacket(). The parameter is the address to DMA instance. The main task of this function is to send a packet of data over MM2S (TX) channel. We use only 1 BD to send the packet.

```
➔ Status = SendPacket(&AxiDma);

  static int SendPacket(XAxiDma * AxiDmaInstPtr)
  {
          XAxiDma_BdRing *TxRingPtr;
          u8 *TxPacket;
```

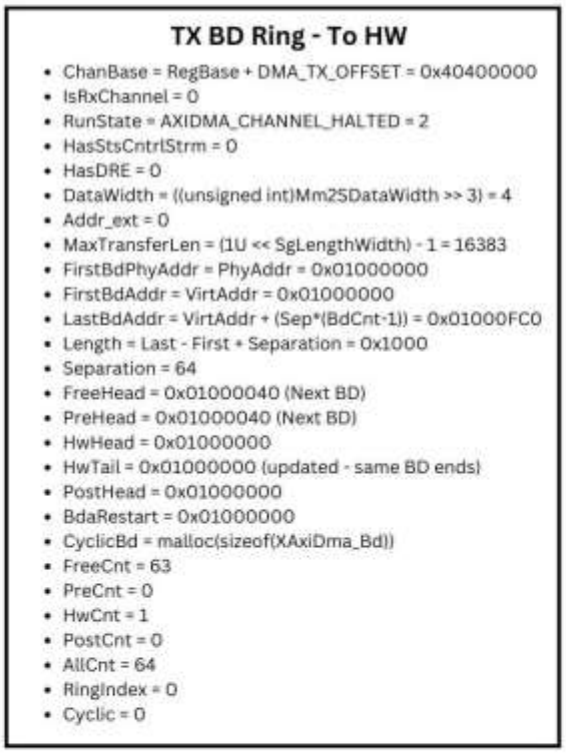
```
            u8 Value;
            XAxiDma_Bd *BdPtr;
            int Status;
            int Index;
            TxRingPtr = XAxiDma_GetTxRing(AxiDmaInstPtr);
            /* Create pattern in the packet to transmit */
            TxPacket = (u8 *) Packet;
            Value = TEST_START_VALUE;
            for(Index = 0; Index < MAX_PKT_LEN; Index ++) {
                  TxPacket[Index] = Value;
                  Value = (Value + 1) & 0xFF;
            }
            /* Flush the SrcBuffer before the DMA transfer, in case the Data
            Cache is enabled */
            Xil_DCacheFlushRange((UINTPTR)TxPacket, MAX_PKT_LEN);
#ifdef __aarch64__
      Xil_DCacheFlushRange((UINTPTR)RX_BUFFER_BASE, MAX_PKT_LEN);
#endif
            /* Allocate a BD */
            Status = XAxiDma_BdRingAlloc(TxRingPtr, 1, &BdPtr);
            if (Status != XST_SUCCESS) {
                  return XST_FAILURE;
            }
            /* Set up the BD using the information of the packet to transmit */
            Status = XAxiDma_BdSetBufAddr(BdPtr, (UINTPTR) Packet);
            if (Status != XST_SUCCESS) {
                  xil_printf("Tx set buffer addr %x on BD %x failed %d\r\n",
                   (UINTPTR)Packet, (UINTPTR)BdPtr, Status);

                  return XST_FAILURE;
            }
            Status = XAxiDma_BdSetLength(BdPtr, MAX_PKT_LEN,
                        TxRingPtr->MaxTransferLen);
            if (Status != XST_SUCCESS) {
                  xil_printf("Tx set length %d on BD %x failed %d\r\n",
                  MAX_PKT_LEN, (UINTPTR)BdPtr, Status);
                  return XST_FAILURE;
            }
#if (XPAR_AXIDMA_0_SG_INCLUDE_STSCNTRL_STRM == 1)
      Status = XAxiDma_BdSetAppWord(BdPtr,
          XAXIDMA_LAST_APPWORD, MAX_PKT_LEN);
      /* If Set app length failed, it is not fatal
       */
      if (Status != XST_SUCCESS) {
            xil_printf("Set app word failed with %d\r\n", Status);
      }
#endif
            /* For single packet, both SOF and EOF are to be set
             */
            XAxiDma_BdSetCtrl(BdPtr, XAXIDMA_BD_CTRL_TXEOF_MASK |
                                    XAXIDMA_BD_CTRL_TXSOF_MASK);
            XAxiDma_BdSetId(BdPtr, (UINTPTR)Packet);
            /* Give the BD to DMA to kick off the transmission. */
            Status = XAxiDma_BdRingToHw(TxRingPtr, 1, BdPtr);
            if (Status != XST_SUCCESS) {
                  xil_printf("to hw failed %d\r\n", Status);
                  return XST_FAILURE;
            }
            return XST_SUCCESS;
      }
      ---------------------------------------------------------------------------
```

After TX BD Ring allocation, 1 of the Free BDs will be send to pre-work group and FreeHead is updated with pointer to second BD in Ring. After sending to HW, pre-work BD is send to work group, updating PreHead with second BD. HwTail is same pointer of 1st BD as it end in the 1st BD itself.



TX BD Ring - Allocate

- ChanBase = RegBase + DMA_TX_OFFSET = 0x40400000
- IsRxChannel = 0
- RunState = AXIDMA_CHANNEL_HALTED = 2
- HasStsCntrlStrm = 0
- HasDRE = 0
- DataWidth = ((unsigned int)Mm2SDataWidth >> 3) = 4
- Addr_ext = 0
- MaxTransferLen = (1U << SgLengthWidth) - 1 = 16383
- FirstBdPhyAddr = PhyAddr = 0x01000000
- FirstBdAddr = VirtAddr = 0x01000000
- LastBdAddr = VirtAddr + (Sep*(BdCnt-1)) = 0x01000FC0
- Length = Last - First + Separation = 0x1000
- Separation = 64
- FreeHead = 0x01000040 (Next BD)
- PreHead = 0x01000000
- HwHead = 0x01000000
- HwTail = 0x01000000
- PostHead = 0x01000000
- BdaRestart = 0x01000000
- CyclicBd = malloc(sizeof(XAxiDma_Bd))
- FreeCnt = 63
- PreCnt = 1
- HwCnt = 0
- PostCnt = 0
- AllCnt = 64
- RingIndex = 0
- Cyclic = 0



TX BD Ring - To HW

- ChanBase = RegBase + DMA_TX_OFFSET = 0x40400000
- IsRxChannel = 0
- RunState = AXIDMA_CHANNEL_HALTED = 2
- HasStsCntrlStrm = 0
- HasDRE = 0
- DataWidth = ((unsigned int)Mm2SDataWidth >> 3) = 4
- Addr_ext = 0
- MaxTransferLen = (1U << SgLengthWidth) - 1 = 16383
- FirstBdPhyAddr = PhyAddr = 0x01000000
- FirstBdAddr = VirtAddr = 0x01000000
- LastBdAddr = VirtAddr + (Sep*(BdCnt-1)) = 0x01000FC0
- Length = Last - First + Separation = 0x1000
- Separation = 64
- FreeHead = 0x01000040 (Next BD)
- PreHead = 0x01000040 (Next BD)
- HwHead = 0x01000000
- HwTail = 0x01000000 (updated - same BD ends)
- PostHead = 0x01000000
- BdaRestart = 0x01000000
- CyclicBd = malloc(sizeof(XAxiDma_Bd))
- FreeCnt = 63
- PreCnt = 0
- HwCnt = 1
- PostCnt = 0
- AllCnt = 64
- RingIndex = 0
- Cyclic = 0

- We get the pointer to TX BD Ring from the DMA instance, which is stored into variable TxRingPtr.
- Packet is a 32 bit pointer to TX_BUFFER_BASE, from where we will be sending data. But in this function, since we want to write to every byte and not word, we define another 8 bit pointer, TxPacket, pointing to starting address of TX buffer base address.
- Using a for loop, we update all bytes of the buffer address, upto required value, which is MAX_PKT_LEN whose value is 32 bytes. We assign known values here.
- We allocate 1 BD from the TX BD Ring for transmission purpose. The function used in XAxiDma_BdRingAlloc(). It is same as what we have seen in RX Setup. This function assigns 1 BD from free group to pre-work group, updating pointer FreeHead, counters FreeCnt and PreCnt. It also returns the pointer to this 1 allocated BD as output parameter.
- Update the buffer address for this single BD with Packet, pointing to TX_BUFFER_BASE.
- Set the length field of the control register using user defined value, which is MAX_PKT_LEN (32 bytes).
- Set both control bits (SOF and EOF) for this BD in the control register. Since this BD send a single packet of data, both must be set, otherwise the initial BD should have SOF active and last BD with EOF active.
- Update the ID register for this BD with Packet pointer value.
- Finally, we send this BD to hardware. It will check if SOF and EOF is set for this single BD, set the complete bit to 0 in the status register of this BD, reset the byte length field to 0 within the status register.
- Send this BD to work group from pre-work group, updating attributes of TX BD Ring such as HwTail pointer, PreHead pointer, PreCnt and HwCnt counters.
- Since the TX channel is running from the TX Setup, we update the tail descriptor with physical address of next (2nd) BD.

This should start sending data over MM2S channel.

8. CHECK DATA TRANSFER IN LOOP BACK

Last function in main function is to check if the received signal is same as transmitted signal. So it will wait until DMA transaction is finished, check data and clean up.

```
static int CheckDmaResult(XAxiDma * AxiDmaInstPtr)
{
	XAxiDma_BdRing *TxRingPtr;
	XAxiDma_BdRing *RxRingPtr;
	XAxiDma_Bd *BdPtr;
	int ProcessedBdCount;
	int FreeBdCount;
	int Status;

	TxRingPtr = XAxiDma_GetTxRing(AxiDmaInstPtr);
	RxRingPtr = XAxiDma_GetRxRing(AxiDmaInstPtr);
	/* Wait until the one BD TX transaction is done */
	while ((ProcessedBdCount = XAxiDma_BdRingFromHw(TxRingPtr,
						XAXIDMA_ALL_BDS,&BdPtr)) == 0) {

	}
	/* Free all processed TX BDs for future transmission */
	Status = XAxiDma_BdRingFree(TxRingPtr, ProcessedBdCount, BdPtr);
	if (Status != XST_SUCCESS) {
		xil_printf("Failed to free %d tx BDs %d\r\n",
			ProcessedBdCount, Status);
		return XST_FAILURE;
	}
	/* Wait until the data has been received by the Rx channel */
	while ((ProcessedBdCount = XAxiDma_BdRingFromHw(RxRingPtr,
						XAXIDMA_ALL_BDS, &BdPtr)) == 0) {

	}
	/* Check received data */
	if (CheckData() != XST_SUCCESS) {
		return XST_FAILURE;
	}
	/* Free all processed RX BDs for future transmission */
	Status = XAxiDma_BdRingFree(RxRingPtr, ProcessedBdCount, BdPtr);
	if (Status != XST_SUCCESS) {
		xil_printf("Failed to free %d rx BDs %d\r\n",
			ProcessedBdCount, Status);
		return XST_FAILURE;
	}
	/* Return processed BDs to RX channel so we are ready to receive new
	 * packets:
	 *    - Allocate all free RX BDs
	 *    - Pass the BDs to RX channel
	 */
	FreeBdCount = XAxiDma_BdRingGetFreeCnt(RxRingPtr);
	Status = XAxiDma_BdRingAlloc(RxRingPtr, FreeBdCount, &BdPtr);
	if (Status != XST_SUCCESS) {
		xil_printf("bd alloc failed\r\n");
		return XST_FAILURE;
	}

	Status = XAxiDma_BdRingToHw(RxRingPtr, FreeBdCount, BdPtr);
	if (Status != XST_SUCCESS) {
		xil_printf("Submit %d rx BDs failed %d\r\n",FreeBdCount,
			Status);
		return XST_FAILURE;
	}
	return XST_SUCCESS;
}
--------------------------------------------------------------------------
```

- The first step of the function CheckDmaResult() is to take the pointers to TX and RX BD Ring and store them in variables TxRingPtr and RxRingPtr, respectively.

- We wait in the while loop until 1 BD transaction is done. It uses the function XAxiDma_BdRingFromHw(), which takes in parameters – pointer to a BD Ring, which is TX channel in this case; maximum number of BDs to be returned in the set, since we want to return all BDs, we use XAXIDMA_ALL_BDS, so no specific number need to be mentioned; and last parameter is output, which points to first BD to be examined after hardware processing.

```
-------------------------------------------------------------------------------------------------------
int XAxiDma_BdRingFromHw(XAxiDma_BdRing * RingPtr, int BdLimit,
             XAxiDma_Bd ** BdSetPtr)
{
      XAxiDma_Bd *CurBdPtr;
      int BdCount;
      int BdPartialCount;
      u32 BdSts;
      u32 BdCr;
      CurBdPtr = RingPtr->HwHead;
      BdCount = 0;
      BdPartialCount = 0;
      BdSts = 0;
      BdCr = 0;

      /* If no BDs in work group, then there's nothing to search */
      if (RingPtr->HwCnt == 0) {
            *BdSetPtr = (XAxiDma_Bd *)NULL;
            return 0;
      }

      if (BdLimit > RingPtr->HwCnt) {
            BdLimit = RingPtr->HwCnt;
      }
      /* Starting at HwHead, keep moving forward in the list until:
      *   - A BD is encountered with its completed bit clear in the
      Status word which means hardware has not completed processing
      of that BD.
      *   - RingPtr->HwTail is reached
      *   - The number of requested BDs has been processed
      */

      while (BdCount < BdLimit) {
            /* Read the status */
            XAXIDMA_CACHE_INVALIDATE(CurBdPtr);
            BdSts = XAxiDma_BdRead(CurBdPtr, XAXIDMA_BD_STS_OFFSET);
            BdCr = XAxiDma_BdRead(CurBdPtr, XAXIDMA_BD_CTRL_LEN_OFFSET);

            /* If the hardware still hasn't processed this BD then we are
            * done
            */
            if (!(BdSts & XAXIDMA_BD_STS_COMPLETE_MASK)) {
                  break;
            }

            BdCount++;

            /* Hardware has processed this BD so check the "last" bit. If
            * it is clear,then there are more BDs for the current packet.
            * Keep a count of these partial packet BDs.
            *
            * For tx BDs, EOF bit is in the control word
            * For rx BDs, EOF bit is in the status word
```

```
        */
        if (((!(RingPtr->IsRxChannel) &&
            (BdCr & XAXIDMA_BD_CTRL_TXEOF_MASK)) ||
                ((RingPtr->IsRxChannel) && (BdSts &
                XAXIDMA_BD_STS_RXEOF_MASK)))) {
                BdPartialCount = 0;
        }
        else {
            BdPartialCount++;
        }

        if (RingPtr->Cyclic) {
            BdSts = BdSts & ~XAXIDMA_BD_STS_COMPLETE_MASK;
            XAxiDma_BdWrite(CurBdPtr, XAXIDMA_BD_STS_OFFSET,
                BdSts);
            XAXIDMA_CACHE_FLUSH(CurBdPtr);
        }
        /* Reached the end of the work group */
        if (CurBdPtr == RingPtr->HwTail) {
            break;
        }
        /* Move on to the next BD in work group */
        CurBdPtr = (XAxiDma_Bd *)((void
            *)XAxiDma_BdRingNext(RingPtr, CurBdPtr));
    }

    /* Subtract off any partial packet BDs found */
    BdCount -= BdPartialCount;

    /* If BdCount is non-zero then BDs were found to return. Set return
     * parameters, update pointers and counters, return success
     */
    if (BdCount) {
        *BdSetPtr = RingPtr->HwHead;
        if (!RingPtr->Cyclic) {
            RingPtr->HwCnt -= BdCount;
            RingPtr->PostCnt += BdCount;
        }
        XAXIDMA_RING_SEEKAHEAD(RingPtr, RingPtr->HwHead, BdCount);
        return BdCount;
    }
    else {
        *BdSetPtr = (XAxiDma_Bd *)NULL;
        return 0;
    }
}
--------------------------------------------------------------------
```
For the TX channel, we already know that the allocated BD/s are in work group as indicated by HwCnt value. So we take the current BD pointer value as the pointer value of HwHead (which is basically the first BD).
Check if the HwCnt is 0, if so return 0.

Since we have specified the BdLimit to max possible value, we will update this value with attribute HwCnt, as it represents how many BDs are there in hardware.
We search through all the BDs in the work group and it is stopped under certain conditions:
Condition 1 - unless we encounter any BD which is not yet finished processing by hardware.
Condition 2 - all the BDs in the work group is checked.
Condition 3 - requested number of BDs has been reached.

It is done by iterating each BDs In the work group until it ends (upto HwCnt).

Inside this while loop, we will read the status and control registers of this current BD, after cache invalidating, and check the complete bit (31) of status register. If it detects an incomplete BD, it break out of while loop (condition 1).
We update the BdCount variable.
Since the complete bit of status register for the current BD is set, we know the hardware processing for that BD is over, but we are not sure whether a packet is spread across multiple BDs. It is checked by looking into EOF bit, depending on channel we use. For TX, it is read from control register as we set it according to data we send. For RX channel, we read it from status register, as now it will be set by application.
Using an if condition, we check if given channel is RX or TX by reading the IsRxChannel attribute of BD Ring. If TX, check EOF mask value for control register and if RX channel, check EOF mask value for status register. If these conditions pass, we keep the BdPartialCount variable as 0. This means that packet is finished within that BD itself. If it fails, we increment the BdPartialCount variable to keep count of how many BDs store that packet. Since in while loop we iterate over all processed BDs, multiple BDs, we check for the EOF bit.
If it is a cyclic BD, we modify the read status register, clear the complete bit (31) and rewrite it back to status register for further processing.
Check if the current BD is the last BD in work group. If so, break from while loop (condition 2).
Update the current BD pointer with next BD pointer. Iterate through while loop until it is finished (condition 3).

Subtract the partial BD count value from total BD count which is most probably HwCnt. It is to be noted that if multiple BDs represent a packet, on completion of transfer, partialBdCount register is made 0. So it will have some value only if we process some BDs of a packet and returned from while loop before last BD of that packet is detected. Thus last few BDs belonging to incomplete packet should not be returned. That's why we subtract them from total count. If the BdCount is positive number, then we return the output parameter with pointer to the 1st BD in work group, which is HwHead.

Also check if the BD Ring is cyclic or not. If cyclic and BdCount is non-zero, we will not pass the work group BDs to post-work group. Else we move them and it is done by updating the HwCnt with –BdCount and PostCnt with +BdCount. Also update the HwHead with BdCount BDs apart.

If the BdCount is 0, return 0 and thus it will be stuck in while loop.

NOTE: In cyclic mode, once the BDs are available in work group, it will not be further taken into post-work group. The DMA engine will repeatedly process the same set of BDs in circular fashion, for both TX and RX channels. As long as DMA is active and transfers are going, DMA will continuously reprocess the BD set.
In case of non-cyclic BD Ring, after all BDs in work group is processed, it will move to post-work group and free group, as it is a one-time transfer scenario.

- Next task is to free the processed BDs so they can be used for other transfers. We use the function XAxiDma_BdRingFree() with parameters – pointer to BD Ring, number of BDs to be freed from post-work group and pointer to first BD in the set returned from work group.

```
int XAxiDma_BdRingFree(XAxiDma_BdRing * RingPtr, int NumBd,
        XAxiDma_Bd * BdSetPtr)
{
        if (NumBd < 0) {
                xdbg_printf(XDBG_DEBUG_ERROR,
                        "BdRingFree: negative BDs %d\r\n", NumBd);
                return XST_INVALID_PARAM;
        }
        /* If the BD Set to free is empty, do nothing
        */
        if (NumBd == 0) {
                return XST_SUCCESS;
        }
        /* Make sure we are in sync with XAxiDma_BdRingFromHw() */
        if ((RingPtr->PostCnt < NumBd) || (RingPtr->PostHead !=
        BdSetPtr)) {
                xdbg_printf(XDBG_DEBUG_ERROR, "BdRingFree: Error free
                        BDs: post count %d to free %d, PostHead %x to
                        free ptr %x\r\n",RingPtr->PostCnt, NumBd,
                        (UINTPTR)RingPtr->PostHead,(UINTPTR)BdSetPtr);
                return XST_DMA_SG_LIST_ERROR;
        }
        /* Update pointers and counters */
        RingPtr->FreeCnt += NumBd;
        RingPtr->PostCnt -= NumBd;
        XAXIDMA_RING_SEEKAHEAD(RingPtr, RingPtr->PostHead, NumBd);
        return XST_SUCCESS;
}
```

Checks the number of BDs to be freed, ensuring it is a valid number and if 0, return without doing anything. Make sure that nothing has changed after FromHw function. Pass all these BDs into free group and remove them from post-work group. It is done my modifying the FreeCnt, PostCnt, PostHead attributes of the BD Ring of the channel provided.

- For the RX channel, do the function FromHw() which we have already seen for TX channel.
- Check the received data.

```
static int CheckData(void)
{
        u8 *RxPacket;
        int Index = 0;
        u8 Value;
        RxPacket = (u8 *) RX_BUFFER_BASE;
        Value = TEST_START_VALUE;
        /* Invalidate the DestBuffer before
        receiving the data, in case the Data Cache is enabled*/
#ifndef __aarch64__
        Xil_DCacheInvalidateRange((UINTPTR)RxPacket, MAX_PKT_LEN);
#endif
        for(Index = 0; Index < MAX_PKT_LEN; Index++) {
                if (RxPacket[Index] != Value) {
                        xil_printf("Data error %d: %x/%x\r\n",
                         Index, (unsigned int)RxPacket[Index],
                        (unsigned int)Value);
                        return XST_FAILURE;
                }
                Value = (Value + 1) & 0xFF;
        }
        return XST_SUCCESS;
}
```

This function defines an 8 bit pointer, pointing to the location where we will store the data. Using for loop, it iterates through all bytes in the RX storage and compare with transmitted data. If all are matching, returns from the function with success.

- Free all returned BDs from RX channel using the function BdRingFree() which we have seen previously.
- Finally, we get the FreeBdCount of the RX channel and again allocate it next reception. Then pass these allocated BDs to HardWare.

RETURN from main function. End of example code.