# Implementing Multi-channel DMA with the GN412x IP

## Application Note

# Contents


1. Related Documentation ..... 3
2. Overview ..... 4
    2.1 Getting Started ..... 5
3. Multi-Channel DMA Hardware Design ..... 6
    3.1 DMA Hardware Operation ..... 7
        3.1.1 DMA Data Buffering ..... 7
        3.1.2 DMA Transfer Size ..... 7
    3.2 Source FIFO Design (L2P Direction) ..... 9
    3.3 Sink FIFO Design (P2L Direction) ..... 9
        3.3.1 Support for Multiple Outstanding Reads ..... 9
        3.3.2 Implementation of Reordering Hardware ..... 10
4. Multi-Channel DMA Software Design ..... 12
    4.1 DMA Programming ..... 12
        4.1.1 Data Structures for Scatter/Gather Operation ..... 12
            Extended SG Data Structure ..... 13
            Accommodating Large DMA Channel Count ..... 13
            DMA End Cases ..... 14
        4.1.2 VDMA Sequencer Programming ..... 14
            Channel Scheduling ..... 15
            DMA Channel Servicing ..... 15
            High-Priority Channels ..... 17
    4.2 Hardware/Software Interaction ..... 18
        4.2.1 Semaphore Mechanisms ..... 19
            EVENT Interrupts ..... 19
            Host Polling of the Endpoint ..... 19
            Host Memory Semaphores ..... 20
    4.3 Host Software ..... 22
        4.3.1 Scatter/Gather List Management ..... 22
            Fixed DMA List ..... 22
            Dynamic DMA List ..... 22
5. DMA Coding Examples and Best Practices ..... 25
    5.1 3-DW Descriptor Processing ..... 25
    5.2 4-DW Descriptor Processing ..... 27
        5.2.1 Example: Large Contiguous Host Buffer ..... 30
6. Conclusion ..... 36


# Revision History

| Version | ECR | Date | Changes and / or Modifications |
|---|---|---|---|
| 0 | 153259 | Dec. 2009 | New document. |

# 1. Related Documentation

**Industry Standards:**

- PCI Express Base Specification Revision 1.1, PCI-SIG, March28, 2005

**Related Gennum Documentation:**

- GN412x FlexDMA Sequencer Design Guide, Document ID: 52179
- GN4124 x4 Lane PCI Express to Local Bus Bridge Data Sheet, Document ID: 48407
- GN4121 x1 Lane PCI Express to Local Bus Bridge Data Sheet, Document ID: 51539
- GN412x PCI Express Family Reference Manual, Document ID: 52624
- GN4124 Gullwing RDK User Guide, Document ID: 50932
- GN4121 Gullwing-x1 RDK User Guide, Document ID: 52162
- GN412x RDK Software Design Guide, Document ID:51859
- GN412x FPGA IP Hardware Application Note, Document ID: 51860
- GN412x Simulation Test Bench User Guide, Document ID: 53716

# 2. Overview

The purpose of this application note is to document the multi-channel DMA use model for the GN412x IP core supplied by Gennum from both the hardware and software perspectives.

The GN412x IP core (Figure 2-1)contains the FlexDMA IP core which is a highly flexible and scalable DMA controller. FlexDMA enables up to 512 DMA channels to be accommodated. There may be a small number of high bandwidth channels or a large number of lower bandwidth channels. The limitation is the available bandwidth of the GN412x, host system (root complex), and any intermediate switches.

The internal operation of the VDMA Sequencer is describe in the document "GN412x FlexDMA Sequencer Design Guide" and familiarity with that document will be assumed. This document builds on that to describe how to program the VDMA sequencer and develop a driver/application code for the purpose of multi-channel DMA.

**Figure 2-1: Gennum Local Bus IP Core with FlexDMA**

## 2.1 Getting Started

In order to illustrate multi-channel DMA operation, this document shall refer to the "Lambo" project which consist of the standard Gennum FPGA IP together with a simple application layer containing counters for DMA data sinking and sourcing.

The lambo project is designed to run on the Gullwing RDK cards (either the 4-lane or 1-lane version) and is available in the FPGA IP release found on MyGennum.com under the Gullwing RDK directory. It is available for both Altera and Xilinx platforms. This IP should be downloaded from MyGennum.com and extracted.

Additional files and documents should be downloaded from MyGennum for reference:

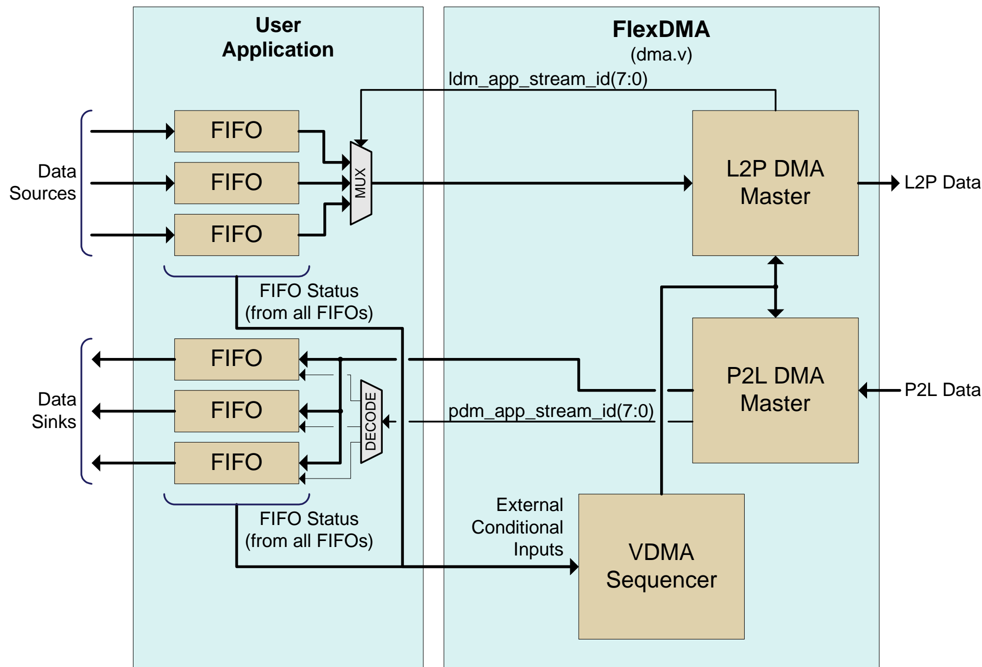
- The GN412x RDK software
- GN412x FlexDMA Sequencer Design Guide, Document ID: 52179
- GN412x PCI Express Family Reference Manual, Document ID: 52624
- GN4124 Gullwing RDK User Guide, Document ID: 50932 or GN4121 Gullwing-x1 RDK User Guide, Document ID: 52162
- GN412x RDK Software Application Note, Document ID:51859
- GN412x FPGA IP Hardware Application Note, Document ID: 51860

# 3. Multi-Channel DMA Hardware Design

The GN412x IP is designed to accommodate an application layer that requires multi-channel DMA capabilities. This could include multiple channels of DMA in each direction or all in a single direction. Since the IP is provided in source format, users are free to modify the blocks inside the Application Attachment Layer code to suit their needs. However, this should not be necessary in most cases.

Figure 3-1 depicts how multiple DMA sources/sinks are connected through the User Application Layer (UAL) to the FlexDMA block inside the Gennum IP.

**Figure 3-1: Interfacing Multiple DMA Streams to the FlexDMA Block**




The FlexDMA block has a single low-level DMA master for each direction of data flow. Multiple channels of DMA are accommodated by time multiplexing the use of the L2P and P2L DMA masters. Since there can be only a single packet traversing the GN412x local bus in each direction at any one instant in time, there is no need to have more than a single DMA master in each direction.

It is the function of the VDMA sequencer to "virtualize" the DMA masters so that they can behave as multi-channel DMA[1]. This is accomplished by intelligently scheduling a sequence of small DMA operations using microcode loaded into the Descriptor RAM inside the VDMA sequencer (this is described later). In order for hardware to accommodate the use of the sequencer and DMA, a User Application must have:

- Each independent or asynchronous data source or sink shall be buffered through a small FIFO.[2]

1. The "V" in VDMA stands for "virtual".

- For the L2P direction, a data selector to select one of the data source FIFO outputs
- For the P2L direction, a decoder to select one of the data sink FIFO inputs.
- Each FIFO shall produce a “watermark” signal that is connected to the conditional input signals of the FlexDMA controller. This watermark is effectively a DMA request signal. The DMA request watermark shall indicate to the VDMA sequencer that a DMA operation should be scheduled.

**Important Note:** It is important to prevent stalling of the DMA controllers. Consequently, an L2P DMA should not be initiated unless the application layer has all the data required for a given transfer length (up to 4K). Otherwise, the L2P bus will be occupied with idle cycles until the remainder of the data he is ready. When the L2P bus is idle in this way there is no opportunity for other data transfer or for P2L DMA read requests.

## 3.1 DMA Hardware Operation

It is important to explain a few basic principles of operation of the FlexDMA controller.

### 3.1.1 DMA Data Buffering

A DMA should not be initiated unless the full size of the DMA request can be accommodated. That is because the DMA block itself does not buffer data. Data buffering is done in the application layer so that buffering resources can be optimized according to the application.

For example, if a L2P DMA of 4KB transfer size is initiated, the FlexDMA controller should be able to pull the data from a FIFO in the application layer at full bus speed without wait states. While it is possible for the application layer to stall the interface using the “app_ldm_data_valid signal”, this will result in stalling the local bus and thus lower the usable local bus bandwidth.

### 3.1.2 DMA Transfer Size

The raw DMA hardware is designed for only small transfers up to 4KB. This limitation is intentional! The small DMA size is imposed in order to limit the amount of time that a single DMA stream occupies the local bus. It is the job of the VDMA sequencer to fairly schedule a series of small DMA operations so that large transfers are possible.

As an example, consider 2 DMA streams in the L2P direction each having a FIFO receiving data at a continuous rate (where the combined rate is well within the limits of the system). There are at least 2 ways that the 2 FIFOs may be treated by the DMA controller.

1. For the first source channel, initiate the largest sized DMA possible to fulfil a large total transfer in one operation. Then, do the same for the second channel.

---

2. For data sources/sinks that are always “ready”, a FIFO may not be required.

2. Schedule a series of small sized DMA operations that interleave between the 2 source channels

These 2 options are illustrated in Figure 3-2.

**Figure 3-2: Large vs. Small DMA Size Interleaving**

The disadvantage with large/coarse interleaving is that larger buffers are required compared to interleaving more often.

The choice of what DMA transfer length to use for interleaving requires a trade-off and some understanding of the limitations of the host system and root complex:

- In favour of large DMA size: The longer the DMA transfer, the longer the local bus and PCIe packet size. The longer the packet size, the higher the bus utilization and potential throughput.
- Against large DMA size: The longer the DMA transfer, the larger the buffer size required.

The DMA size can be set by the user. However, as a practical matter, here are some recommendations:

- For L2P direction:
  - Almost all PCI Express root complexes in existence today use either a 128B or 256B MAX_PAYLOAD. Consequently, any 4KB DMA transfer will be sent as a series of 128B or 256B transfers. Therefore, there is little reason to use a transfer size of more than 256B from this perspective.
  - The overhead of the sequencer will favor a larger transfer size. DMA transfer sizes should be at least big enough to “hide” the time the sequencer takes to set up the next DMA. Sequencer instructions require 2 to 3 internal clock cycles to execute. When the DMA is operating, each internal clock cycle transfers 8 bytes of data (64 bit). Consequently, a 512 Byte transfer happens in 64 clocks which allows about 21 to 32 sequencer instructions to execute in that same time. This is typicaly enough time to keep issuing contiguous DMA instructions that will keep the L2P DMA always busy (and this is the best that can be done). For more complex DMA microcode, a transfer size of 1KB may be required to saturate the DMA.
- For P2L direction:
  - Most current PCI Express root complexes limit the read request size to either 512B or 1024B.
  - The GN412x and most root complexes allow multiple outstanding reads. Up to 3 outstanding reads are allowed by the GN412x. However, the application layer must be able to handle out-of-order reads if this feature is used. Supporting out-of-order reads will have a significant impact on throughput in the P2L DMA direction. Consequently, for applications that must obtain the best possible throughput in the P2L direction, 3-1KB buffers should be used as described in Section 3.3.2.

## 3.2 Source FIFO Design (L2P Direction)

Unless a data source can always produce data whenever called upon by the FlexDMA, a FIFO will be required to buffer up a packet worth of data.

There are several purposes for the FIFO:

- It allows the DMA core to operate without stalling the L2P bus. Once a DMA transfer is started it is committed to complete the block size specified. There is a data valid signal that can be used to stall the L2P transfer. However, it should be used only for very brief periods otherwise the throughput of the local bus will be limited as stalling will block any other L2P traffic (including the request phase of P2L DMA).
- It provides a convenient mechanism for asynchronous clocking between the application layer data source and the core clock of the Gennum IP

The source FIFO should provide a status signal that signals the VDMA sequencer when there is enough data available to do the maximum sized transfer that the DMA could request for the stream. This size could be up to the maximum size of a DMA command (4KB) or could be less than that value depending on how the VDMA sequencer is programmed.

## 3.3 Sink FIFO Design (P2L Direction)

In the case of P2L DMA direction, the FIFO status must indicate that there is enough room left in the FIFO to absorb the largest DMA request. If the status falsely indicates a DMA request, then there is the risk that the FIFO will overflow.

### 3.3.1 Support for Multiple Outstanding Reads

Achieving maximum throughput in the P2L direction is only possible by supporting the Multiple Outstanding Read feature of the PCI Express protocol and supported by the GN412x. The Multiple Outstanding Read feature allows a DMA device to initiate multiple read requests to the host. In the case of the GN412x and local bus IP, up to 3 outstanding reads can be initiated[1].

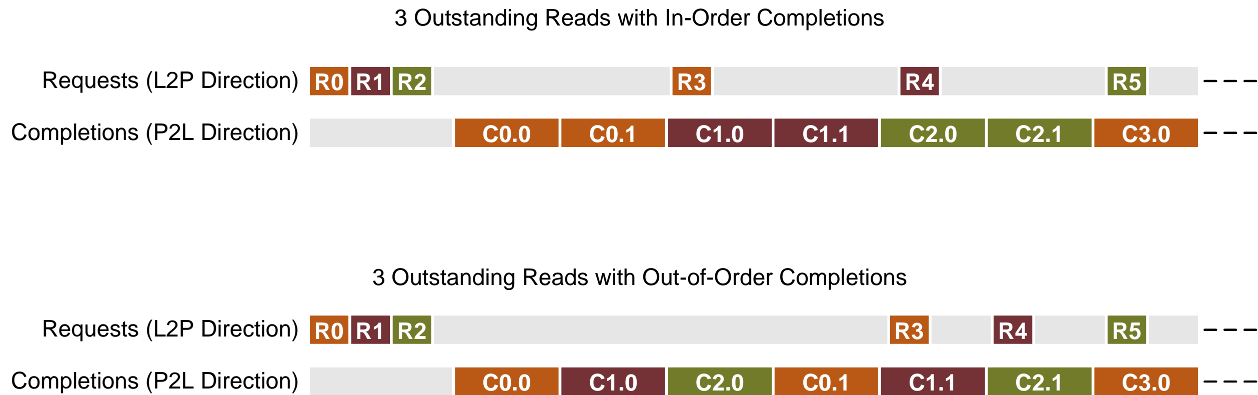
There are 2 ways that a root complex can handle read completions:

- Return each completion in-order: first-in-first-out
- Return completions out of order by interleaving completion packets

both of these completion scenarios are depicted below in Figure 3-3.

1. The GN412x supports up to 3 outstanding reads per virtual channel and per direction for a total of 12 outstanding reads. However, in the context of this document we are only considering the master mode DMA direction on a single virtual channel.

**Figure 3-3: In-order Compared to Out-of-Order Completions**




Most host root complexes use a cache line fetching mechanism. Consequently, they will return completion data using packets up to 64 Bytes and no larger. When a request larger that 64 Bytes is made, the completion will be a series of 64 Byte (or less) packets. Some root complexes implement "read completion combining" where multiple 64 Byte completions are opportunistically combined into larger completions. However, it is not practical to predict this behaviour as it is based on the host memory fetching that is taking place deep in the root complex. in other words, the packet sizes of DMA read completions may vary and could be anywhere from 1 DW up to the MAX_PAYLOAD size. The GN412x will pass through the completion packets in whatever size they are received on the PCI Express interface.

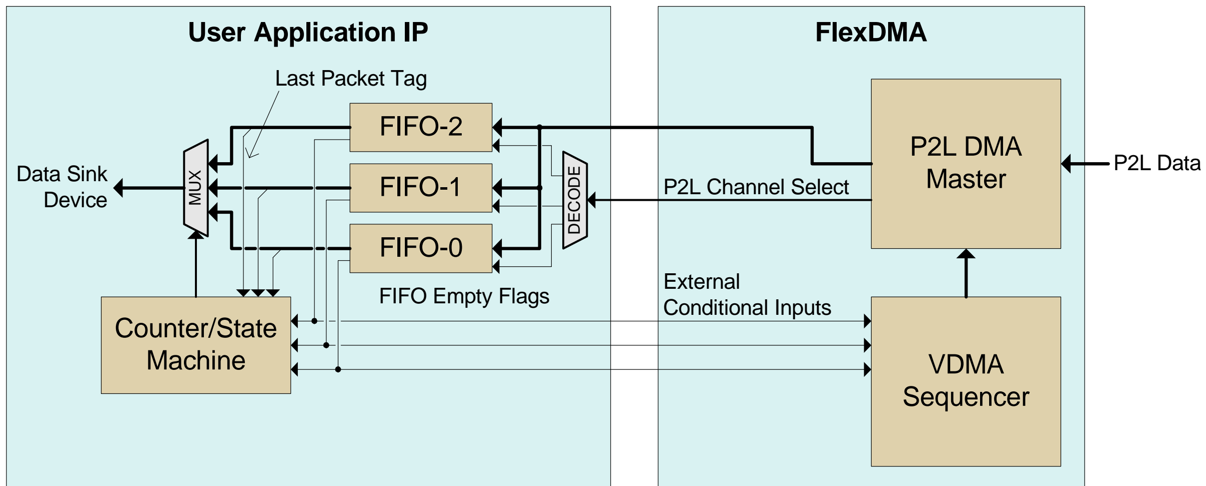
Referring again to Figure 3-3, each scenario begins when the DMA controller issues three sequential read requests (R0, R1, R2). For the purposes of illustration, suppose that each request was for 128 Bytes which are all aligned on a cache line boundary. this will typically result in 2 - 64 Byte completion packets per request (C0.0, C0.1 for R0 and so on).

The out-of-order scenario requires a reordering of completion packets when all of the completions are destined for the same DMA device, However, if each request is for a different DMA stream, then no reordering is necessary.

### 3.3.2 Implementation of Reordering Hardware

The GN412x local bus IP does not provide reordering circuitry. This is because not all applications require this feature and therefore it is desirable to have this accomplished in the application layer as needed rather than using up FPGA resources by default. The local bus IP does, however, provide the hooks to facilitate implementation of this feature in the application layer.

**Figure 3-4: Block Diagram of a Completion Reordering Mechanism**




A simple method of reordering is depicted in Figure 3-4. When the system is initialized, the FIFOs in the end-user application IP are all empty and the counter/state machine is selecting FIFO-0. Seeing that all three FIFO flags show empty, microcode in the VDMA sequencer will initiate three read DMA transfers starting with FIFO-0. Each of the read requests will have a unique stream ID as a destination in order to select one of the 3 FIFOs. As completion packets flow back through the P2L DMA master, they will get pushed into the appropriate FIFO. The counter/state machine will pop data out of FIFO-0 until the FIFO goes empty AND a “last packet” tag is detected. The last packet tag is pushed through the FIFO as an extra data bit that comes from the P2L DMA master. This is derived from the L bit of completion packets received over the GN412x local bus and indicates that it is the last packet in fulfilment of a read request.

Once FIFO-0 has been drained of all completion data from the first request, then the counter/state machine will move on to FIFO-1 and repeat this process indefinitely.

# 4. Multi-Channel DMA Software Design

There are three main aspects to designing software for multichannel DMA operation:

- The microcode that runs on the VDMA sequencer
- The host driver and application code
- The interaction between the host driver and the VDMA sequencer and FlexDMA

The FlexDMA controller is designed to be highly flexible to accommodate virtually any DMA scenario. Consequently, there are many ways that the software and hardware/software interaction may be designed. While it is not possible to describe all feasible use models, a typical use model will be considered. That is, 2 DMA channels in each direction where data on the host side is contained in scatter gather lists with a 4KB page size.

## 4.1 DMA Programming

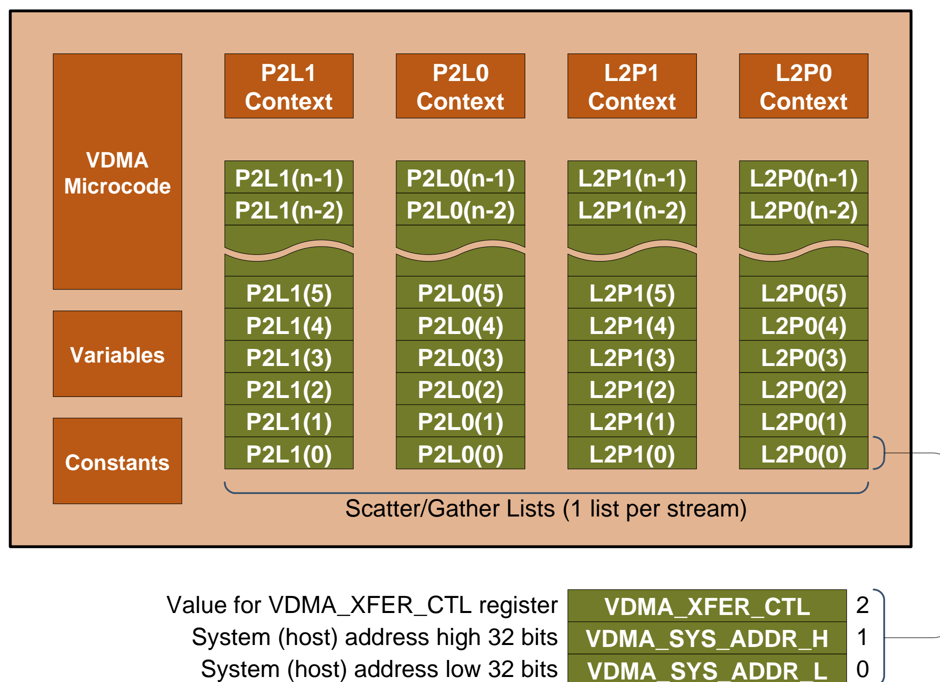
Programming of the DMA controller involves two main aspects:

- Setting up and maintaining data structures and control registers inside the FlexDMA hardware. This is typically done during initialization and also during DMA operation.
- Programming the VDMA sequencer through microcode

### 4.1.1 Data Structures for Scatter/Gather Operation

The first aspect of programming the FlexDMA hardware is to design an appropriate data structure. In general, a data structure will contain the elements depicted in Figure 4-1.

**Figure 4-1: Data Organization in the Descriptor RAM of the VDMA Sequencer**



In Figure 4-1, the main elements are:

- The VDMA microcode: this is where the VDMA sequencer instructions are stored in the descriptor RAM.
- Variables and constants: these are referred to by the VDMA microcode for general use as needed.
- Scatter/gather lists (one per DMA stream): these are lists of DMA operations to be performed. This area is data that is processed according to the VDMA microcode. This data will be initialized by the host, operated on by the VDMA sequencer during DMA operation, and then later reloaded by the host driver as each individual scatter gather entry is processed.
- Stream context: this is where state information is stored for each stream. This data will be initialized by the host and then maintained by the VDMA sequencer.

Each entry in the scatter gather list is a data structure such as that shown at the bottom of Figure 4-1. This is only one possible data structure that could be used.

#### 4.1.1.1 Extended SG Data Structure

It is also possible to extend the data structure of Figure 4-1. For example, it may be desirable to facilitate having one entry accommodate DMA sizes larger than 4KB. In this case a "repeat" value could be added to the data structure to indicate how many times to repeat the specified DMA transfer length. This is useful for cases where the data will not be highly fragmented.

**Figure 4-2: Scatter/Gather Data Structure Supporting Large Contiguous Blocks**

| Description | Field | Offset |
|---|---|---|
| Number of times to repeat transfer (only the low 16 bits used) | VDMA_XFER_RPT | 3 |
| Value for VDMA_XFER_CTL register | VDMA_XFER_CTL | 2 |
| System (host) address high 32 bits | VDMA_SYS_ADDR_H | 1 |
| System (host) address low 32 bits | VDMA_SYS_ADDR_L | 0 |

In order to support the extended data structure of Figure 4-2, the VDMA sequencer microcode will use the VDMA_XFER_RPT value to repeat the transfer specified by VDMA_XFER_CTL. For example, if VDMA_XFER_RPT specifies a repeat value of 2048, and VDMA_XFER_CTL specifies a transfer size of 4KB, then the total DMA transfer would be 8MB.

In the case of multi-stream DMA, a 4MB transfer should not be allowed to exclusively monopolize the local bus. Otherwise, other DMA streams will be starved. Instead, the DMA sequencer should do only a single 4KB transfer before checking to see if another stream requires servicing. If another stream requires servicing, then the context for the current DMA stream should be stored away and the context for another DMA stream loaded. The fairness and granularity of stream servicing is entirely controlled through VDMA sequencer microcode.

#### 4.1.1.2 Accommodating Large DMA Channel Count

When designing for a relatively few DMA channels, a 4KB transfer size is manageable in terms of required FPGA resources. However, as channel count increases 4KB per channel may be prohibitive because each channel will require a data FIFO of at least that

size. In this case the SG format of Figure 4-2 should be used even with a 4KB segmented SG list.

As an example consider the case where each 4KB entry will be played out as 32 DMA transfers of 128B each. In between each 128B transfer of a particular stream, other channels we'll have a chance to operate. In this case, an entry in the list will specify a 128 byte transfer size for the VDMA_XFER_CTL and a repeat count of 32 in the VDMA_XFER_RPT.

A transfers size of 128B per individual DMA operation is recommended as a minimum. This is because it is large enough to minimize packet overhead and yet small enough to avoid starving other FIFO channels and therefore use minimal memory resources in the FPGA. 128B also happens to be the typical max payload size for most systems. The 4-word list type (Figure 4-2) allows the DMA to do a 128B section of a 4K page for one stream, then switch to another stream to do a 128B section of a 4K entry in the scatter/gather list for that second stream. You can then go back and do additional 128B sections of those 4K pages until they are complete and then move on to the next 4K page entry.

#### 4.1.1.3 DMA End Cases

When the host side driver or application software allocates memory dynamically then the result is typically a 4KB segmented scatter/gather list. However, the list may not begin or end on even 4KB boundaries. Consequently, these end cases must be considered particularly when using the 4-word list type of Figure 4-2.

First of all, precautions should be taken to ensure that buffers are aligned to a 32 bit (DW) address. This may be done by allocating an additional DW of buffer space and then not using some of the first bytes as necessary to force DW alignment.

Another consideration is that end cases may prohibit the use of the repeat count method of Figure 4-2. This may happen when the size of the first or last host SG entry is not an even multiple of the size used for VDMA_XFER_CTL. In this case, the host driver should break a single host side SG entry into two VDMA side list entries. One of the list entries will have a VDMA_XFER_RPT value of 1 and the transfer length that will allow the other list entry to use a repeat value. The other list entry will have a VDMA_XFER_RPT value of 1 or more as required to complete the host-side entry using an even multiple of the length specified in VDMA_XFER_CTL.

While there are other methods of handling these end cases, this method has the advantage of not requiring additional VDMA programming complexity while optimizing precious descriptor memory resources.

### 4.1.2 VDMA Sequencer Programming

This section describes how the microcode that drives the FlexDMA hardware is written to accommodate multichannel DMA.

### 4.1.2.1 Channel Scheduling

The typical use model for the VDMA sequencer is to sit in a tight loop waiting for the chance to initiate a DMA operation. An example of this is depicted in Figure 4-3.

**Figure 4-3: Basic Top-level DMA Servicing Flowchart**

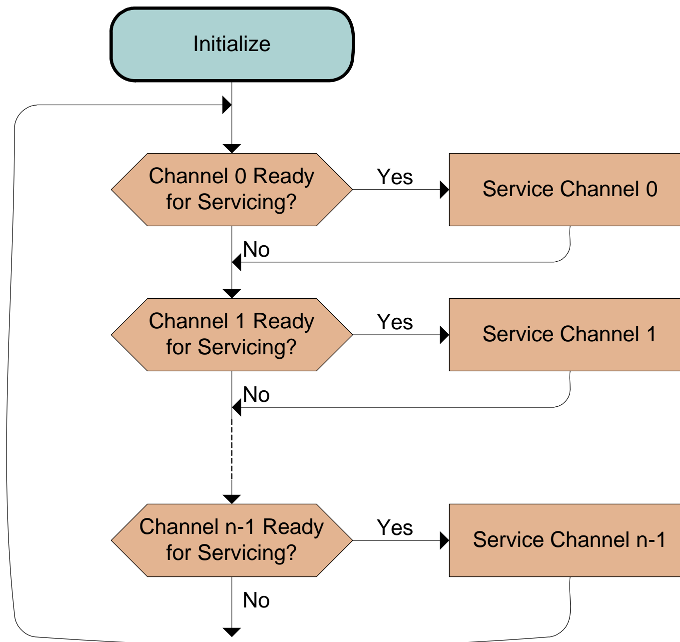



Figure 4-3 represents a simple scheduler that operates as a round robin arbiter. Other scheduling algorithms may also be employed. For example, a two level round robin arbiter that gives a higher priority to one set of channels and lower priority to lower bandwidth channels that are less likely to underflow/overflow.

In order to determine if a particular channel requires servicing, the VDMA sequencer can use the conditional jump instruction and specify one of the external conditions as the value to test. The external condition is generated by the hardware in the user application layer as described previously.

The hardware signal(s) coming from the user application layer to signal to the VDMA sequencer that a channel requires servicing (DMA request in other words) must be generated carefully so that stalling is avoided. In the L2P direction, a DMA request should only be asserted when there is enough data ready in the FIFO to fulfil the largest single transfer size that may be requested. When the DMA engine is started using the VDMA_LOAD_XFER_CTL instruction, it is committed to transferring the entire length of the requested transfer. While it is possible for the application layer to stall the DMA engine, this will have a direct impact on performance and will block other types of traffic on the local bus of the G412x.

### 4.1.2.2 DMA Channel Servicing

Each DMA stream will require a block of microcode to instruct the sequencer how to service the stream when servicing is required.

One of the key properties of a channel servicing subprogram is that it should never stall or monopolize the sequencer for any sizable period of time. Otherwise, the other channels will go un-serviced with the risk of overflowing/under flowing their respective FIFOs. In order to guarantee this several issues need to be considered:

- Never busy wait. That is, do not sit in a loop waiting for something to be ready. if some condition needed to allow servicing of a channel is not ready, then move on to testing the next channel.
- Do only a single atomic DMA operation on each channel rather than a long sequence of DMA operations on a single channel before checking that another channel requires servicing. This is important when the data structure of Figure 4-2is used.

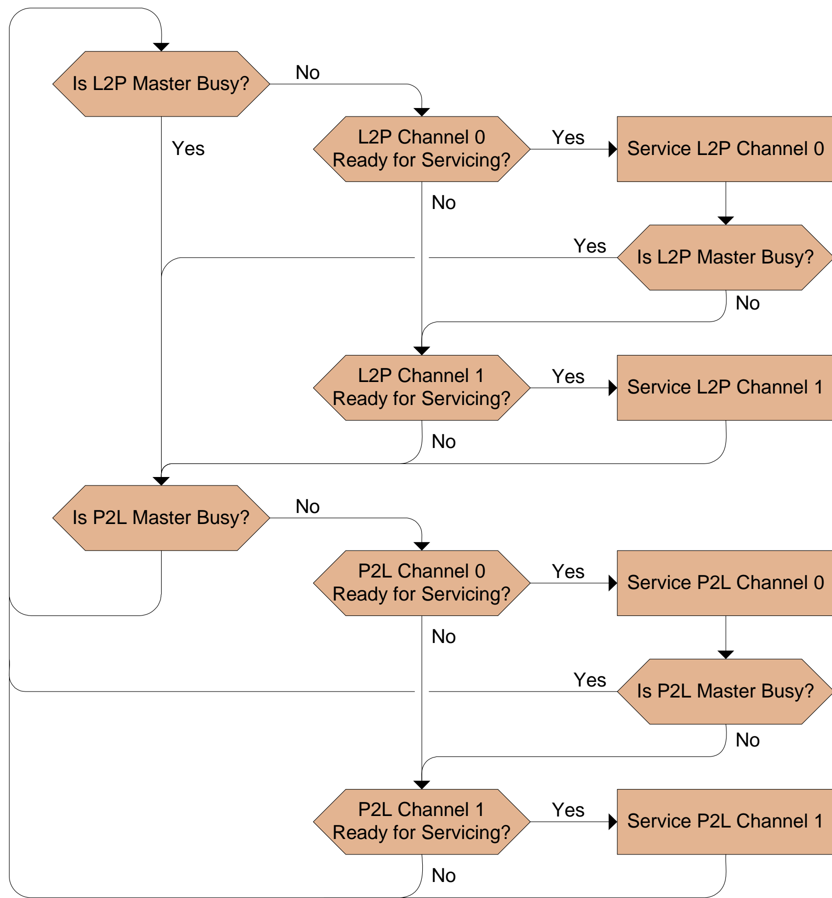
There are only two reasons that the sequencer should be looping around without DMA transfers being initiated. That is:

- No FIFO requires servicing
- A FIFO requires servicing but the DMA engine it requires is busy

One of the hazards to avoid is executing a LOAD_XFER_CTL when the specified DMA engine cannot accept additional commands. Each DMA engine is double buffered so that it may be executing a command and then have one additional command queued up and ready to go. However, when the DMA engine is in this state, the execution of another LOAD_XFER_CTL will stall the sequencer. In order to avoid this, the JMP instruction can be used to check the status of the DMA before trying to start the DMA.

For example, if the L2P DMA master is busy and cannot accept additional commands, then the sequencer program should attempt to service P2L channels.

Figure 4-4: DMA Servicing That Avoids DMA Master Busy Stalling



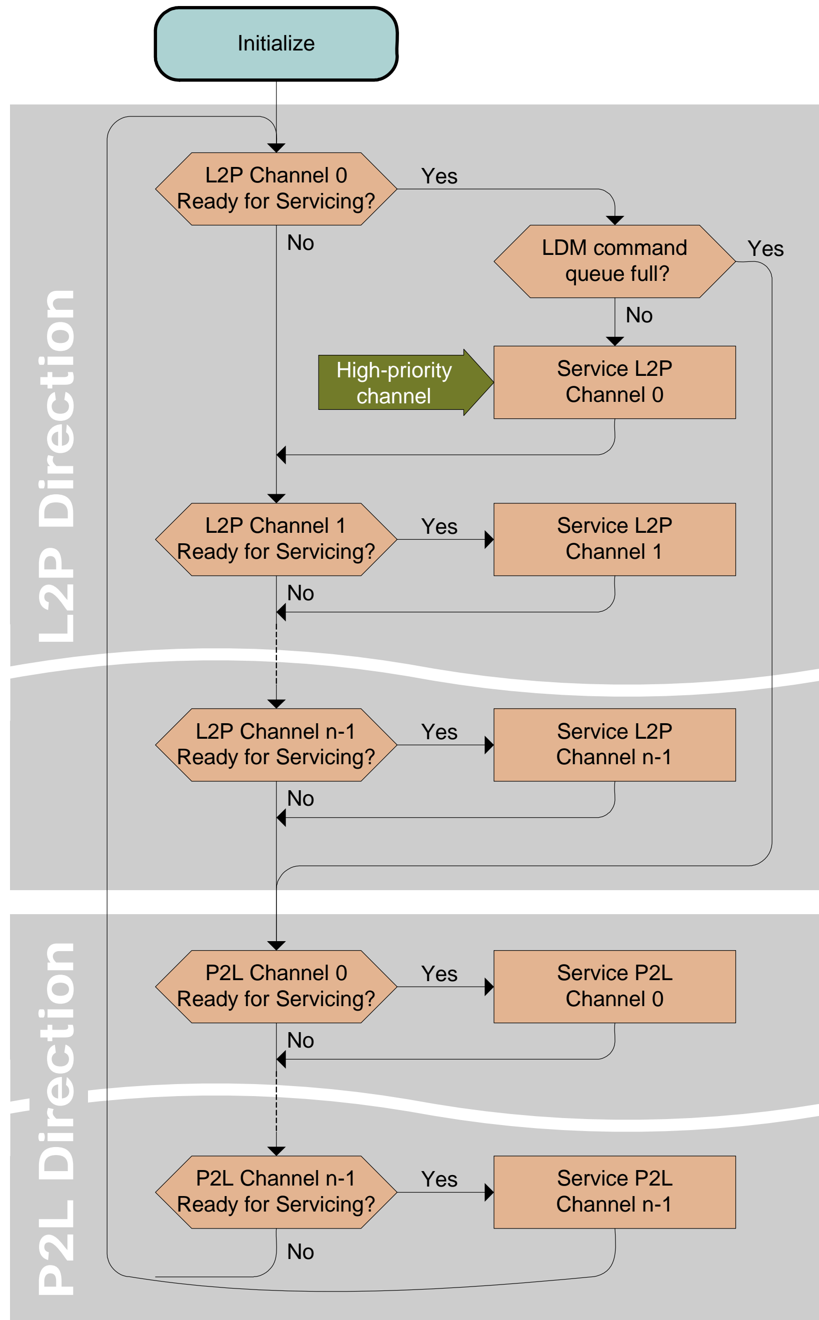

The basic steps of channel servicing are:

- Load the context for the channel: at a minimum this would include a pointer to the scatter gather data structure for that channel.
- Load and execute the current scatter gather entry: this will initiate a DMA transfer.
- Tag the scatter gather list entry as “done”
- Update any pointers/counters and store away the channel context

### 4.1.2.3 High-Priority Channels

There may be cases where there are high-priority channels that require it the lowest possible latency. Usually high priority channels are for time critical but relatively small amounts of data. An example of this would be time-code synchronization packets that need to be forwarded to the host as quickly as possible. For channels like this, the vdma_jmp instruction may be used to detect a high priority channel request and ensure that it gets the next available L2P DMA. This is depicted in Figure 4-5.

**Figure 4-5: Channel Servicing for a High-Priority DMA Channel**




In Figure 4-5, L2P channel 0 is the low latency high-priority channel. For this scenario, all other L2P traffic is postponed until L2P Channel 0 has been serviced.

## 4.2 Hardware/Software Interaction

This section describes some of the typical techniques for hardware software interaction.

Given the flexibility of the FlexDMA IP block, many hardware/software interaction models are possible. In general, the criteria of hardware/software interaction is to minimize the overhead of host side CPU utilization and guarantee quality of service. In order for this to be realized, it is important that the endpoint device employing the GN412x be able to operate for relatively long periods of time without servicing by the host processor.

## 4.2.1 Semaphore Mechanisms

The VDMA sequencer can be thought of as a simple I/O processor. consequently, it is necessary for the host processor to coordinate its activities with the state of the VDMA sequencer. This may be accomplished through several semaphore mechanisms:

- Interrupts using the “EVENT” mechanism
- Host polling of endpoint registers either in the VDMA block or user application hardware
- Host polling of host memory-based semaphores

### 4.2.1.1 EVENT Interrupts

The event register inside the VDMA sequencer is designed to enable activity in the sequencer to generate interrupts to the host system. For an interrupt to be generated, several things must happen:

- The Main PCI express interrupt generation must be enabled on the GN412x. This will be done through either the MSI or INTx interrupt mechanism. See the “GN412x RDK Software Design Guide”.
- The hardware interrupt coming from the FPGA to the GN412x via GPIO requires that the GPIO and interrupt registers inside the GN412x be properly configured.
- The appropriate VDMA_EVENT_EN bits inside the VDMA sequencer need to be set in order to allow VDMA_EVENT bits to be routed to the interrupt controller

Once VDMA_EVENT interrupt generation is enabled, the following VDMA sequencer microcode will assert an interrupt to the host:

```
VDMA_SIG_EVENT(0, 1, 0x0008)
```

Here, VDMA_EVENT bit 3 will be asserted. In its interrupt service routine, the host may then clear the interrupt by writing to the VDMA_EVENT_CLR register has in:

```
write(VDMA_EVENT_CLR, 0x0008);
```

Having one side (VDMA sequencer) assert the event, and having the other side (host driver) clear the event is a typical use model for the event register.

### 4.2.1.2 Host Polling of the Endpoint

As an alternative to interrupts, the host may periodically poll the endpoint device to determine if servicing is required. The VDMA_EVENT register may also be used for this purpose only without interrupts being enabled.

### 4.2.1.3 Host Memory Semaphores

Rather than the host polling the endpoint device directly, the VDMA sequencer can use a small DMA operation to clear a semaphore that is located in host memory. That way, the host processor may read locally attached DDR memory so that the latency cost of polling is significantly reduced.

In this use model, a host memory DW sized location is first initialized to a nonzero value by the host driver. In the VDMA sequencer, a DMA of size 1DW in the is placed in the DMA list for a particular channel. This would be, for example, at the end of a series of DMA operations completing a video frame. Since data for all L2P DMA operations comes from the user application hardware, the source of data for the single DW transfer is a stream ID that will produce zero on the data for the L2P direction (see Figure 4-6).

**Figure 4-6: Hardware Setup to Support Semaphore DMA**



The scheduling of semaphore transfers can be accomplished within the existing scatter/gather list structure defined in Figure 4-1. However, some of the scatter gather list entries will be semaphore DMA transfers. This is illustrated in the example of Figure 4-7.

**Figure 4-7: Scatter/Gather List With Semaphore Entries**

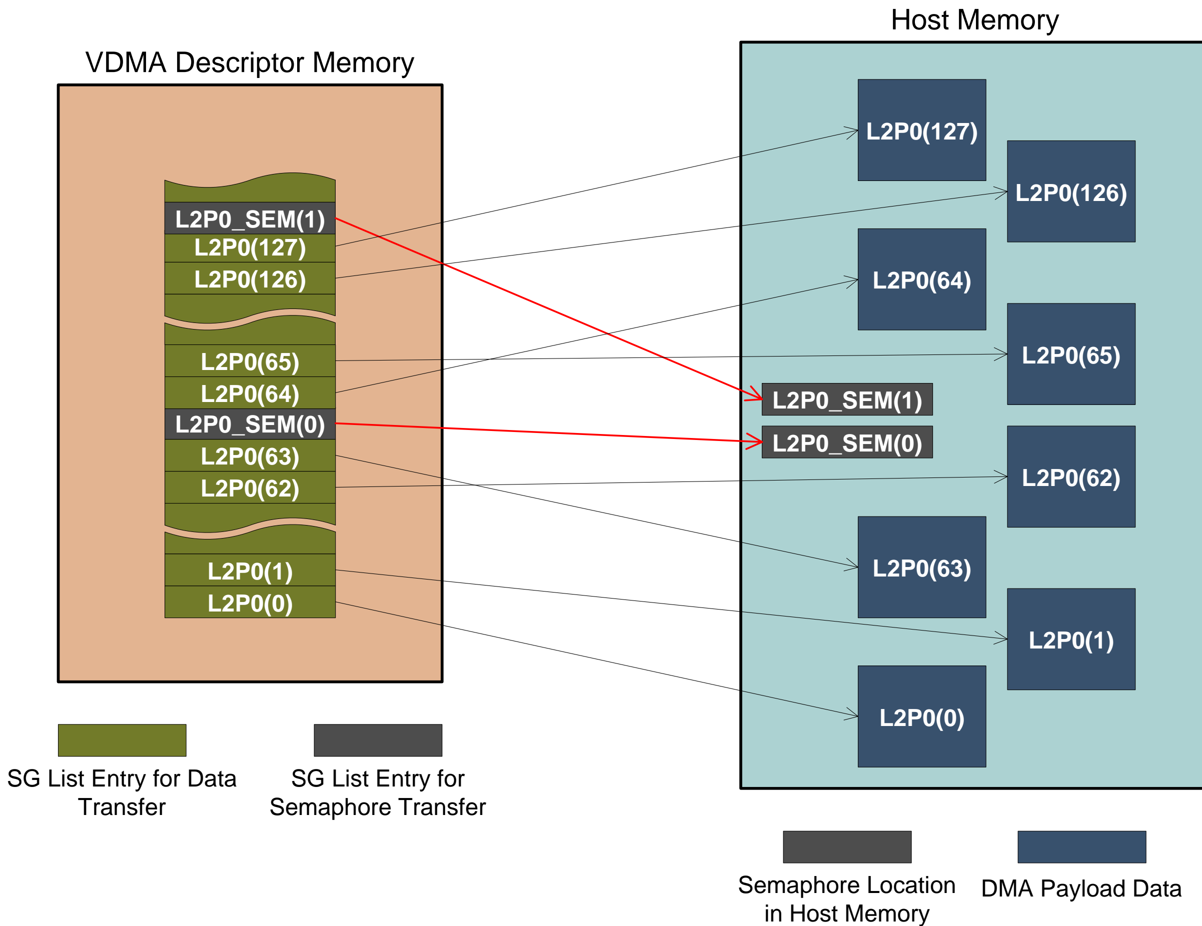



Figure 4-7 shows a semaphore transfer happening after every 64 SG list entries. The frequency at which semaphore transfers should occur is application dependent. In general, semaphores should be inserted frequently enough for the host processor to be able to service the application adequately.

Semaphore DMA transfers may also be used to signal specific milestones. For example, when a frame of video has been completed.

This concept may be expanded so that, instead of transferring only a single DW of data, a small block of data containing useful status information may be transferred instead. That way, the host has even fewer needs to interact directly with the endpoint hardware.

When using semaphores in this way, it is important that the host not be polling from cache memory where the data may be out of date. Consequently the semaphore should be located in a non-cache region or precautions should be taken so that stale data is updated.

## 4.3 Host Software

The main functions of the host software are:

- Setup and initialize the GN412x based endpoint device
  - enumerate (if no BIOS or OS enumeration supported) and set up the GN412x
  - download the FPGA IP
  - Initialize the GN412x FPGA IP core
  - Initialize the user application hardware (UAL)
- Manage DMA operation
  - allocate memory for DMA transfer in the host address space
  - Write S/G lists into the VDMA descriptor ram
- Detect exception conditions and process
  - Stream set up and tear down

### 4.3.1 Scatter/Gather List Management

There are two main scenarios for DMA management:

1. Fixed DMA list: host side memory buffer(s) are repeatedly reused
2. Dynamic DMA list: host side memory buffer(s) are being dynamically allocated

#### 4.3.1.1 Fixed DMA List

In the case of a fixed DMA list, the list can be set up initially so that it loops infinitely or until shut down by a state change in the application software. In this case, there is no need for the host to dynamically update the list. An example of this use model would be video capture where there is a fixed frame buffer (or perhaps multiple frame buffers). Alternately, the buffers may be organized as a contiguous block of physical memory rather than being scattered around host address space.

When a DMA list can be set up in the VDMA sequencer and remain fixed, the host/endpoint interaction is greatly simplified. As far as DMA operation is concerned, the main host/endpoint interaction will be to maintain synchronization between the host application and the hardware data transfer. This could be, for example, indicating the end of the frame of video. This synchronization may be communicated through one of the semaphore mechanisms outlined in Section 4.2.1 on page 19.

Even if a DMA list is fixed, it may be too large to fit into the descriptor memory of the VDMA sequencer. If this is the case, then it will be necessary to treat it more like the case of a dynamic DMA list where the host must periodically update the portion of the list that resides in the descriptor memory.

#### 4.3.1.2 Dynamic DMA List

In a typical use model of a dynamic DMA list, each DMA stream will have a list as depicted in Figure 4-1. Also, the list for each stream is managed as a circular list. As the scatter gather list for a particular stream is processed then list entries will be “retired”, that is, marked as “complete” so that it will not execute again until updated by the host. This is accomplished through descriptor write back where the VDMA_XFER_CTL value is written with zero.

When the VDMA_XFER_CTL value is zero, no hardware DMA operation will be performed. Consequently, a circular loop of scatter/gather entries will only get executed once when descriptor write back method is used.

The typical microcode for descriptor write back is:

```
//RB is the list pointer
VDMA_LOAD_SYS_ADDR(VDMA_RB,0)       //Load system address
VDMA_LOAD_XFER_CTL(VDMA_RB,2)       //Start DMA
VDMA_LOAD_SYS_ADDR(0,ZERO)          //Load zero
VDMA_STORE_SYS_ADDR(VDMA_RB,1)      //Clear the VDMA_XFER_CTL in list
```

In this microcode, both the VDMA_XFER_CTL list value and the VDMA_SYS_ADDR_H value is overwritten with zero. That is because VDMA_STORE_SYS_ADDR is a 64-bit data type and there is no 32-bit equivalent.

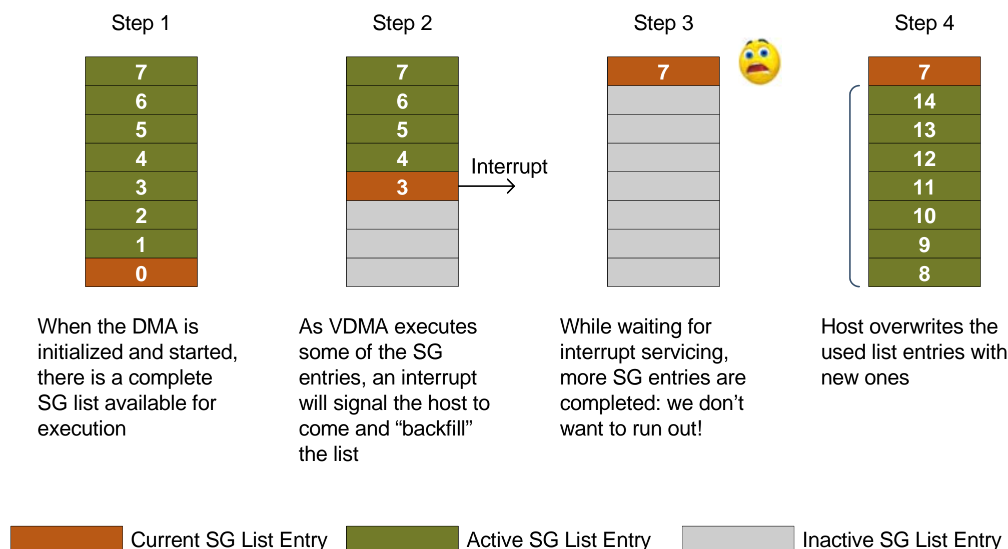
Initially, a list for a particular stream will contain all nonzero values for VDMA_XFER_CTL and then as the list is processed they will sequentially be zeroed out. It is then the job of the host side driver to replenish/reuse zeroed out entries before the entire list becomes zero. If all list entries become zero, then DMA transfers for that stream will cease. In the case of audio or video, this will result in unacceptable temporary loss of sound and picture.

The relationship between the size of the scatter/gather list and servicing latency must be characterized in order to guarantee that no data will be lost in the case of isochronous data sources/sinks. In its default configuration, the FlexDMA IP core has a total of 2K words of descriptor RAM[1]. Part of that RAM is used for microcode and other types of data storage. Consequently, there will be room for approximately 600 list entries total when using the 3-word list type of Figure 4-1. If each list entry is for a 4KB DMA size, then up to 2.4MB of transfers to be queued up for unattended operation.

Take as an example a 1080p60 un-compressed video stream at 250MB/s. For this, 2.4MB of DMA setup will execute in approximately 9.6ms. in other words, the host driver must update the descriptor memory within 9.6ms of the last time the descriptors were completely updated. If descriptor memory must service two of the 1080p60 streams, then the servicing latency will be half that value (4.8ms) since each stream will have to share the available descriptor memory.

1. The “Pinto project has 4K words (16KB) of descriptor memory.

**Figure 4-8: Process of Dynamic SG List Management**




Studies show that interrupt latency can vary dramatically depending on the hardware software environment of the host computer[1]. In a carefully controlled embedded system employing a real-time kernel, interrupt latency can be guaranteed well within a couple of hundred microseconds. However, endpoint devices that may be plugged into any arbitrary PC using non-real-time operating systems may experience significantly longer worst-case interrupt latencies. For example, a Windows PC running background tasks, such as virus scanning, can result in interrupt latencies in excess of 20ms even though the average latency may be several orders of magnitude less.

In cases of non-isochronous (i.e. "best effort") dataflow, the occasional 20ms interrupt servicing latency will temporarily halt DMA data transfer with no significant side effects. However, when data is isochronous, such as real-time video or audio, then the worst case interrupt latency must be accommodated. Methods for mitigating long servicing latency are:

- Use the 4-word descriptor format of Figure 4-2. This will allow each list entry to specify more than 4KB of total transfer size. This will require the use of buffers that occupy contiguous physical address space.
- Increase the size of the descriptor RAM in the Gennum FPGA IP To be able to accommodate larger lists.

1. See "Optimizing PCIe® Performance in PCs & Embedded Systems" available at gennum.com.

# 5. DMA Coding Examples and Best Practices

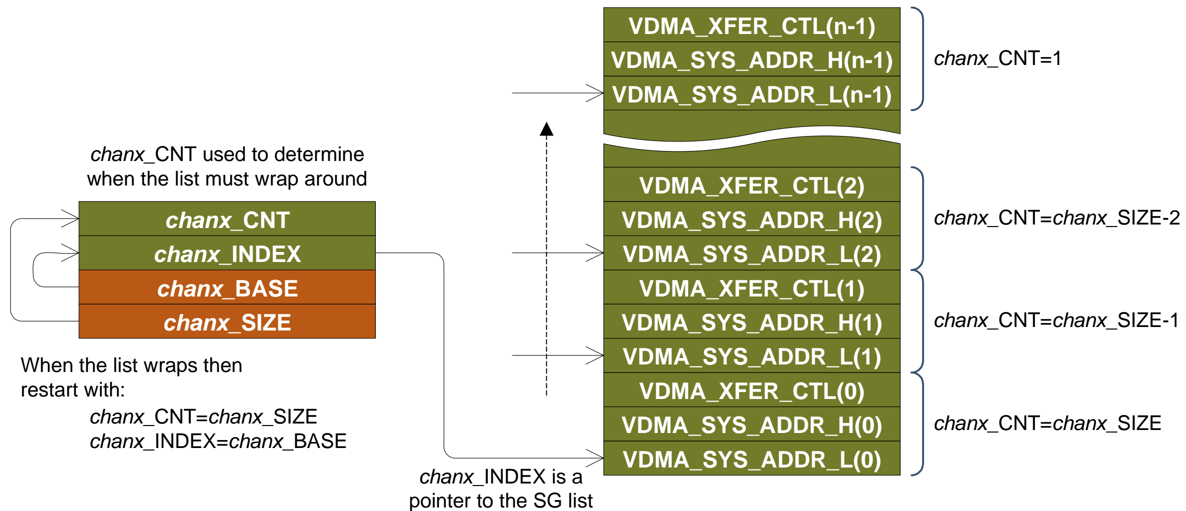
In order to better illustrate DMA programming, several examples will be described. These examples have been coded and simulated to provide a reliable framework for VDMA programming.

## 5.1 3-DW Descriptor Processing

This section describes the use of a 3-word SG list. For this list type, each list entry is exactly 3-32 bit words (3-DW) in length and contains the list structure of Figure 4-1. It contains a 64-bit system address in the first two locations and a transfer control word in the third location. The 3-DW descriptor type does not support the repeat count feature. That will be described in the next section.

Figure 5-1 shows the data organization in descriptor RAM in order to support 3-DW descriptor processing.

**Figure 5-1: Data Structure for 3-DW Descriptor Processing**

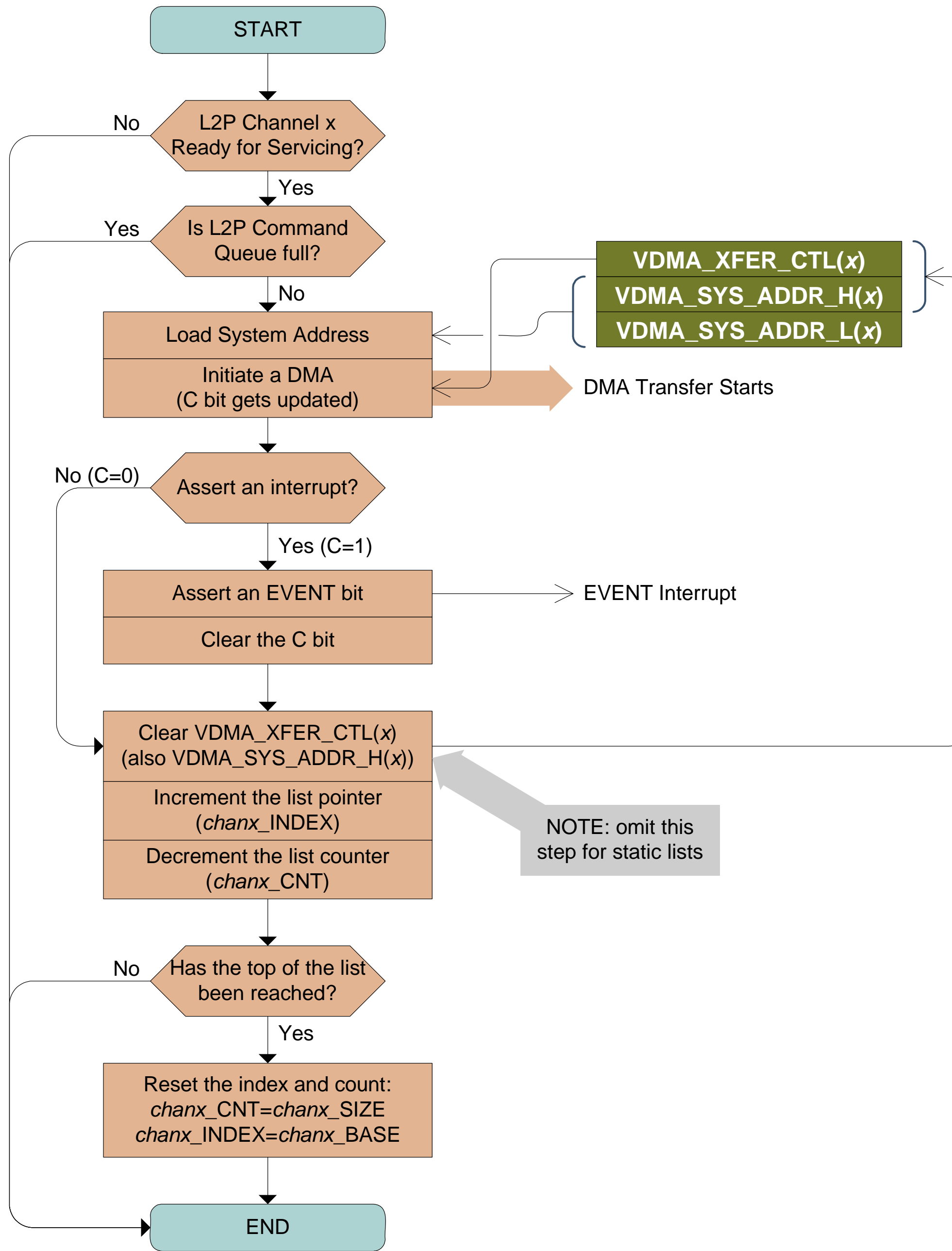



In addition to the scatter gather list itself there are the following locations used in the descriptor memory:

| | |
|---|---|
| *chanx*_CNT | List counter used to determine when the top of the list has been reached. The value of *chanx*_CNT is dynamically updated. |
| *chanx*_INDEX | List pointer used as an index into the list array. The value of *chanx*_INDEX is dynamically updated. |
| *chanx*_BASE | Contains the address of the base of the SG list for *chanx*. The value of *chanx*_BASE is set during initialization and then remains constant. |
| *chanx*_SIZE | Contains the size, in number of entries, of the SG list for *chanx*. The value of *chanx*_SIZE is set during initialization and then remains constant. |

The flowchart for the channel servicing microcode is depicted in Figure 5-2:

**Figure 5-2: 3-DW Descriptor Processing Microcode Flowchart**

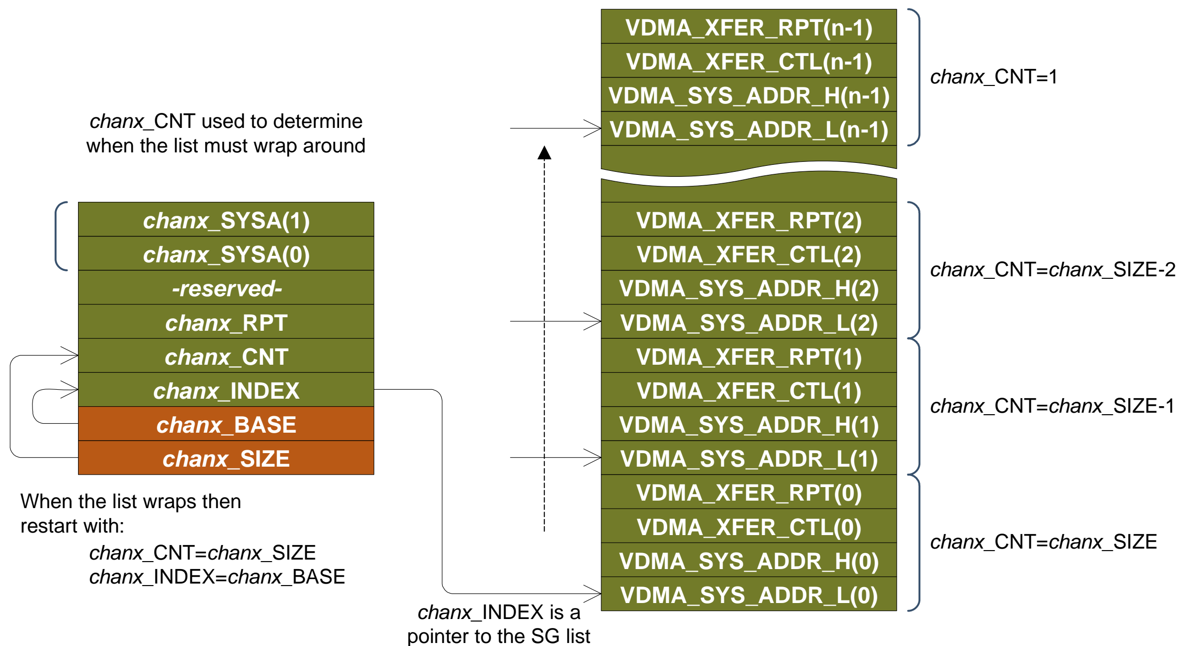


## 5.2 4-DW Descriptor Processing

The 4-DW list type is similar to the 3-DW type with the addition of a repeat count. With a repeat count, the same DMA can be run up to 64K times over a contiguous block (the repeat count is a 16 bit value). The data structure for this list type is depicted in Figure 4-2.

Figure 5-3 shows the data organization in descriptor RAM used to support 4-DW descriptor processing.

**Figure 5-3: Data Structure for 4-DW Descriptor Processing**



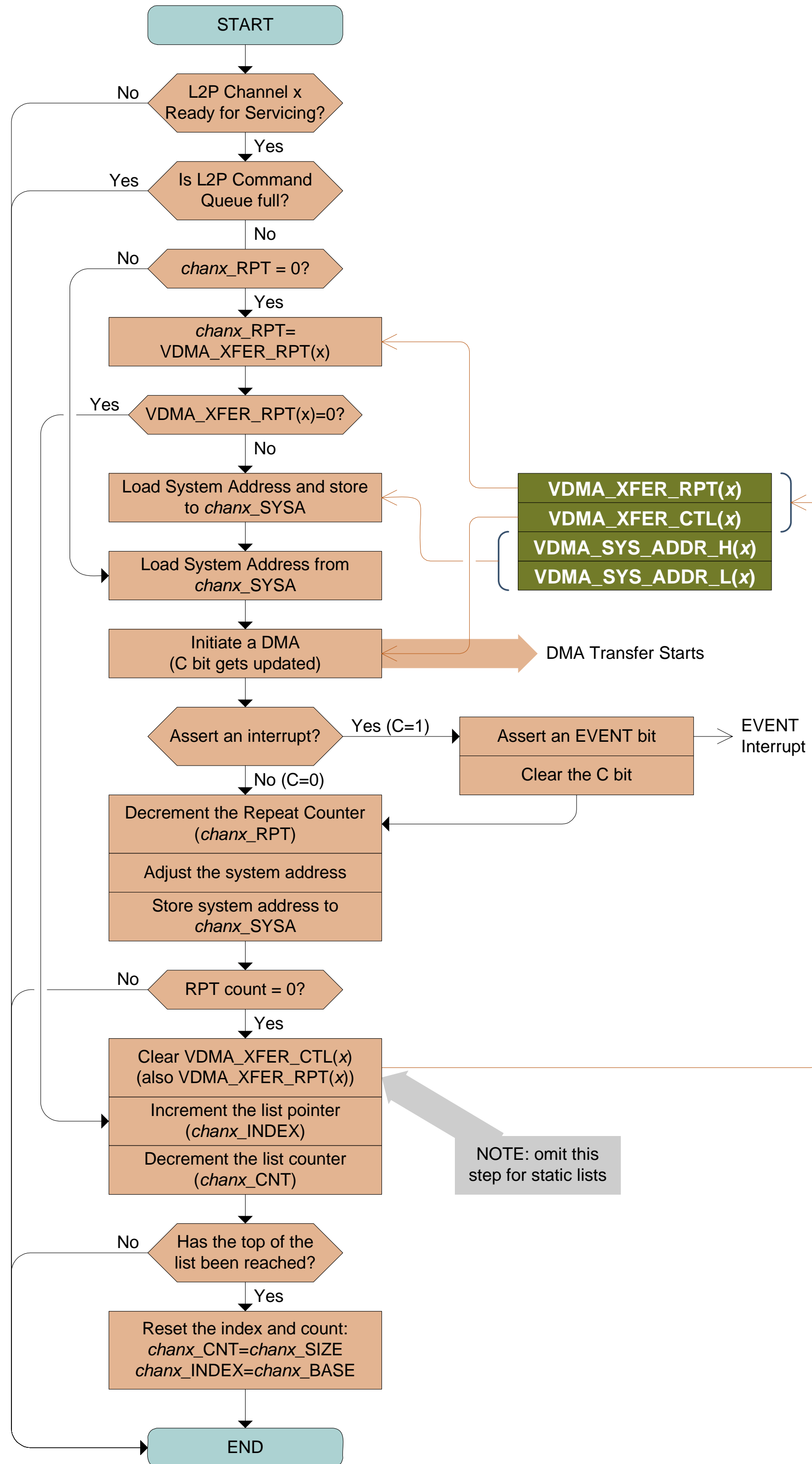

In addition to the scatter gather list itself there are the following locations used in the descriptor memory:

| | |
|---|---|
| *chanx*_SYSA | 64 bit system address. The value of *chanx*_SYSA is updated each time the DMA repeats so that a series of repeated DMA will reference a contiguous block of memory. |
| *chanx*_RPT | Repeat counter storage. The value of *chanx*_RPT is dynamically updated. Note that there is a reserved word next to *chanx*_RPT. This is because the only indexed load operation is vdma_load_sys_addr which is always 64-bit. This instruction, together with vdma_store_sys_addr are used to initialize *chanx*_RPT. |
| *chanx*_CNT | List counter used to determine when the top of the list has been reached. The value of *chanx*_CNT is dynamically updated. |
| *chanx*_INDEX | List pointer used as an index into the list array. The value of *chanx*_INDEX is dynamically updated. |

| | |
|---|---|
| *chanx*_BASE | Contains the address of the base of the SG list for *chanx*. The value of *chanx*_BASE is set during initialization and then remains constant. |
| *chanx*_SIZE | Contains the size, in number of entries, of the SG list for *chanx*. The value of *chanx*_SIZE is set during initialization and then remains constant. |

The flowchart for the channel servicing microcode is depicted in Figure 5-4:

**Figure 5-4: 4-DW Descriptor Processing Microcode Flowchart**

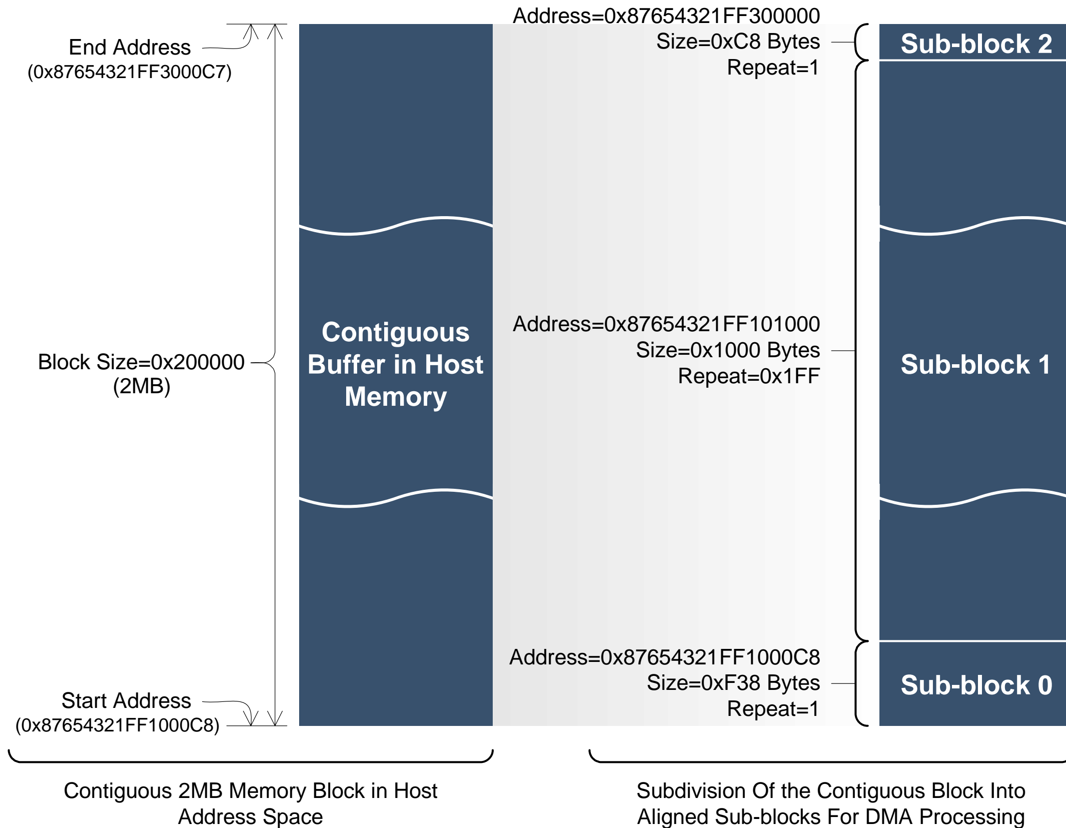


The 4DW descriptor format is the more flexible compared to the 3DW format as it can handle both fragmented SG lists in addition to large contiguous memory blocks. Consequently, this format is recommended for most applications.

### 5.2.1 Example: Large Contiguous Host Buffer

In the case of a large contiguous host buffer, the 4DW list type is ideal. Consider the example of Figure 5-5. For this example, assume that the desired DMA transfer size for each individual DMA transfer is to be 4KB (the maximum allowed for a single VDMA_LOAD_XFER_CTL instruction).

**Figure 5-5: Mapping of a Contiguous 2MB Memory Block to 4DW Descriptor List**




In the example of Figure 5-5, a contiguous host memory buffer of 2MB is shown on the left. However, the buffer is not aligned to an even 4KB address. Consequently, the block must be broken into three sub blocks as shown on the right-hand side of the diagram. The first and last sub blocks will have a repeat count of 1 so that they can properly deal with the beginning and end cases. The intermediate sub-block (Sub-block 1) begins and ends on even 4KB boundaries and is therefore an exact multiple of 4KB. This being the case, Sub-block 1 requires only a single 4DW descriptor to process such a large transfer size.

When Figure 5-5 is converted to a descriptor list, the result is shown in Figure 5-6:

**Figure 5-6: Descriptor List for the Example 2MB Contiguous Memory Buffer**

| Sub-block | Value | Entry |
|---|---|---|
| Sub-block 2 | 0x00000001 | VDMA_XFER_RPT(2) |
| | 0x800100C8 | VDMA_XFER_CTL(2) |
| | 0x87654321 | VDMA_SYS_ADDR_H(2) |
| | 0xFF300000 | VDMA_SYS_ADDR_L(2) |
| Sub-block 1 | 0x000001FF | VDMA_XFER_RPT(1) |
| | 0x00010000 | VDMA_XFER_CTL(1) |
| | 0x87654321 | VDMA_SYS_ADDR_H(1) |
| | 0xFF101000 | VDMA_SYS_ADDR_L(1) |
| Sub-block 0 | 0x00000001 | VDMA_XFER_RPT(0) |
| | 0x00010F38 | VDMA_XFER_CTL(0) |
| | 0x87654321 | VDMA_SYS_ADDR_H(0) |
| | 0xFF1000C8 | VDMA_SYS_ADDR_L(0) |

For the example of Figure 5-6, the stream ID, embedded in the VDMA_XFER_CTL(x) value is set to 0x01. The layout for the VDMA_XFER_CTL entry is an image of the VDMA_XFER_CTL register inside the VDMA sequencer:

**VDMA_XFER_CTL**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C | | | | | | | | STREAM_ID(7:0) | | | | | | | | | | D | | CNT(11:0) | | | | | | | | | | | |

Bit 31 (the "C" bit) is set for Sub-block 2. This is the mechanism for indicating an interrupt back to the host.

For Sub-block 1, the CNT value is set to 0x000 which indicates a transfer length of 4096 Bytes. Also for Sub-block 1, the repeat count is set for 0x1FF so that a transfer length of 4KB will be repeated 511 times.

**Figure 5-7: Listing for Descriptor Microcode and Data for Example 2MB Contiguous Memory Buffer**

```
0x0000 0x00000000                    START: vdma_nop()
0x0001 0x17220021             L2P_CHAN0: vdma_jmp(c=_EXT_COND_LO, ext_cond=_EXT_COND_0, "L2P_CHAN0_END")
0x0002 0x1D000021                       vdma_jmp(c=_LDM_CMD_QUEUE_FULL_HI, ext_cond=NA, "L2P_CHAN0_END")
0x0003 0x24000202                          vdma_load_rb("L2P_CHAN0_INDEX")
0x0004 0x20000204                          vdma_load_ra("L2P_CHAN0_RPT")
0x0005 0x1000000C                          vdma_jmp(c=_RA_NEQZ, ext_cond=NA, "L2P_CHAN0_RPT_NOT_DONE")
0x0006 0x43000003                          vdma_load_sys_addr(r=_RB, "0x3")
0x0007 0x50000204                          vdma_store_sys_addr(r=_IM, "L2P_CHAN0_RPT")
0x0008 0x20000204                          vdma_load_ra("L2P_CHAN0_RPT")
0x0009 0x18000019                          vdma_jmp(c=_RA_EQZ, ext_cond=NA, "L2P_CHAN0_NEXT_DESC")
0x000A 0x43000000                          vdma_load_sys_addr(r=_RB, "0x0")
0x000B 0x50000206                          vdma_store_sys_addr(r=_IM, "L2P_CHAN0_SYSA")
0x000C 0x21000102  L2P_CHAN0_RPT_NOT_DONE: vdma_add_ra("MINUS1")
0x000D 0x40000206                          vdma_load_sys_addr(r=_IM, "L2P_CHAN0_SYSA")
0x000E 0xF3000002                          vdma_load_xfer_ctl(_RB, "0x2")
0x000F 0x13000013                          vdma_jmp(c=_C_LO, ext_cond=NA, "L2P_CHAN0_NO_INT")
0x0010 0x10000013                          vdma_jmp(c=_RA_NEQZ, ext_cond=NA, "L2P_CHAN0_NO_INT")
0x0011 0x84000001                          vdma_sig_event(s=0, a=1, event_en=0x0001)
0x0012 0xF0000100                          vdma_load_xfer_ctl(_IM, "ZERO")
0x0013 0xA2000204        L2P_CHAN0_NO_INT: vdma_store_ra("L2P_CHAN0_RPT")
0x0014 0x60001000                          vdma_add_sys_addr(4096)
```

```
0x0015 0x50000206                          vdma_store_sys_addr(r=_IM, "L2P_CHAN0_SYSA")
0x0016 0x10000020                          vdma_jmp(c=_RA_NEQZ, ext_cond=NA, "L2P_CHAN0_UPDATE_B")
0x0017 0x40000100                          vdma_load_sys_addr(r=_IM, "ZERO")
0x0018 0x53000002                          vdma_store_sys_addr(r=_RB, "0x2")
0x0019 0x25000104     L2P_CHAN0_NEXT_DESC: vdma_add_rb("FOUR")
0x001A 0x20000203                          vdma_load_ra("L2P_CHAN0_CNT")
0x001B 0x21000102                          vdma_add_ra("MINUS1")
0x001C 0x1000001F                          vdma_jmp(c=_RA_NEQZ, ext_cond=NA, "L2P_CHAN0_UPDATE")
0x001D 0x20000200                          vdma_load_ra("L2P_CHAN0_SIZE")
0x001E 0x24000201                          vdma_load_rb("L2P_CHAN0_BASE")
0x001F 0xA2000203        L2P_CHAN0_UPDATE: vdma_store_ra("L2P_CHAN0_CNT")
0x0020 0xA3000202      L2P_CHAN0_UPDATE_B: vdma_store_rb("L2P_CHAN0_INDEX")
0x0021 0x00000000                          vdma_nop()
0x0022 0x1A000001                          vdma_jmp(c=_ALWAYS, ext_cond=NA, "MAIN")
0x0100 0x00000000                    ZERO: vdma_constant_n64(0x0000000000000000)
0x0101 0x00000000                          // vdma_constant_n64 - upper data
0x0102 0xFFFFFFFF                  MINUS1: vdma_constant_n(0xFFFFFFFF)
0x0103 0x00000003                   THREE: vdma_constant_n(0x00000003)
0x0104 0x00000004                    FOUR: vdma_constant_n(0x00000004)
0x0200 0x00000003          L2P_CHAN0_SIZE: vdma_constant_n(0x00000003)
0x0201 0x00000208          L2P_CHAN0_BASE: vdma_constant_l("L2P_CHAN0_LIST")
0x0202 0x00000208         L2P_CHAN0_INDEX: vdma_constant_l("L2P_CHAN0_LIST")
0x0203 0x00000003           L2P_CHAN0_CNT: vdma_constant_n(0x00000003)
0x0204 0x00000000           L2P_CHAN0_RPT: vdma_constant_n(0x00000000)
0x0205 0x00000000                          vdma_constant_n(0x00000000)
0x0206 0x00000000          L2P_CHAN0_SYSA: vdma_constant_n64(0x0000000000000000)
0x0207 0x00000000                          // vdma_constant_n64 - upper data
0x0208 0xFF1000C8          L2P_CHAN0_LIST: vdma_constant_n64(0x87654321FF1000C8)
0x0209 0x87654321                          // vdma_constant_n64 - upper data
0x020A 0x00010F38                          vdma_constant_n(0x00010F38)
0x020B 0x00000001                          vdma_constant_n(0x00000001)
0x020C 0xFF101000                          vdma_constant_n64(0x87654321FF101000)
0x020D 0x87654321                          // vdma_constant_n64 - upper data
0x020E 0x00010000                          vdma_constant_n(0x00010000)
0x020F 0x000001FF                          vdma_constant_n(0x000001FF)
0x0210 0xFF300000                          vdma_constant_n64(0x87654321FF300000)
0x0211 0x87654321                          // vdma_constant_n64 - upper data
0x0212 0x800100C8                          vdma_constant_n(0x800100C8)
0x0213 0x00000001                          vdma_constant_n(0x00000001)
// Label Listing:
// 0x0000 : "START"
// 0x0001 : "MAIN"
// 0x0001 : "L2P_CHAN0"
// 0x000C : "L2P_CHAN0_RPT_NOT_DONE"
// 0x0013 : "L2P_CHAN0_NO_INT"
// 0x0019 : "L2P_CHAN0_NEXT_DESC"
// 0x001F : "L2P_CHAN0_UPDATE"
// 0x0020 : "L2P_CHAN0_UPDATE_B"
// 0x0021 : "L2P_CHAN0_END"
// 0x0200 : "L2P_CHAN0_SIZE"
// 0x0201 : "L2P_CHAN0_BASE"
// 0x0202 : "L2P_CHAN0_INDEX"
// 0x0203 : "L2P_CHAN0_CNT"
// 0x0204 : "L2P_CHAN0_RPT"
// 0x0206 : "L2P_CHAN0_SYSA"
// 0x0208 : "L2P_CHAN0_LIST"
// 0x0100 : "ZERO"
// 0x0102 : "MINUS1"
// 0x0103 : "THREE"
// 0x0104 : "FOUR"
```

The microcode in Figure 5-7 uses the algorithm of Figure 5-4. Since developing microcode like this can be very tedious, Gennum provides a framework to simplify microcode generation. This is provided as part of the Test_Builder framework used in the GN412x simulation test bench[1].

The microcode for Figure 5-7 was created using the "C" based Test_Builder framework with the source code of:

**Figure 5-8: C Source Code for Example 2MB Contiguous Memory Buffer**

```
//==============================================================================================
// Define the Memory Map
//==============================================================================================
#define BAR0_BASE     0xFF00000010000000ll
#define BAR1_BASE     0xFF000000A0000000ll
#define BFM_BAR0_BASE 0x87654321F0000000ll
#define BFM_BAR1_BASE 0xBB00000040000000ll

#define CHAN0_DESC_LIST_SIZE 3

#define L2P_CHAN0_SUB_DMA_LENGTH 0x1000
#define L2P_CHAN0_XFER_CTL 0x00010000

//==============================================================================================
// Define the SG List
//==============================================================================================
struct sg_entry_struct sg_list_chan0[] =
{
    { BFM_BAR0_BASE|0xFF1000C8, L2P_CHAN0_XFER_CTL | 0xF38, 1 },
    { BFM_BAR0_BASE|0xFF101000, L2P_CHAN0_XFER_CTL | (L2P_CHAN0_SUB_DMA_LENGTH & 0xFFF), 511 },
    { BFM_BAR0_BASE|0xFF300000, L2P_CHAN0_XFER_CTL | 0x0C8 | 0x80000000, 1 }, //Assert an interrupt
    { 0x011, 0, 0 }
};

//**********************************************************************************************
//**
//** vdma_main: This will insert the DMA microcode and data into the test script
//**
//**  The last function call, vdma_process(), will cross reference all of the labels in the
//**  source code so that you end up with the proper hexadecimal values that need to be written
//**  to descriptor RAM.
//**
//**********************************************************************************************

void vdma_main()
{

    vdma_org(0x0000);            //This initializes the program address counter
    data_address = 0x200;       //This initializes the data space address counter
    vdma_label("START");
        vdma_nop();    //do nothing

//==============================================================================================
// START of the main program loop
//==============================================================================================
    vdma_label("MAIN");
        vdma_channel_service_4
        (
```

1. Refer to the document: "GN412x Simulation Test Bench User Guide" for more details.

```
            "L2P_CHAN0",               //label to be used for this specific channel
            'l',                       //direction='l' for l2p or 'p' for p2l
            0,                         //The event register bit to be used for interrupt generation
            _EXT_COND_0,               //External condition used for this channel
            _EXT_COND_LO,              //Set to either _EXT_COND_LO or _EXT_COND_HI
            0,                         //set to non zero when the list will be updated dynamicaly
            CHAN0_DESC_LIST_SIZE,      //SYS_ADDR step size of list entries that have a repeat count
            L2P_CHAN0_SUB_DMA_LENGTH, //Step size of list entries that have a repeat count
            sg_list_chan0              //SG List itself
        );

        vdma_nop();    // This is not required (can be replaced with more channel servicing)
        vdma_jmp(_ALWAYS, 0,"MAIN");    //loop forever

//=================================================================================================
// END of the main program loop
//=================================================================================================

//-------------------------------------------------------------------------------------------------
// Global Constants
//-------------------------------------------------------------------------------------------------
    vdma_org(0x0100);
    vdma_label("ZERO");
        vdma_constant_n64(0);           //The constant 0
    vdma_label("MINUS1");
        vdma_constant_n(0xFFFFFFFF);    //The constant -1
    vdma_label("THREE");
        vdma_constant_n(3);             //The constant 3
    vdma_label("FOUR");
        vdma_constant_n(4);             //The constant 4

//=================================================================================================
// Must run vdma_process to resolve the cross-references and generate the microcode
//=================================================================================================
    vdma_process(BAR0_BASE + 0x4000);

}
```

Most of the work is done in the function "`vdma_channel_service_4`" which is a generalized channel servicing function for 4DW descriptors based on the algorithm of Figure 5-4. The data structure "`sg_list_chan0`" contains the information shown in Figure 5-5. the data structure could easily be changed to accommodate a host memory semaphore is described in Section 4.2.1.3. The data structure could be modified as:

```
struct sg_entry_struct sg_list_chan0[] =
{
    { BFM_BAR0_BASE|0xFF1000C8, L2P_CHAN0_XFER_CTL | 0xF38, 1 },
    { BFM_BAR0_BASE|0xFF101000, L2P_CHAN0_XFER_CTL | (L2P_CHAN0_SUB_DMA_LENGTH & 0xFFF), 511 },
    { BFM_BAR0_BASE|0xFF300000, L2P_CHAN0_XFER_CTL | 0x0C8, 1 },
    { BFM_BAR0_BASE|0xFF400000, 0x00080004 | 0x80000000, 1 }, //Semaphore and interrupt
    { 0x011, 0, 0 }
};
```

Here, a fourth descriptor is added to do the semaphore transfer. The example uses STREAM_ID=0x08 as the source of data for the semaphore (the data would typically be the value zero as described in Section 4.2.1.3). In order to include the additional entry in the scatter gather list, the constant "`CHAN0_DESC_LIST_SIZE`" should be changed to 4.

From this modified example, it is clear that no change to the microcode is required in order to facilitate the host memory semaphore.

Additional channels may be accommodated by modifying the function "`vdma_main`" to make additional calls to the function "`vdma_channel_service_4`" or its counterpart "`vdma_channel_service_3`" if the 3DW list type is desired.

# 6. Conclusion

The GN412x together with the Gennum FPGA IP, provides a very powerful and flexible platform that is capable of handling many DMA scenarios. The use models described here are based on practical considerations about how host systems operate, in terms of both hardware and software. However, there are certainly other use models that could be developed that would also fit within the programmability of the FlexDMA controller.