Texas Instruments

Application Report
SPRABF3—November 2010

Connecting AIF2 to FFTC Guide for KeyStone Devices

High-Performance and Multicore Processors

Abstract/Purpose/Benefit

The purpose of the document is to explain how to configure AIF2 and FFTC so that they can work together seamlessly in any usage scenarios.

Contents

List of Tables

List of Figures

1 Introduction

The Antenna Interface Subsystem (AIF2) and Fast Furrier Transform Coprocessor Subsystem (FFTC) are useful for OFDM-based applications like the LTE physical layer. Both of the peripherals are Packet DMA (PKTDMA) peripherals and data need to be passed in and transferred out by the PKTDMA engine and managed by the Queue Manager Subsystem (QMSS) queues. This application note focuses on how to configure AIF2 and FFTC so that they can work together seamlessly. The description is focused on how to set up the PKTDMA from a high-level perspective, which is comprised of the QMSS and PKTDMA subsystems. It is beyond the scope of this document to configure registers in these peripherals.

2 Basic Elements for Using PKTDMA with KeyStone Devices

When PKTDMA is used, it involves setting up several elements. On the transmit side, it needs to set up the Tx PKTDMA channel, Tx queue, and Tx completion queue. On the receive side, it needs to set up the Rx PKTDMA channel, Rx flow, Rx free descriptor queue (FDQ) and Rx queue. In this application note, Tx queue and input queue are used interchangeably, and Rx queue, destination queue, and output queue are used interchangeably.

When a job needs to be submitted to a PKTDMA peripheral, a packet descriptor is prepared and pushed into the Tx queue of the peripheral. The Tx PKTDMA will pop the descriptor from the Tx queue, stream the data, and recycle the descriptor to the Tx completion queue. Often the packet descriptors can be prepared in advance and held in a queue, which we call Tx free descriptor queue (FDQ) in the document. When a packet is pushed to a Tx queue, it is required that packet descriptor size is pushed together with the descriptor address using the 4 LSB of the descriptor address. However, when the Tx PKTDMA recycles the packet descriptor to the Tx completion queue, the size information is not pushed to the Tx completion queue. Therefore the Tx completion queue can not be one of the Tx queues. But the Tx completion queue can still act as a holding queue for free descriptors. Throughout this document, we interchangeably use Tx completion queue and Tx FDQ whenever there is no confusion.

For receive, the Rx PKTDMA uses a flow which tells the Rx PKTDMA how to process the Rx packet. Each PKTDMA peripheral supports a pre-defined number of flows and the flow setups are stored in the flow table. When PKTDMA needs to transfer a packet, the Rx PKTDMA will use the flow ID to find the Rx FDQ associated with the flow in the flow table, pop one or more packet descriptors if necessary when host packet is used, stream the data and push the packet descriptor to the Rx queue. Usually the flow ID comes from the packet descriptor that is submitted to the Tx queue. For FFTC it is the *src_tag_lo* field in the Tx packet descriptor that is used as flow ID. But for AIF2 the flow ID doesn't come from the packet descriptor, instead it has a one to one mapping with the PKTDMA channel number. Rx FDQ needs to be populated before Rx PKTDMA can send packets out.

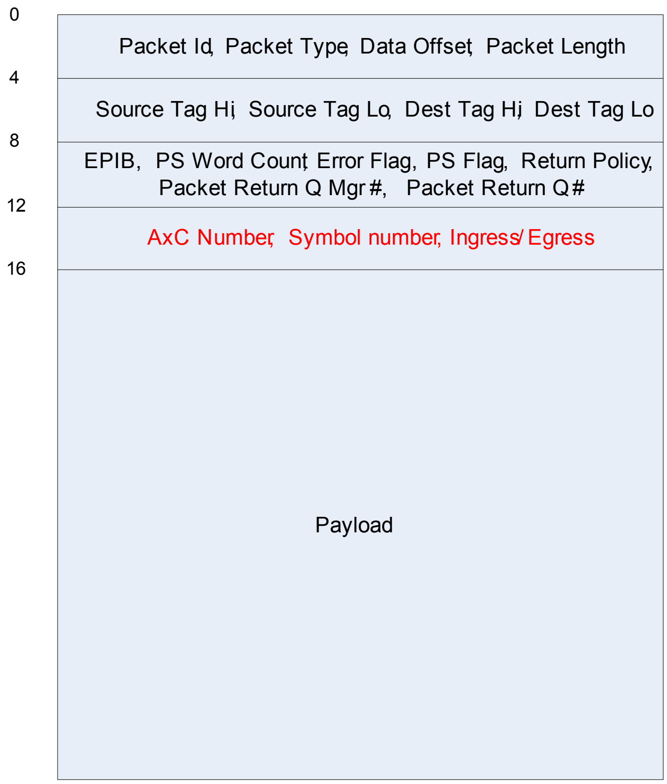
QMSS supports a total of 8192 queues, among which Tx queues are designated queues for PKTDMA peripherals and they are hard allocated at time of chip layout. All the other queues are available for general purpose use, although some can have special functionality associated with them like accumulation. The usage of them depends on the application. Rx FDQ and Tx completion queue can be any queue other than Tx queues, and Rx queue can be any queue including Tx queues if it is intended to forward the packet to another PKTDMA peripheral.

3 AIF2 Packet for Antenna Traffic

3.1 AIF2 Packet Descriptor Format

AIF2 can be set up to work in DIO mode or packet mode. DIO mode is not the focus of this application note, but it will be briefly discussed later. Although AIF2 supports both monolithic packet and host mode packet types, when AIF2 is used to transfer antenna traffic, it is mandatory to use the monolithic packet type. The AIF2 packet format for antenna data is shown in Figure 1.

Figure 1 AIF2 Packet Format for Antenna Data


0
Packet Id, Packet Type, Data Offset, Packet Length
4
Source Tag Hi, Source Tag Lo, Dest Tag Hi, Dest Tag Lo
8
EPIB, PS Word Count, Error Flag, PS Flag, Return Policy, Packet Return Q Mgr #, Packet Return Q #
12
AxC Number, Symbol number, Ingress/Egress
16
Payload

The highlighted words in red contain protocol specific (PS) data for AIF2. AIF2 can have another form of protocol specific data which contains OBSAI address and ingress/egress indication, which is beyond the scope of this application note.

One important thing to note here, from the PKTDMA standpoint the monolithic packet format can be different than what is shown in Figure 1, because optional EPIB and NULL regions may be present in the packet descriptor. However for performance reasons, it is strongly recommended to use the packet format shown in Figure 1 for AIF2 antenna traffic, especially with egress antenna traffic. When there are a large number of antenna carriers to support, it is strongly recommended to set *tx_AIF_mono_mode* in the Tx PKTDMA channel configuration of AIF2 to guarantee the real time requirement for DMA. When *tx_AIF_mono_mode* is set, the packet descriptor is assumed to have the fixed form shown in Figure 1, which means 12 bytes of the mandatory header for a monolithic packet, 4 bytes of the protocol specific data, and payload starts at an offset of 16 bytes from the beginning of the packet descriptor.

Tx PKTDMA will fetch the packet descriptor and three quad-words of the payload together in one transaction of four quad-words (64 bytes) instead of fetching the packet descriptor and the payload in separate transactions as it normally does when *tx_AIF_mono_mode* is not set.

Another point of interest, AIF2 does not necessarily guarantee that the packets in the output queue are in antenna order. AIF2 processes the packets in the order of arrival. When multiple antenna carriers arrive at different times, the order of packets in the output queue will be the same as the arrival order. When multiple antenna carriers arrive at the same time, if they are within the same AIF2 link, as the data of antenna carriers are time-division multiplexed together, the order of packets in the output queue will be the same as the order of how they are multiplexed. If they are across multiple AIF2 links, because of the timing variations from link to link, it is not guaranteed which link will be processed first. Therefore, the order of the output packets cannot be guaranteed.

3.2 Setting Up AIF2 Ingress Rx FDQ Packet Descriptors

On ingress traffic, it is necessary to set up an Rx FDQ per flow. Different flows can use different Rx FDQs or share the same Rx FDQ queue which can be set in the flow table. Depending on the application, different fields of the packet descriptor in the Rx FDQ need to be set accordingly.

Due to the Rx overwrite feature, only the fields that will not be overwritten by Rx PKTDMA need to be set, which include:

- Return push policy
- Packet return queue mgr #
- Packet return queue #
- Source tag hi (configurable)
- Source tag lo (configurable)
- Dest Tag hi (configurable)
- Dest tag lo (configurable)

When the AIF2 output queue is connected directly to another PKTDMA peripheral's input queue, then the first 3 fields in the above list have to be set so that the Tx PKTDMA of the next PKTDMA peripheral knows how to recycle the packet descriptor. Otherwise, the application code can make the recycle decision independent of what is set in the descriptor. For the next 4 fields, the flow configuration determines whether or not each is independently overwritten. It is only when the Rx overwrite feature is not configured then the value set in the packet descriptor in Rx FDQ will appear in the Rx packet. However AIF2 doesn't use those 4 fields set in the packet descriptors in Rx FDQ.

All the other fields shown in Figure 1 in the Rx packet will be overwritten by the Rx PKTDMA. Note if the flow is configured to have EPIB in the Rx packet, since AIF2 doesn't provide any EPIB information, the EPIB fields in the Rx packet will be filled with 0s. In other words, the EPIB information is either not present or all 0s in the AIF2 Rx packets.

The final values appearing in Rx packet are a result of the joint configuration of Rx FDQ packet descriptor and the flow table.

3.3 Setting Up AIF2 Egress Packet Descriptors

For egress packet, the mandatory fields are:

- Packet ID
- Data offset
- Packet length
- Protocol specific valid word count
- Return push policy
- Packet return queue mgr #
- Packet return queue #
- AxC number
- Symbol number
- Egress

It is fine to set all the necessary fields in the packet descriptor shown in Figure 1 based on the application requirements. However the following fields are not used by AIF2:

- Packet type
- Source tag hi
- Source tag lo
- Dest Tag hi
- Dest tag lo
- EPIB info block present flag
- Error flags
- Protocol specific flags
- EPIB info if present
- PS words more than 4 bytes

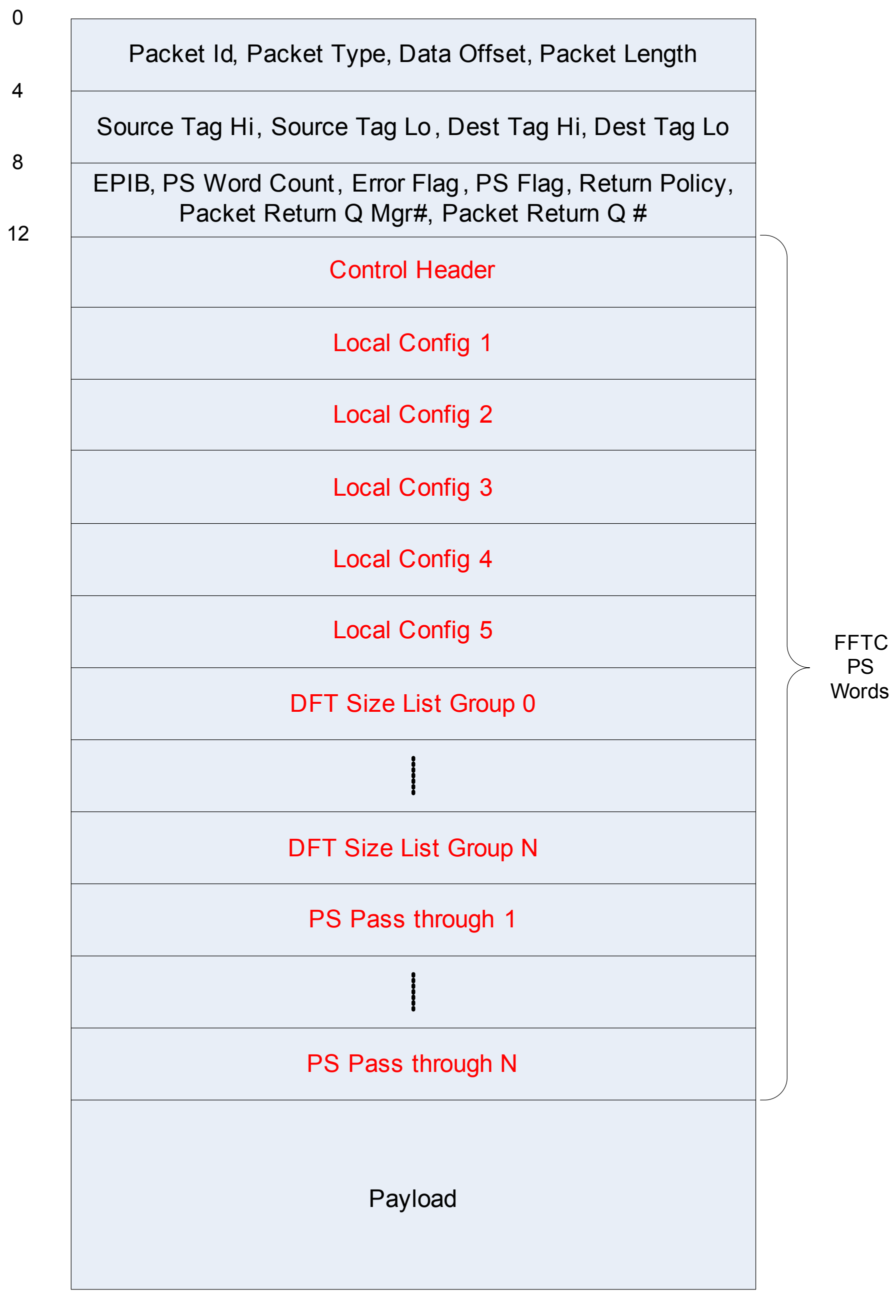
4 FFTC Packet Format

4.1 FFTC Packet Descriptor Format

FFTC supports both monolithic and host mode packets. The input and output can choose to use either mode independently. It is up to the application to choose which packet mode to use.

The monolithic input packet format is shown in Figure 2.

Figure 2 FFTC Input Packet Monolithic Packet Format


0
4
8
12
Packet Id, Packet Type, Data Offset, Packet Length
Source Tag Hi, Source Tag Lo, Dest Tag Hi, Dest Tag Lo
EPIB, PS Word Count, Error Flag, PS Flag, Return Policy, Packet Return Q Mgr#, Packet Return Q #
Control Header
Local Config 1
Local Config 2
Local Config 3
Local Config 4
Local Config 5
DFT Size List Group 0
DFT Size List Group N
PS Pass through 1
PS Pass through N
FFTC PS Words
Payload

The highlighted fields in red are protocol-specific words for FFTC packet. All of them are optional and the application can choose to use them selectively. If any are present they should appear in the order depicted. All five of the local configuration words need to be specified if local configuration words are present. Control header specifies whether or not local configuration words, DFT size list and pass through fields are present.

However, the pass-through field can also be present without control header in the packet. Whether or not the control header is present is specified by the protocol specific flag (PS flag) in word 2 of the monolithic packet. For easy reference here, several tables regarding the PS words are taken from the FFTC user's guide. The PS flag is given in Table 1.

Table 1 Bit Description for TX PS Flags

Bit	Field	Description
0	Header Present	0: Control header not present in packet. 1: Control header present in packet.
1	Debug Halt	0: Do nothing 1: Issue a halt
2	EOP Interrupt	0: Do nothing 1: FFTC generates the EOP interrupt.
3	Reserved	
End of Table 1		

Control header is given in Table 2.

Table 2 PS Control Header Word

Bit	Field	Description
31:29	RESERVED	
28:24	PS_field_length	The length of the pass-through data, in 32-bit words (1 to 4)
23:21	RESERVED	
20:16	DFT_sizes_list_length	Indicates the length of the DFT sizes list in 32-bit words (1 to 26)
15:3	RESERVED	
2	PS pass-through present	Indicates that there is a protocol specific field that should be forwarded to the receiver 0 – Pass-through word not present 1 – Pass-through word present
1	DFT sizes list present	Indicates that the list of DFT sizes is present 0 – DFT sizes list not present. 1 – DFT sizes list present.
0	Local configuration data present	Indicates that the five local control registers are present. If present, configuration data is always 5 32-bit words. 0 – Configuration word not present. 1 – Configuration word present.
End of Table 2		

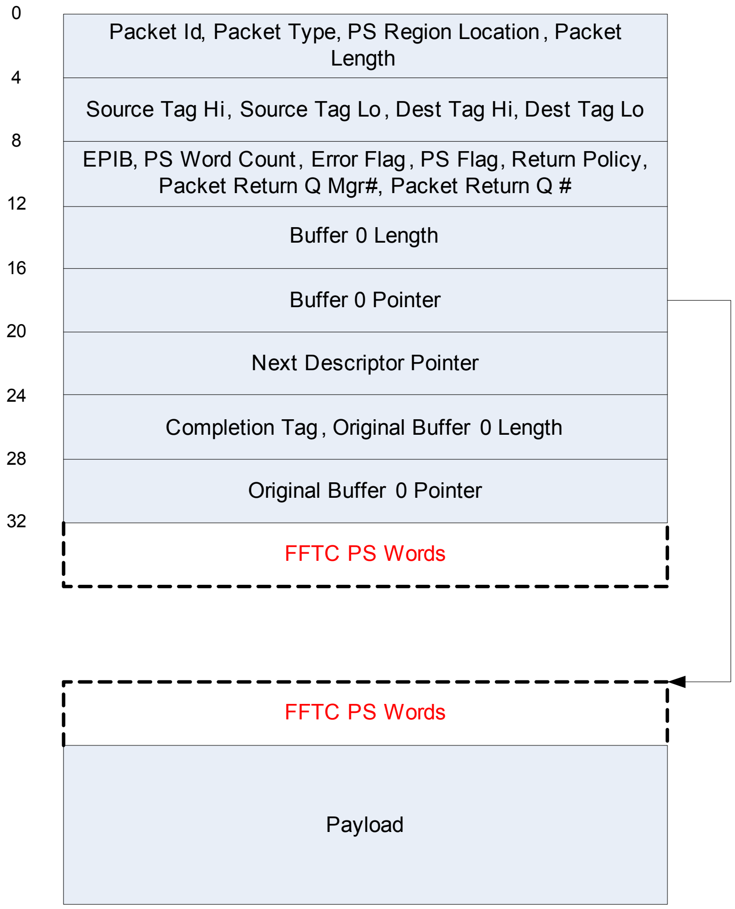
If local configuration words are present, then all 5 of them must be specified. Each word maps to a corresponding FFTC configuration register. The mapping of the words and registers are shown in Table 3. Although the local configuration words belong to a packet, the configuration changes the register setting for the input queue to which the packet is submitted. Therefore it affects all the packets submitted to that queue afterwards until the register setting is changed again either directly or through the local configuration words of another packet submitted to the queue.

Table 3 PS Local Configuration Words

Word	Register
1	FFTC_QUEUE_x_DESTINATION_QUEUE_AND_SHIFTING REGISTER
2	FFTC_QUEUE_x_SCALING _REGISTER
3	FFTC_QUEUE_x_CYCLIC_PREFIX_REGISTER
4	FFTC_QUEUE_x_CONTROL_REGISTER
5	FFTC_QUEUE_x_LTE_FREQUENCY_SHIFT_REGISTER
End of Table 3	

The host packet format is shown in Figure 3.

Figure 3 FFTC Input Packet Host Packet Format

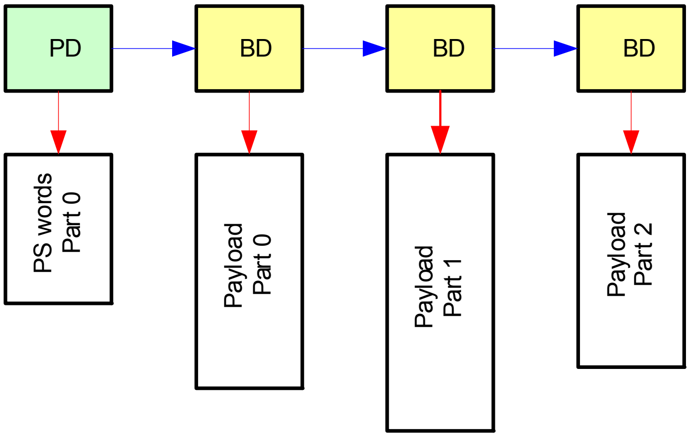

0
4
8
12
16
20
24
28
32
Packet Id, Packet Type, PS Region Location, Packet Length
Source Tag Hi, Source Tag Lo, Dest Tag Hi, Dest Tag Lo
EPIB, PS Word Count, Error Flag, PS Flag, Return Policy, Packet Return Q Mgr#, Packet Return Q #
Buffer 0 Length
Buffer 0 Pointer
Next Descriptor Pointer
Completion Tag, Original Buffer 0 Length
Original Buffer 0 Pointer
FFTC PS Words
FFTC PS Words
Payload

The highlighted fields are protocol-specific words, which are the same as in the monolithic packet, so it is condensed into one block for simplicity of the picture. There are two choices for where the PS words can be in host mode, at the end of the packet descriptor or at the start of payload (SOP), as shown by the dashed line rectangle boxes.

When it is at the end of the packet descriptor, it must be there in its entirety. When it is at the SOP, due to the nature of the host mode packet, it doesn't have to be co-located with the payload. However it has to be in one contiguous buffer in its entirety. This is illustrated in Figure 4.

The example shown in Figure 4 has one buffer for PS words, and three parts of payload buffer. The blue arrows show the linked packet descriptors and buffer descriptors. The red arrows show the buffer pointers. Although FFTC will use the existing register configurations to process the input packet unless the register setting is changed directly or through the local configuration words of the input packet, this feature can be useful when the PS words are the same for many packets and homogeneous processing of them is desired. Therefore there is no need to repeat PS words in every packet so as to save some memory. This feature is also particularly useful when connecting AIF2 and FFTC in the advanced use case without DSP intervention, which will be explained later.

Figure 4 FFTC input packet host packet example



4.2 Setting Up FFTC Tx Packet Descriptor

For an input packet to FFTC, the mandatory fields to be set are:

- Packet ID
- Packet length
- Source tag lo – if Qx_FLWID_OVRD is not set
- Protocol specific valid word count
- Protocol specific flags
- Return push policy
- Packet return queue mgr #
- Packet return queue #

If the packet is a host mode packet, the additional mandatory fields for the input packet include:

- Protocol specific region location – if PS words are present
- Return policy
- Buffer length
- Buffer pointer
- Next descriptor buffer

The following fields can be set in the Tx packet which are not used by FFTC but can be configured to pass through to Rx packet

- Packet type
- Source tag hi
- Dest tag hi
- Dest tag lo

Note that *source tag lo* field can also be configured to pass through to Rx packet. The following fields are dropped by FFTC:

- EPIB info block present
- Error flags
- EPIB info
- Completion tag
- Original buffer length – if host packet
- Original buffer pointer – if host packet

Note that although original buffer length and original buffer pointer fields are not used by FFTC for Tx packet, it is a good coding practice to fill them as they will not be overwritten. Therefore it may be useful sometime or helpful for debugging purposes.

4.3 Setting Up FFTC Rx FDQ Descriptors

It is necessary to set up at least one Rx FDQ for each flow. Different flows can have different Rx FDQs or share the same Rx FDQ queue which can be set in the flow table. Which flow to use can be different from packet to packet or on an input queue basis. For each input queue, the Qx_FLWID_OVRD bit of the FFTC_CONFIGURATION_REGISTER can be set to configure the flow used. When Qx_FLWID_OVRD is set, the incoming queue number will be used as the flow ID. If the Qx_FLWID_OVRD is not set, then the source tag lo field in the Tx packet will be used as the flow ID.

The mandatory fields to set in the packet descriptor if host packet is used include:

- Original buffer length – if host packet
- Original buffer pointer – if host packet

Due to Rx overwrite, only the fields that will not be overwritten by Rx PKTDMA need to be set, which include:

- Return policy – if host packet
- Return push policy
- Packet return queue mgr #
- Packet return queue #
- Source tag hi (configurable)
- Source tag lo (configurable)
- Dest Tag hi (configurable)
- Dest tag lo (configurable)

When the FFTC output queue is connected to another PKTDMA peripheral's input queue directly, then the first 3 (or 4 if host packet) fields in the above list have to be set so that the Tx PKTDMA of the next PKTDMA peripheral knows how to recycle the packet descriptor. Otherwise, the application can make the recycle decision independent of what is set in the descriptor. For the next 4 fields, the flow configuration determines whether or not each is independently overwritten. It is only when the Rx overwrite feature is not configured then the value set in the packet descriptor in Rx FDQ will appear in the Rx packet. However FFTC doesn't use those 4 fields set in the packet descriptors in Rx FDQ.

If the flow is configured to have EPIB in the Rx packet, since FFTC doesn't provide any EPIB information, the EPIB fields in the Rx packet will be filled with 0s. In other words, the EPIB information is either not present or all 0s in the FFTC Rx packets.

The final value which will appear in Rx packet is a result of joint configuration of Rx FDQ packet descriptor and the flow table.

5 High Level View of Data Flow Between AIF2 and FFTC

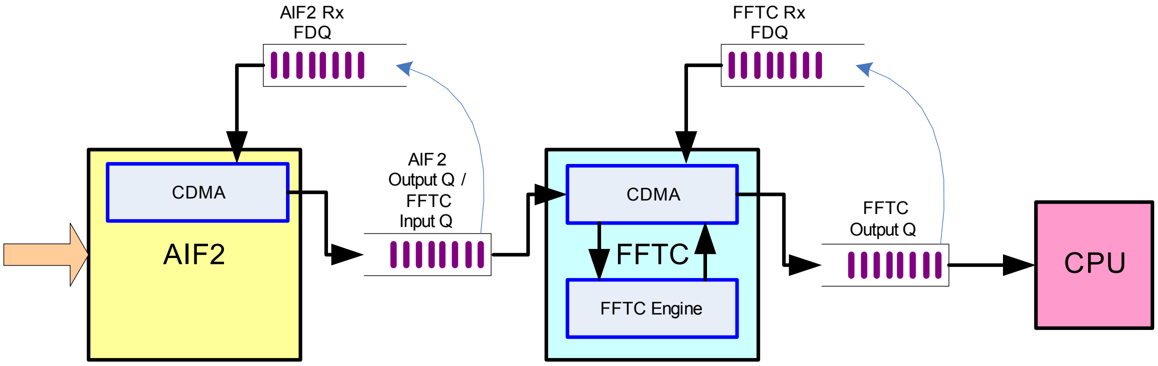
5.1 Inbound Data Flow

When antenna data arrives at AIF2, the Rx PKTDMA of AIF2 will pop a packet descriptor from the Rx free descriptor queue (FDQ) for the antenna carrier, fill the data in the buffer and push the packet descriptor in the output queue when data is complete for the packet. AIF2 can support up to 128 antenna carriers and each antenna carrier corresponds to one PKTDMA channel and one flow.

Since FFTC is also a peripheral using PKTDMA, it is possible to connect the AIF2 and FFTC directly through queues. In this case, the output queue of AIF2 can be specified as one of the input queues of FFTC. When AIF2 pushes the packet descriptor into the queue, it triggers the QPEND status of the corresponding queue. Then the Tx PKTDMA of FFTC will pop the descriptor from the input queue and start transferring the data. The FFTC then begins to process the data. When the processing completes, the Rx PKTDMA will pop a descriptor from FFTC Rx FDQ, transfer the results into the buffer and push the packet descriptor to the FFTC output queue.

The output queue of FFTC can be further processed by other PKTDMA modules or the DSP. The data flow is illustrated in Figure 5. The blue curved arrows indicate the possible routes for recycling. After the packet in the input queue of FFTC is retrieved by FFTC Tx PKTDMA, if there is no other module that needs to use the same data, then the packet can be set to return to the AIF2 Rx FDQ which is performed automatically by FFTC Tx PKTDMA. Depending on the next stage of processing after the FFTC, either the DSP needs to recycle the FFTC output queue back to the FFTC Rx FDQ, or the next PKTDMA module can do it automatically.

Figure 5 Inbound Data Flow Between AIF2 and FFTC with Direct Connection

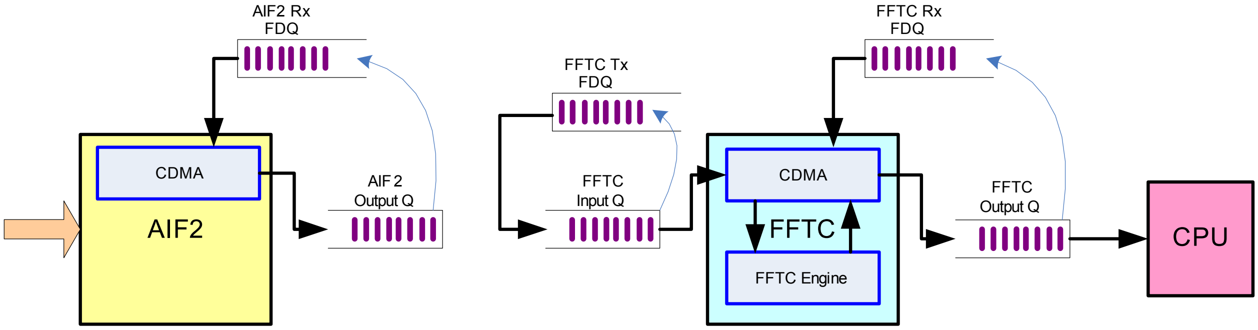


Since it is not always desirable to have the output queue of AIF2 be the same as the input queue of FFTC, they can be set to use different queues. This is shown in Figure 6. In this case, when a packet descriptor is pushed into the AIF2 output queue, the packet needs to be reconstructed by popping a descriptor from the FFTC Tx FDQ, linking the data buffer to the received antenna data then pushing the packet into a FFTC input queue. This can be done using the DSP or using other methods, which will be explained later.

When it comes to recycling, AIF2 output queue, FFTC input queue and FFTC output queue all need to be recycled. The blue curved arrows in Figure 6 indicate the possible recycle routes. In this figure, only FFTC input queue can be automatically recycled by FFTC Tx PKTDMA. The other two need to be recycled by the DSP or using other methods. Because the reconstructed packets are recycled by FFTC Tx PKTDMA while

AIF2 output packets are recycled by other methods, the timing of the recycling may not be aligned. Therefore it is important to allocate enough descriptors in AIF2 Rx FDQ so that it will not starve or the recycled packets will not be overwritten before it is used by FFTC.

Figure 6 Inbound Data Flow Between AIF2 and FFTC Without Direct Connection

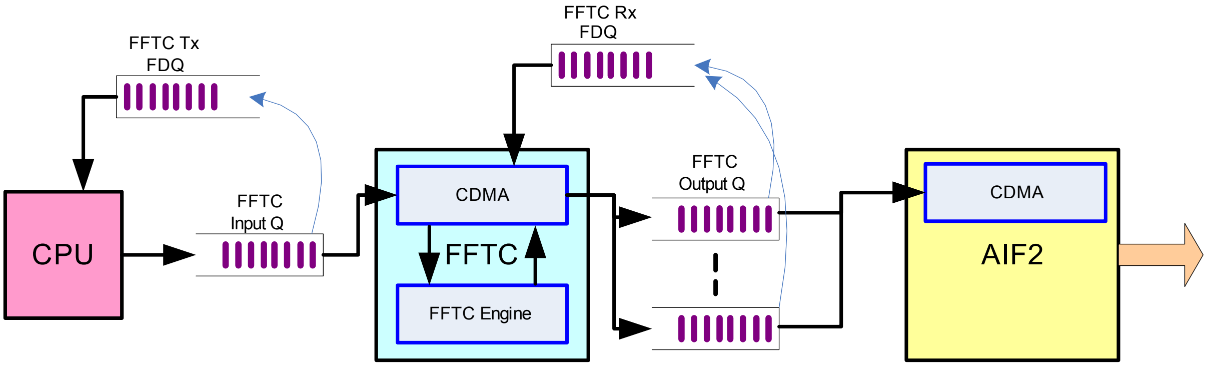


In AIF2, each antenna carrier corresponds to a separate flow, and each flow can be set up to have different Rx FDQs and output queues, they can also be set up to have the same Rx FDQ and output queue independently.

5.2 Outbound Data Flow

On the outbound side, it is the DSP's responsibility to feed the input queue of FFTC either directly or by other mechanisms. For example, the DSP needs to pop a packet descriptor from Tx FDQ, configure all the fields of the packet descriptor, prepare the payload data, and push the packet descriptor into one of the input queues of FFTC. The Tx PKTDMA of FFTC will pop the descriptor, stream the data in and recycle the packet descriptor to the queue specified in the packet descriptor. When FFTC is finished processing, it pops a packet descriptor from Rx FDQ, fills in the data and deposits the packet descriptor into the output queue specified either in the flow table or in the packet descriptor. If no other processing is needed before data goes out to the antenna interface, then the output queue of FFTC can be one of the input queues of AIF2. AIF2 can support up to 128 antenna carriers, and each antenna carrier needs to have a separate queue on the outbound traffic. After the Tx PKTDMA of AIF2 pops the packet from its input queue, it will stream the data out and recycle the packet descriptor to the queue specified in the packet descriptor. The blue curved arrows show a sample recycling path.

Figure 7 Outbound Data Flow Between FFTC and AIF2



6 Connecting AIF2 and FFTC for Inbound Antenna Traffic

6.1 Basic Pass-Through Usage

This section describes a very basic use case for connecting AIF2 and FFTC on inbound traffic. The AIF2 output queue is one of the input queues of FFTC, so they are connected directly. As shown in Figure 5, this use case uses 4 logically different kinds of queues: AIF2 Rx FDQ, FFTC Rx FDQ, output queue of AIF2 (which is one of the input queues of FFTC) and output queue of FFTC. AIF2 Rx FDQ and FFTC Rx FDQ can be any queue that is not a Tx queue of a PKTDMA peripheral. The output queue of FFTC can be any queue that is not a Tx queue of a PKTDMA peripheral or can be a Tx queue of another PKTDMA peripheral if FFTC output is taken and processed by the PKTDMA peripheral directly.

Example 1: One or more antennas direct connection between AIF2 and FFTC

This example shows how to configure the AIF2 and FFTC such that they can connect together through queues. All the antenna carrier output will be directed to a single FFTC input queue, be processed in the same way by FFTC, then go to a single destination queue.

This example assumes that antenna carrier 1 of AIF2 is used, therefore AIF2 flow 1 is used. When there are multiple antenna carriers, more AIF2 flows are used but all flows can be configured to have the same parameters. Assume FFTC_A input queue 1 is used and FFTC_A flow 1 is used. Note that for FFTC, the input queue and flow don’t necessarily have any connection and they can be independently chosen by application. Monolithic packets are used for FFTC output.

- AIF2 flow 1 setup

```
FLOW_TBL *flowPtr;

/* there is PS info in the Rx packet */

flowPtr -> Reg_A.rx_psinfo_present = 1;

/* Rx packet is a monolithic packet, so the Rx FDQ packet descriptors need to be
monolithic packet type */

flowPtr -> Reg_A.rx_desc_type = 2;

/* payload starts at the offset of 16 bytes in the packet descriptor */

flowPtr -> Reg_A.rx_sop_offset = 16;

/* the output queue of this flow is FFTCA input queue 1 */

flowPtr -> Reg_A.rx_dest_qmgr = (FFTCA_INPUT_Q1 >> 12) & 0x3;

flowPtr -> Reg_A.rx_dest_qnum = FFTCA_INPUT_Q1 & 0x0fff;

/* rx packet descriptor src_tag_lo field is set not to be overwritten by the Rx
Packet DMA */

flowPtr -> Reg_C.rx_src_tag_lo_sel = 0;

/* this flow uses AIF2_RX_FDQ as the FDQ */

flowPtr -> Reg_E.rx_fdq0_sz0_qmgr = (AIF2_RX_FDQ >> 12) & 0x3;

flowPtr -> Reg_E.rx_fdq0_sz0_qnum = AIF2_RX_FDQ & 0x0fff;
```

Other fields of the flow table can be cleared to 0. The key thing to note here is that by setting *rx_src_tag_lo_sel* in register C to 0, AIF2 Rx PKTDMA will not overwrite the *src_tag_lo* field specified in the packet descriptor in Rx FDQ.

- AIF2_RX_FDQ packet descriptor setup

```
MONOLITHIC_DESCRIPTOR *pdPtr;

/* src_tag_lo specified here will not be overwritten by Rx Packet DMA because it is
set in the flow register. Since the output queue of AIF2 is the input queue of FFTC,
src_tag_lo field will be used by FFTC as the flow ID for FFTC which is 1 */

pdPtr -> src_tag_lo = 1;

/* this information is used by FFTC Tx Packet DMA and the value can be set to other
value which depends on the application */

pdPtr -> ret_push_policy = 0;

/* the packet descriptor will be recycled to AIF2_RX_FDQ by FFTC Tx Packet DMA */

pdPtr -> pkt_return_qmgr = (AIF2_RX_FDQ >> 12) & 0x3;

pdPtr -> pkt_return_qnum = AIF2_RX_FDQ & 0x0fff;
```

The Rx FDQ can be set up to have as many descriptors as needed based on the application. All of them can have the same parameters.

Since in flow setup, it is set not to overwrite the *src_tag_lo* field. The *src_tag_lo* field set in the packet descriptors in Rx FDQ, which is 1, will be preserved in the AIF2 output packet. When this packet is presented to FFTC, FFTC will take the *src_tag_lo* as the flow ID to use. This is one of the methods to configure the FFTC flow ID through the AIF2 packet. There are other methods to set this field which will be explained in other examples later.

Rx PKTDMA of AIF2 doesn’t need the recycling information. However since the output queue of AIF2 is the input queue of FFTC, the Tx PKTDMA of FFTC needs to know the recycling policy and which queue to recycle the packet descriptor to. Other fields of the packet descriptor will be overwritten by the Rx PKTDMA based on flow setup, so it is not necessary to set them here.

Because monolithic packets have packet descriptor headers that are contiguous with the data buffer and don’t support linked packets, Rx PKTDMA doesn’t check if the buffer is big enough to hold the data. It is up to the application to make sure that the buffer is big enough. The maximum allowed payload buffer is 65535 bytes. Rx PKTDMA will place the data contiguously in the payload buffer until the end of the actual payload or until it reaches 65535 bytes, and writes the actual size to the packet length field.

- FFTC flow 1 setup

```
FLOW_TBL *flowPtr;

/* there is PS info in the Rx packet, as we want the PS words of AIF2 to be carried
to the output of FFTC */
flowPtr -> Reg_A.rx_psinfo_present = 1;

/* Rx packet is a monolithic packet, so the Rx FDQ packet descriptors need to be
monolithic packet type */
flowPtr -> Reg_A.rx_desc_type = 2;

/* payload starts at the offset of 16 bytes in the packet descriptor */
flowPtr -> Reg_A.rx_sop_offset = 16;

/* the output queue of this flow is any desired output queue */
flowPtr -> Reg_A.rx_dest_qmgr = (FFTCA_OUTPUT_Q >> 12) & 0x3;
flowPtr -> Reg_A.rx_dest_qnum = FFTCA_OUTPUT_Q & 0x0fff;
```

```
/* this flow uses FFTC_RX_FDQ as the FDQ */
flowPtr -> Reg_E.rx_fdq0_sz0_qmgr = (FFTC_RX_FDQ >> 12) & 0x3;
flowPtr -> Reg_E.rx_fdq0_sz0_qnum = FFTC_RX_FDQ & 0x0fff;
```

Other fields of the flow table can be cleared to 0. Because the AIF2 doesn't guarantee the order of the packet in the output queue to be in antenna order, it is necessary to have the AIF2 protocol specific words at the output of the FFTC so that the next stage processing knows to which antenna the data belongs by checking the AxC (antenna carrier) number in the descriptor.

- FFTC_RX_FDQ packet descriptor setup

```
MONOLITHIC_DESCRIPTOR *pdPtr;

/* the following information is only needed if FFTC output is sent to another input
queue of Packet DMA peripheral directly and the value can be set to other value
depending on the application, otherwise it is optional if the recycling of FFTC
output queue is handled by host and host can choose what to do */
pdPtr -> ret_push_policy = 0;
pdPtr -> pkt_return_qmgr = (FFTC_RX_FDQ >> 12) & 0x3;
pdPtr -> pkt_return_qnum = FFTC_RX_FDQ & 0x0fff;
```

Other fields of the packet descriptor will be overwritten by the Rx PKTDMA of FFTC based on flow setup, so it is not necessary to set them here.

When AIF2 and FFTC are connected directly through a queue, no FFTC PS words can be added in the packet descriptor. For each input queue of FFTC, specific settings can be set directly in the registers corresponding to the input queue. For LTE inbound antenna data flow, although the cyclic prefix length can be different in each symbol, no special handling is needed as the FFTC takes the last N samples from the CYCLIC_PREFIX_REMOVE_OFFSET when FFT size is N. Since the offset and the samples taken are counted from the tail of data buffer in the packet, different lengths of cyclic prefix are removed from the beginning of the buffer if the packet lengths vary. When cyclic prefix removal is enabled, each packet can only contain a single FFT block.

After AIF2 Rx PKTDMA fills in the fields in the packet descriptor, the output packet of AIF2 will look like:

```
Packet descriptor:
packet_id: 2
packet_type: 0
data_offset: 16
packet_length: (OFDM symbol length + CP length)*size of sample in bytes
source_tag_hi: 0
source_tag_lo: 1
dest_tag_hi: 0
dest_tag_lo: 0
extended_packet_info_block_present: 0
protocol_specifc_valid_word_count: 1
error_flags: 0
protocol_specific_flag: 0
ret_push_policy: 0
pkt_return_qmgr: (AIF2_RX_FDQ >> 12) & 0x3
pkt_return_qnum: AIF2_RX_FDQ & 0x0fff
protocol_specifc_word: AxC = 1, Symbol number within a frame/subframe, ingress
payload: cyclic prefix and OFDM symbol
```

The *packet type* field is a user defined field and can be set in the Ingress Data Buffer Channel Configuration Registers field PKT_TYPE. Error flag is set by AIF2 to indicate whether or not there is error for this packet. In case of an error, the last bit of the *error_flags* is set. The *protocol_specific_flag* field is not used in AIF2 and is set to 0.

This packet will be presented to FFTC which will use flow 1 as specified in the *src_tag_lo* field. Since FFTC flow 1 is configured to have protocol specific data in the output packet, the AIF2 protocol specific data will be passed through to the FFTC output packet. Otherwise the protocol specific data in the input packet will be dropped. Assuming cyclic prefix removal is enabled, the FFTC output packet will look like:

```
Packet descriptor:
packet_id: 2
packet_type: 0
data_offset: 16
packet_length: OFDM symbol length*size of sample in bytes
source_tag_hi: 0
source_tag_lo: 0
dest_tag_hi: 0
dest_tag_lo: 0
extended_packet_info_block_present: 0
protocol_specifc_valid_word_count: 1
error_flags: 0
protocol_specific_flag: 0
ret_push_policy: 0
pkt_return_qmgr: (FFTC_RX_FDQ >> 12) & 0x3
pkt_return_qnum: FFTC_RX_FDQ & 0x0fff
protocol_specifc_word: AxC = 1, Symbol number within a frame/subframe, ingress
payload: FFT output data
```

FFTC passes through *packet type* and *protocol specific flags* from Tx packet to Rx packet. The last bit of *Error flags* is set when there is an error for the packet.

6.2 Intermediate Usage

In some applications, a simple homogeneous setting for all antennas is not always desired. For example, it may be necessary to have different cores process different antenna's data after FFTC. Another example is, since AIF2 can support multiple bandwidths at the same time, in the LTE context, different antenna output may need different frequency shift setup. In this section, more control and flexibility can be programmed to allow different settings for different antennas.

Example 2: Directing different antenna data to different destinations with direct connection between AIF2 and FFTC

This example shows how to set different antenna carriers to different destinations by using one option of Rx PKTDMA overwrite capability. There are four antenna carriers used in this example with two antenna carriers being processed by each of two cores after FFTC.

Assume antenna 1, 2, 3, and 4 are used in AIF2, therefore AIF2 flow 1, 2, 3 and 4 are used. Since there is no need to set the FFTC processing configuration differently in this example, all antenna carrier's output goes to FFTC_A input queue 1. Two flows of FFTC_A are used, FFTC_A flow 1 and flow 2. Both FFTC_A output flows will use host mode packets with protocol specific data in the start of the payload. The output of antenna carrier 1 and 2 of AIF2 will use FFTC_A flow 1, and the output of antenna carrier 3 and 4 of AIF2 will use FFTC_A flow 2. The output of FFTC_A flow 1 will be stored in the L2 memory of core 1 and processed by core 1. The output of FFTC_A flow 2 will be stored in the L2 memory of core 2 and processed by core 2.

For simplicity, the common part of flow or FDQ setup is elaborated for the first example when multiple flows or FDQs are used, and omitted in other flow or FDQ setup.

- AIF2 flow 1 setup

```
FLOW_TBL *flowPtr;
```

```
/* there is PS info in the Rx packet */
flowPtr -> Reg_A.rx_psinfo_present = 1;

/* Rx packet is a monolithic packet, so the Rx FDQ packet descriptors need to be
monolithic packet type */
flowPtr -> Reg_A.rx_desc_type = 2;

/* payload starts at the offset of 16 bytes in the packet descriptor */
flowPtr -> Reg_A.rx_sop_offset = 16;

/* the output queue of this flow is FFTCA input queue 1 */
flowPtr -> Reg_A.rx_dest_qmgr = (FFTCA_INPUT_Q1 >> 12) & 0x3;
flowPtr -> Reg_A.rx_dest_qnum = FFTCA_INPUT_Q1 & 0x0fff;

/* rx packet descriptor src_tag_lo field is overwritten by the src_tag_lo field
specified in the register B here. Since the output queue of AIF2 is the input queue
of FFTC, src_tag_lo field in the Rx packet will be used by FFTC as the flow ID for
FFTC which is 1 */

flowPtr -> Reg_B.rx_src_tag_lo = 1;

flowPtr -> Reg_C.rx_src_tag_lo_sel = 1;

/* this flow uses AIF2_RX_FDQ as the FDQ */
flowPtr -> Reg_E.rx_fdq0_sz0_qmgr = (AIF2_RX_FDQ >> 12) & 0x3;
flowPtr -> Reg_E.rx_fdq0_sz0_qnum = AIF2_RX_FDQ & 0x0fff;
```

- AIF2 flow 2 setup

```
FLOW_TBL *flowPtr;

/* rx packet descriptor src_tag_lo field is overwritten by the src_tag_lo field
specified in the register B here. Since the output queue of AIF2 is the input queue
of FFTC, src_tag_lo field in the Rx packet will be used by FFTC as the flow ID for
FFTC which is 1 */

flowPtr -> Reg_B.rx_src_tag_lo = 1;
flowPtr -> Reg_C.rx_src_tag_lo_sel = 1;
```

- AIF2 flow 3 setup

```
FLOW_TBL *flowPtr;

/* rx packet descriptor src_tag_lo field is overwritten by the src_tag_lo field
specified in the register B here. Since the output queue of AIF2 is the input queue
of FFTC, src_tag_lo field in the Rx packet will be used by FFTC as the flow ID for
FFTC which is 2 */

flowPtr -> Reg_B.rx_src_tag_lo = 2;
flowPtr -> Reg_C.rx_src_tag_lo_sel = 1;
```

- AIF2 flow 4 setup

```
FLOW_TBL *flowPtr;

/* rx packet descriptor src_tag_lo field is overwritten by the src_tag_lo field
specified in the register B here. Since the output queue of AIF2 is the input queue
of FFTC, src_tag_lo field in the Rx packet will be used by FFTC as the flow ID for
FFTC which is 2 */

flowPtr -> Reg_B.rx_src_tag_lo = 2;
flowPtr -> Reg_C.rx_src_tag_lo_sel = 1;
```

Other fields of the flow table can be cleared to 0. The key thing to note here is that by setting *rx_src_tag_lo_sel* in register C to 1, AIF2 streaming interface will use the *src_tag_lo* field set in the flow register to set the *src_tag_lo* field in the Rx packet. When the packet is presented to FFTC, FFTC will take the *src_tag_lo* as the flow ID to use. This is one option demonstrating how the flow used in FFTC is configured through the AIF2 packet.

- AIF2_RX_FDQ packet descriptor setup

```
MONOLITHIC_DESCRIPTOR *pdPtr;
```

```
/* this information is used by FFTC Tx Packet DMA and the value can be set to other
value which depends on the application */
pdPtr -> ret_push_policy = 0;

/* the packet descriptor will be recycled to AIF2_RX_FDQ by FFTC Tx Packet DMA */
pdPtr -> pkt_return_qmgr = (AIF2_RX_FDQ >> 12) & 0x3;
pdPtr -> pkt_return_qnum = AIF2_RX_FDQ & 0x0fff;
```

All antenna carriers can use the same FDQ.

- FFTC flow 1 setup

```
FLOW_TBL *flowPtr;

/* there is PS info in the Rx packet, as we want the PS words of AIF2 to be carried
to the output of FFTC */
flowPtr -> Reg_A.rx_psinfo_present = 1;

/* Rx packet is a host packet, so the Rx FDQ packet descriptors need to be host
packet type */
flowPtr -> Reg_A.rx_desc_type = 0;

/* the PS words will be in front of the payload in data buffer */
flowPtr -> Reg_A.rx_ps_location = 1;

/* the SOP offset is 0, so the first word of the data buffer is the PS word */
flowPtr -> Reg_A.rx_sop_offset = 0;

/* the output queue of this flow is any desired output queue */
flowPtr -> Reg_A.rx_dest_qmgr = (FFTCA_OUTPUT_Q_1 >> 12) & 0x3;
flowPtr -> Reg_A.rx_dest_qnum = FFTCA_OUTPUT_Q_1 & 0x0fff;

/* this flow uses FFTC_RX_FDQ as the FDQ */
flowPtr -> Reg_E.rx_fdq0_sz0_qmgr = (FFTC_RX_FDQ_1 >> 12) & 0x3;
flowPtr -> Reg_E.rx_fdq0_sz0_qnum = FFTC_RX_FDQ_1 & 0x0fff;
```

- FFTC flow 2 setup

```
FLOW_TBL *flowPtr;

/* the output queue of this flow is any desired output queue */
flowPtr -> Reg_A.rx_dest_qmgr = (FFTCA_OUTPUT_Q_2 >> 12) & 0x3;
flowPtr -> Reg_A.rx_dest_qnum = FFTCA_OUTPUT_Q_2 & 0x0fff;

/* this flow uses FFTC_RX_FDQ as the FDQ */
flowPtr -> Reg_E.rx_fdq0_sz0_qmgr = (FFTC_RX_FDQ_2 >> 12) & 0x3;
flowPtr -> Reg_E.rx_fdq0_sz0_qnum = FFTC_RX_FDQ_2 & 0x0fff;
```

- FFTC_RX_FDQ_1 packet descriptor setup

```
HOST_DESCRIPTOR *pdPtr;

/* this information is read by Rx Packet DMA to determine if the buffer size is big
enough to hold the entire packet or not, if not, then another packet descriptor needs
to be popped from the 2nd FDQ specified in the flow */
pdPtr -> original_buffer0_length = symbol length*size of sample in bytes;

/* this information is read by Rx Packet DMA to determine where the buffer is */
pdPtr -> original_buffer0_pointer = fft_output_buffer_core1_ptr;

/* the following information is only needed if FFTC output is sent to another input
queue of Packet DMA peripheral and the value can be set to other which depends on
the application, otherwise it is optional if the recycling of FFTC output queue is
handled by host and host can choose what to do */
pdPtr -> ret_policy = 0;
pdPtr -> ret_push_policy = 0;
pdPtr -> pkt_return_qmgr = (FFTC_RX_FDQ_1 >> 12) & 0x3;
pdPtr -> pkt_return_qnum = FFTC_RX_FDQ_1 & 0x0fff;
```

The Rx FDQ can be set up to have as many descriptors as needed based on the application. All of them can have the same parameters except the buffer pointer which can be set depending on the application.

- FFTC_RX_FDQ_2 packet descriptor setup

```
HOST_DESCRIPTOR *pdPtr;

/* this information is read by Rx Packet DMA to determine where the buffer is */
pdPtr -> original_buffer0_pointer = fft_output_buffer_core2_ptr;

pdPtr -> pkt_return_qmgr = (FFTC_RX_FDQ_2 >> 12) & 0x3;
pdPtr -> pkt_return_qnum = FFTC_RX_FDQ_2 & 0x0fff;
```

After AIF2 Rx PKTDMA fills in the fields in the packet descriptor, the output packet of AIF2 will look like:

```
Packet descriptor:
packet_id: 2
packet_type: 0
data_offset: 16
packet_length: (OFDM symbol length + CP length)*size of sample in bytes
source_tag_hi: 0
source_tag_lo: 1 for antenna carrier 1 and 2 or 2 for antenna carrier 3 and 4
dest_tag_hi: 0
dest_tag_lo: 0
extended_packet_info_block_present: 0
protocol_specifc_valid_word_count: 1
error_flags: 0
protocol_specific_flag: 0
ret_push_policy: 0
pkt_return_qmgr: (AIF2_RX_FDQ >> 12) & 0x3
pkt_return_qnum: AIF2_RX_FDQ & 0x0fff
protocol_specifc_word: AxC = {1, 2, 3, 4}, Symbol number within a frame/subframe,
ingress
payload: cyclic prefix and OFDM symbol
```

When this packet is presented to FFTC, FFTC will use either flow 1 or 2 as given in *src_tag_lo* field to decide which Rx FDQ to use, pop a descriptor from the FDQ, store the output packet to the buffer specified in the packet descriptor and direct the output packet to output queue specified by the flow. In another words, Rx FDQ, output buffer and output queue are all tied with the flow used. In this example, since the flows of FFTC_A are specified to have protocol specific data, the AIF2 protocol specific data will be passed through to the FFTC output. Assuming cyclic prefix removal is enabled, the FFTC output packet will look like:

```
Packet descriptor:
packet_id: 0
packet_type: 0
protocol_specific_data_location: 1
packet_length: OFDM symbol length*size of sample in bytes
source_tag_hi: 0
source_tag_lo: 0
dest_tag_hi: 0
dest_tag_lo: 0
extended_packet_info_block_present: 0
protocol_specifc_valid_word_count: 1
error_flags: 0
protocol_specific_flag: 0
ret_policy: 0
ret_push_policy: 0
pkt_return_qmgr: (FFTC_RX_FDQ_1(or 2) >> 12) & 0x3
pkt_return_qnum: FFTC_RX_FDQ_1(or 2) & 0x0fff
buffer0_length: OFDM symbol length*size of sample in bytes
buffer0_pointer: data buffer starting address
next_descriptor_pointer: NULL


Data buffer:
protocol_specifc_word: AxC = {1, 2, 3, 4}, Symbol number within a frame/subframe,
ingress
payload: FFT output data
```

Example 3: Different antenna data requires different FFTC settings with direct connection between AIF2 and FFTC

This example shows how to direct different antenna carriers to different input queues of FFTC by using another option of Rx PKTDMA overwrite capability. The FFTC can be configured based on input queue for different parameters. There are two antenna carriers used in this example, each antenna output goes to a separate input queue of FFTC.

Assume antenna carrier 1 and 2 of AIF2 are used, therefore AIF2 flow 1 and 2 are used. Antenna 1 carries 20MHz LTE data, and antenna 2 carries 10MHz LTE data. LTE 7.5KHz frequency shift needs to be done by FFTC. Since antenna 1 data goes to FFTC_A input queue 1, and Antenna 2 data goes to FFTC_A input queue 2, the frequency shift register for each input queue can be set accordingly based on the bandwidth. Two flows are used, FFTC_A flow 1 and 2, so that the output of each antenna can be stored and processed by different cores. Both FFTC_A output flows will use host mode packets with protocol specific data in the packet descriptor. The output of antenna carrier 1 of AIF2 will use FFTC_A flow 1, and the output of antenna carrier 2 of AIF2 will use FFTC_A flow 2. The output of FFTC_A flow 1 will be stored in the L2 memory of core 1 and processed by core 1. The output of FFTC_A flow 2 will be stored in the L2 memory of core 2 and processed by core 2.

- AIF2 flow 1 setup

```
FLOW_TBL *flowPtr;

/* there is PS info in the Rx packet */
flowPtr -> Reg_A.rx_psinfo_present = 1;

/* Rx packet is a monolithic packet, so the Rx FDQ packet descriptors need to be
monolithic packet type */
flowPtr -> Reg_A.rx_desc_type = 2;

/* payload starts at the offset of 16 bytes in the packet descriptor */
flowPtr -> Reg_A.rx_sop_offset = 16;

/* the output queue of this flow is FFTCA input queue 1 */
flowPtr -> Reg_A.rx_dest_qmgr = (FFTCA_INPUT_Q1 >> 12) & 0x3;
flowPtr -> Reg_A.rx_dest_qnum = FFTCA_INPUT_Q1 & 0x0fff;

/* rx packet descriptor src_tag_lo field is overwritten by the flow ID used in AIF2
which will be 1 in this case. Since the output queue of AIF2 is the input queue of
FFTC, src_tag_lo field will be used by FFTC as the flow ID for FFTC which is 1 */
flowPtr -> Reg_C.rx_src_tag_lo_sel = 2;

/* this flow uses AIF2_RX_FDQ as the FDQ */
flowPtr -> Reg_E.rx_fdq0_sz0_qmgr = (AIF2_RX_FDQ_1 >> 12) & 0x3;
flowPtr -> Reg_E.rx_fdq0_sz0_qnum = AIF2_RX_FDQ_1 & 0x0fff;
```

- AIF2 flow 2 setup

```
FLOW_TBL *flowPtr;

/* the output queue of this flow is FFTCA input queue 2 */
flowPtr -> Reg_A.rx_dest_qmgr = (FFTCA_INPUT_Q2 >> 12) & 0x3;
flowPtr -> Reg_A.rx_dest_qnum = FFTCA_INPUT_Q2 & 0x0fff;

/* rx packet descriptor src_tag_lo field is overwritten by the flow ID used in AIF2
which will be 2 in this case. Since the output queue of AIF2 is the input queue of
FFTC, src_tag_lo field will be used by FFTC as the flow ID for FFTC which is 1 */
flowPtr -> Reg_C.rx_src_tag_lo_sel = 2;

/* this flow uses AIF2_RX_FDQ as the FDQ */
flowPtr -> Reg_E.rx_fdq0_sz0_qmgr = (AIF2_RX_FDQ_2 >> 12) & 0x3;
flowPtr -> Reg_E.rx_fdq0_sz0_qnum = AIF2_RX_FDQ_2 & 0x0fff;
```

Other fields of the flow table can be cleared to 0. The key thing to note here is that by setting *rx_src_tag_lo_sel* in register C to 2, AIF2 Rx PKTDMA will use its flow ID, which is the PKTDMA channel number, to fill the *src_tag_lo* field in the output packet descriptor.

- AIF2_RX_FDQ_1 packet descriptor setup

```
MONOLITHIC_DESCRIPTOR *pdPtr;

/* this information is used by FFTC Tx Packet DMA and the value can be set to other
value which depends on the application */
pdPtr -> ret_push_policy = 0;

/* the packet descriptor will be recycled to AIF2_RX_FDQ_1 by FFTC Tx Packet DMA */
pdPtr -> pkt_return_qmgr = (AIF2_RX_FDQ_1 >> 12) & 0x3;
pdPtr -> pkt_return_qnum = AIF2_RX_FDQ_1 & 0x0fff;
```

- AIF2_RX_FDQ_2 packet descriptor setup

```
MONOLITHIC_DESCRIPTOR *pdPtr;

/* this information is used by FFTC Tx Packet DMA and the value can be set to other
value which depends on the application */
pdPtr -> ret_push_policy = 0;

/* the packet descriptor will be recycled to AIF2_RX_FDQ_2 by FFTC Tx Packet DMA */
pdPtr -> pkt_return_qmgr = (AIF2_RX_FDQ_2 >> 12) & 0x3;
pdPtr -> pkt_return_qnum = AIF2_RX_FDQ_2 & 0x0fff;
```

- FFTC flow 1 setup

```
FLOW_TBL *flowPtr;

/* there is PS info in the Rx packet, as we want the PS words of AIF2 to be carried
to the output of FFTC */
flowPtr -> Reg_A.rx_psinfo_present = 1;

/* Rx packet is a host packet, so the Rx FDQ packet descriptors need to be host
packet type */
flowPtr -> Reg_A.rx_desc_type = 0;

/* the PS words will be in the packet descriptor */
flowPtr -> Reg_A.rx_ps_location = 0;

/* the SOP offset is 0, so the first word of the data buffer is the payload */
flowPtr -> Reg_A.rx_sop_offset = 0;

/* the output queue of this flow is any desired output queue */
flowPtr -> Reg_A.rx_dest_qmgr = (FFTCA_OUTPUT_Q_1 >> 12) & 0x3;
flowPtr -> Reg_A.rx_dest_qnum = FFTCA_OUTPUT_Q_1 & 0x0fff;

/* this flow uses FFTC_RX_FDQ as the FDQ */
flowPtr -> Reg_E.rx_fdq0_sz0_qmgr = (FFTC_RX_FDQ_1 >> 12) & 0x3;
flowPtr -> Reg_E.rx_fdq0_sz0_qnum = FFTC_RX_FDQ_1 & 0x0fff;
```

- FFTC flow 2 setup

```
FLOW_TBL *flowPtr;

/* the output queue of this flow is any desired output queue */
flowPtr -> Reg_A.rx_dest_qmgr = (FFTCA_OUTPUT_Q_2 >> 12) & 0x3;
flowPtr -> Reg_A.rx_dest_qnum = FFTCA_OUTPUT_Q_2 & 0x0fff;

/* this flow uses FFTC_RX_FDQ as the FDQ */
flowPtr -> Reg_E.rx_fdq0_sz0_qmgr = (FFTC_RX_FDQ_2 >> 12) & 0x3;
flowPtr -> Reg_E.rx_fdq0_sz0_qnum = FFTC_RX_FDQ_2 & 0x0fff;
```

- FFTC_RX_FDQ_1 packet descriptor setup

```
HOST_DESCRIPTOR *pdPtr;

/* this information is read by Rx Packet DMA to determine if the buffer size is big
enough to hold the entire packet or not, if not, then another packet descriptor needs
to be popped from the 2nd FDQ */
pdPtr -> original_buffer0_length = symbol length*size of sample in bytes;
```

```
/* this information is read by Rx Packet DMA to determine where the buffer is */
pdPtr -> original_buffer0_pointer = fft_output_buffer_core1_ptr;

/* the following information is only needed if FFTC output is sent to another input
queue of Packet DMA peripheral and the value can be set to other which depends on
the application, otherwise it is optional if the recycling of FFTC output queue is
handled by host and host can choose what to do */
pdPtr -> ret_policy = 0;
pdPtr -> ret_push_policy = 0;
pdPtr -> pkt_return_qmgr = (FFTC_RX_FDQ_1 >> 12) & 0x3;
pdPtr -> pkt_return_qnum = FFTC_RX_FDQ_1 & 0x0fff;
```

- FFTC_RX_FDQ_2 packet descriptor setup

```
HOST_DESCRIPTOR *pdPtr;

/* this information is read by Rx Packet DMA to determine where the buffer is */
pdPtr -> original_buffer0_pointer = fft_output_buffer_core2_ptr;

pdPtr -> pkt_return_qmgr = (FFTC_RX_FDQ_2 >> 12) & 0x3;
pdPtr -> pkt_return_qnum = FFTC_RX_FDQ_2 & 0x0fff;
```

The FFTC_QUEUE_1_LTE_FREQUENCY_SHIFT_REGISTER needs to be set based on 20MHz to remove the 7.5KHz frequency shift. Assuming there is no initial phase needed, the value for the register should be 0x00010001.

The FFTC_QUEUE_2_LTE_FREQUENCY_SHIFT_REGISTER needs to be set based on 10MHz to remove the 7.5KHz frequency shift. Assuming there is no initial phase needed, the value for the register should be 0x00018001.

After AIF2 Rx PKTDMA fills in the fields in the packet descriptor, the output packet of AIF2 will look like:

```
Packet descriptor:
packet_id: 2
packet_type: 0
data_offset: 16
packet_length: (OFDM symbol length + CP length)*size of sample in bytes
source_tag_hi: 0
source_tag_lo: 1 for antenna carrier 1 or 2 for antenna carrier 2
dest_tag_hi: 0
dest_tag_lo: 0
extended_packet_info_block_present: 0
protocol_specifc_valid_word_count: 1
error_flags: 0
protocol_specific_flag: 0
ret_push_policy: 0
pkt_return_qmgr: (AIF2_RX_FDQ_1(or 2) >> 12) & 0x3
pkt_return_qnum: AIF2_RX_FDQ_1(or 2) & 0x0fff
protocol_specifc_word: AxC = 1 or 2, Symbol number within a frame/subframe, ingress
payload: cyclic prefix and OFDM symbol
```

When this packet is presented to FFTC, FFTC will use either flow 1 or 2 as specified in the *src_tag_lo* field to decide which FDQ to use, pop a descriptor from the FDQ, store the output packet to the buffer specified in the packet descriptor and direct the output packet to output queue specified by the flow. In another words, FDQ, output buffer and output queue are all tied with the flow used. In this example, since the flows of FFTC_A are specified to have protocol specific data, the AIF2 protocol specific data will be passed through to the FFTC output. Assuming cyclic prefix removal is enabled, the FFTC output packet will look like:

```
Packet descriptor:
packet_id: 0
packet_type: 0
protocol_specific_data_location: 0
packet_length: OFDM symbol length*size of sample in bytes
source_tag_hi: 0
source_tag_lo: 0
dest_tag_hi: 0
dest_tag_lo: 0
```

```
extended_packet_info_block_present: 0
protocol_specifc_valid_word_count: 1
error_flags: 0
protocol_specific_flag: 0
ret_policy: 0
ret_push_policy: 0
pkt_return_qmgr: (FFTC_RX_FDQ_1(or 2) >> 12) & 0x3
pkt_return_qnum: FFTC_RX_FDQ_1(or 2) & 0x0fff
buffer0_length: OFDM symbol length*size of sample in bytes
buffer0_pointer: data buffer starting address
next_descriptor_pointer: NULL
protocol_specifc_word: AxC = 1 or 2, Symbol number within a frame/subframe, ingress


Data buffer:
payload: FFT output data
```

6.3 Advanced Usage

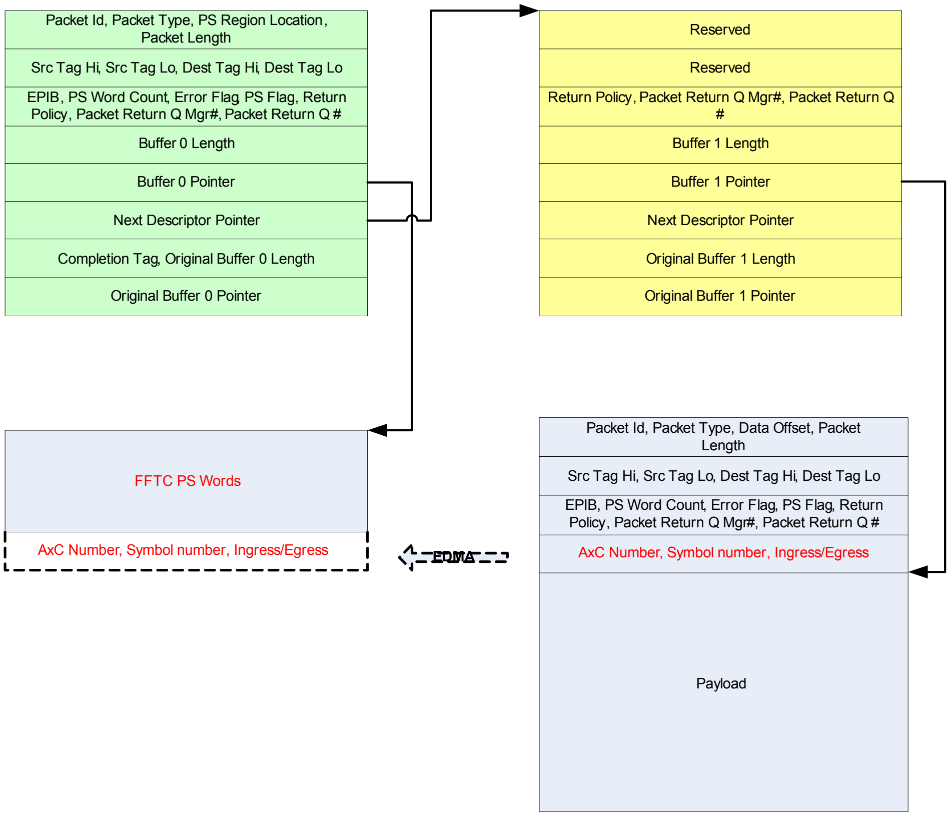
In the previous sections, the FFTC settings are the same for all the packets of the same antenna. In some use cases, it is desirable to have a packet by packet configuration for FFTC. Therefore the protocol specific words of FFTC need to be used to configure the FFTC on a packet by packet basis. An example of LTE PUSCH and PUCCH receiver is given in Section “6.4 Example LTE PUSCH and PUCCH Usage” on page 32. Since when AIF2 and FFTC are connected directly through a queue, no protocol specific words for FFTC can be added for the input packets. Therefore, AIF2 and FFTC can not be connected directly through a queue in this case. It is necessary to reconstruct the FFTC input packet.

6.3.1 Reconstructing the FFTC Packet

As mentioned above, in order to add FFTC protocol-specific words into the packet, it is necessary to reconstruct the packet before sending it to FFTC. The reconstruction is illustrated in Figure 8. The gray block in the lower right hand side is the monolithic packet descriptor of the AIF2 output, the green packet descriptor linked with yellow buffer descriptor is the reconstructed packet. The linked buffer feature supported by host packet is used to reconstruct the packet to avoid buffer copy.

The first buffer contains PS words for FFTC and is pointed by the packet descriptor shown in green background. The second buffer contains the payload which is part of the monolithic packet of the AIF2 output and is pointed by the buffer descriptor shown in yellow background. If the protocol specific words of AIF2 are desired at the output of FFTC, then they need to be copied to the end of the FFTC PS word buffer either by DSP or by EDMA because the PS words have to be contiguous.

Figure 8 Reconstruct FFTC Input Packet


Packet Id, Packet Type, PS Region Location, Packet Length
Src Tag Hi, Src Tag Lo, Dest Tag Hi, Dest Tag Lo
EPIB, PS Word Count, Error Flag, PS Flag, Return Policy, Packet Return Q Mgr#, Packet Return Q #
Buffer 0 Length
Buffer 0 Pointer
Next Descriptor Pointer
Completion Tag, Original Buffer 0 Length
Original Buffer 0 Pointer
Reserved
Reserved
Return Policy, Packet Return Q Mgr#, Packet Return Q #
Buffer 1 Length
Buffer 1 Pointer
Next Descriptor Pointer
Original Buffer 1 Length
Original Buffer 1 Pointer
FFTC PS Words
AxC Number, Symbol number, Ingress/Egress
EDMA
Packet Id, Packet Type, Data Offset, Packet Length
Src Tag Hi, Src Tag Lo, Dest Tag Hi, Dest Tag Lo
EPIB, PS Word Count, Error Flag, PS Flag, Return Policy, Packet Return Q Mgr#, Packet Return Q #
AxC Number, Symbol number, Ingress/Egress
Payload

Example 4: Configure FFTC PS words with non-direct connection between AIF2 and FFTC

This example shows how to reconstruct the packet as shown in Figure 8 to add PS words for FFTC.

- AIF2 flow setup

```
FLOW_TBL *flowPtr;

/* there is PS info in the Rx packet */
flowPtr -> Reg_A.rx_psinfo_present = 1;

/* Rx packet is a monolithic packet, so the Rx FDQ packet descriptors need to be
monolithic packet type */
flowPtr -> Reg_A.rx_desc_type = 2;

/* payload starts at the offset of 16 bytes in the packet descriptor */
flowPtr -> Reg_A.rx_sop_offset = 16;

/* the output queue is AIF2_OUTPUT_Q_x */
flowPtr -> Reg_A.rx_dest_qmgr = (AIF2_OUTPUT_Q_x >> 12) & 0x3;
flowPtr -> Reg_A.rx_dest_qnum = AIF2_OUTPUT_Q_x & 0x0fff;

/* this flow uses AIF2_RX_FDQ_x as the FDQ */
flowPtr -> Reg_E.rx_fdq0_sz0_qmgr = (AIF2_RX_FDQ_x >> 12) & 0x3;
flowPtr -> Reg_E.rx_fdq0_sz0_qnum = AIF2_RX_FDQ_x & 0x0fff;
```

- AIF2_RX_FDQ packet descriptor setup

```
MONOLITHIC_DESCRIPTOR *pdPtr;

/* this information is used by the recycling Packet DMA and the value can be set to
other value which depends on the application */
pdPtr -> ret_push_policy = 0;
```

```
/* the packet descriptor will be recycled to AIF2_RX_FDQ_x by the recycling Packet
DMA */
pdPtr -> pkt_return_qmgr = (AIF2_RX_FDQ_x >> 12) & 0x3;
pdPtr -> pkt_return_qnum = AIF2_RX_FDQ_x & 0x0fff;
```

If the output packets are to be recycled by some other PKTDMA automatically, then it is necessary to set the above fields in the packet descriptors in FDQ. However if the DSP is used to recycle the output queue, it is not mandatory to set those fields. If DSP intervention free recycling is done by EDMA, as explained later, it is not necessary to set the above fields in packet descriptors in Rx FDQ.

Each flow can be set up to use different Rx FDQs and output queues or share the same FDQ and output queue. If different queues are used for different antenna carriers, it is clear which packet belongs to which antenna carrier even if they can arrive out of order. This eliminates the need to copy the AIF2 PS words to the end of the FFTC PS word's buffer. However, as it will be explained later, each DSP intervention free mechanism has its own pros and cons.

Here is an example of how fields of the reconstructed packet descriptor and buffer descriptor need to be set.

- Reconstructed packet descriptor setup

```
HOST_DESCRIPTOR *pdPtr;

/* host packet is used */
pdPtr -> packet_id = 0;

/* protocol specific data is located in SOP buffer. This can also be set to 0 to
leave the PS words in the packet descriptor itself but the packet descriptor needs
to be configured big enough to hold all the PS words including the AIF2 PS word if
a copy is needed */
pdPtr -> ps_region_location = 1;

/* packet length in host packet doesn't include descriptor header or the protocol
specific words */
pdPtr -> packet_length  = (OFDM symbol length + CP length)*size of sample in bytes;

/* the flow used for FFTC can be set to any number between 0~7 depending on
application */
pdPtr -> src_tag_lo = FFTC flow for this packet;

/* the number of PS words in this example is set to 7*4=28 bytes, which include FFTC
control header, local configuration words, and AIF2 PS word */
pdPtr -> psv_word_count = 7;

/* this field indicates FFTC control header is present in protocol specific words */
pdPtr -> ps_flags  = 1;

/* this field sets up the return of the packet including buffer descriptor will be
returned as a whole still linked together to the tail of the queue */
pdPtr -> ret_policy = 0;
pdPtr -> ret_push_policy = 0;

/* this is needed for FFTC Tx Packet DMA to recycle the Tx packet */
pdPtr -> pkt_return_qmgr = (FFTC_TX_FDQ >> 12) & 0x3;
pdPtr -> pkt_return_qnum = FFTC_TX_FDQ & 0x0fff;

/* buffer 0 length is set to 0 as the buffer length doesn't include PS words and
buffer 0 only contains PS words with no payload */
pdPtr -> buffer_len = 0;

/* buffer 0 points to the PS words which include control header and local
configuration words */
pdPtr -> buffer_ptr = fft_ps_word_buffer;

pdPtr -> next_desc_ptr  = (Uint32)bdPtr;
```

- Reconstructed buffer descriptor setup

```
HOST_DESCRIPTOR *bdPtr;
```

```
/* buffer 1 length is set to the payload length as buffer length doesn't include PS
words */
bdPtr -> buffer_len = (OFDM symbol length + CP length)*size of sample in bytes;

/* buffer 1 points to the payload part in AIF2 output packet which is at the offset
of 16 bytes from the beginning of the packet descriptor */
bdPtr -> buffer_ptr = AIF2_output_PD_addr + 16;

bdPtr -> next_desc_ptr = NULL;
```

Since the input packet is reconstructed, the FFTC protocol specific word can be added to each packet, it is possible to assign flow, FFTC parameters and output queue on a packet by packet basis.

If DSP intervention free data path is desired, then the above descriptors can be prepared at initialization time and stored in the Tx FDQ. The number of descriptors to have in the Tx FDQ depends on the application. If DSP intervention is needed, the reconstruction can also be done at run time.

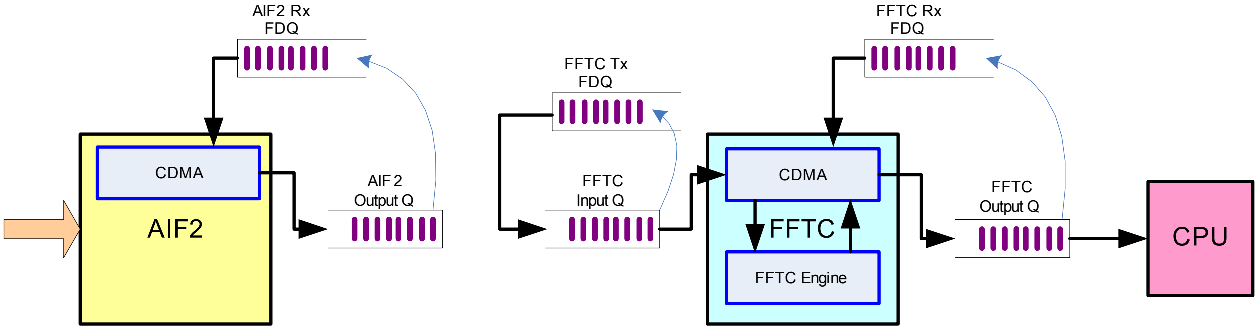
6.3.2 DSP Intervention Free Data Path

The solution to achieving DSP intervention free data path is to use the EDMA to copy the reconstructed packet descriptor from the FFTC Tx FDQ to the FFTC input queue, because popping is accomplished by reading the head packet pointer from and pushing is accomplished by writing the packet descriptor address to a specific address in the Queue Manager that corresponds to the queue. The EDMA transfer needs to be synchronized with the antenna packet arrival. There are several ways to generate the synchronization event. One way is to use the accumulator functionality of the QMSS to trigger the event when the desired number of antenna packets has arrived in the AIF2 output queue. The other way is to use AIF2 timer which can be set to align with the OFDM symbol timing with programmable offset.

6.3.2.1 Single Queue Data Path

If the AIF2 PS word is copied to the end of FFTC PS words buffer, then it is not necessary to have different Rx FDQs or output queues per antenna carrier. The queue set up is shown in Figure 9.

Figure 9 Single Queue DSP Intervention Data Path



If the accumulator is used to generate the synchronization event, it can be generated when all antenna data of one OFDM symbol are received. Upon the synchronization event trigger, the following steps need be done by the EDMA either by chaining different EDMA channels or by self-chaining and linking to the same EDMA channel:

1. Copy the AIF2 PS words to the end of the FFTC PS word buffer

2. En-queue FFTC by reading from the FFTC Tx FDQ and writing to the input queue of FFTC
3. Recycle the AIF2 output queue by reading the AIF2 output queue and writing to the AIF2 Rx FDQ
4. Clear CP_INTC interrupt pending register by writing to system interrupt status indexed clear register (STATUS_CLR_INDEX_REG)
5. Clear QMSS interrupt distributor (INTD) event of the corresponding accumulator queue by writing to the end of interrupt register (EOI)

If AIF2 timer is used to generate the synchronization event, then the trigger can be set to happen with all antenna packets of one OFDM symbol are received. Since AIF2 timer events are routed directly to TPCC1 as primary events, there is no need to clear the interrupt pending registers of CP_INTC and INTD. Therefore only the first 3 steps are needed.

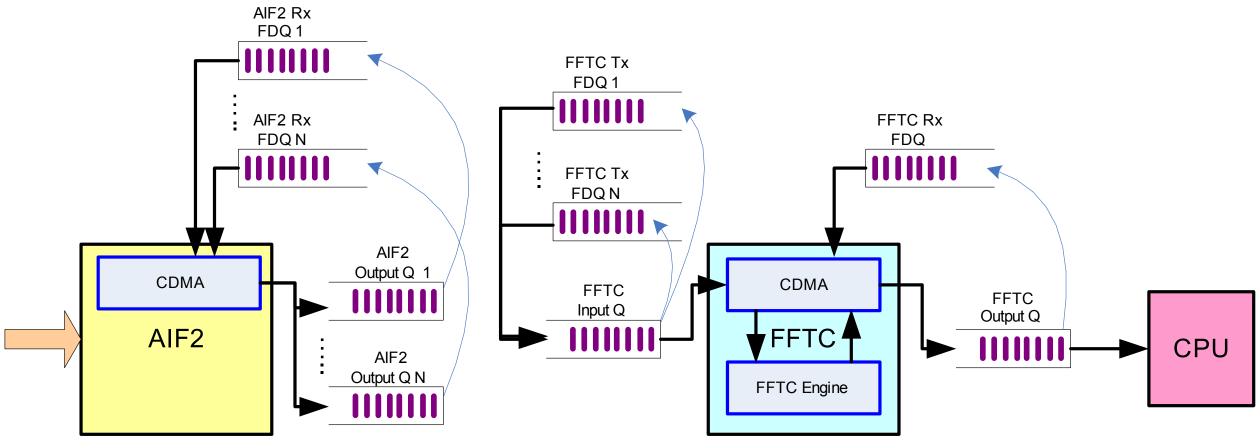
Since the reconstructed FFTC Tx packet will be recycled by FFTC Tx PKTDMA automatically while the AIF2 output packet which contains the antenna data will be recycled by EDMA, the timing of them can be different. In order not to starve AIF2 Rx FDQ or overwrite the data buffer which has not been taken by FFTC yet, enough packet descriptors need to be allocated in the AIF2 FDQ which depends on the application. For example, for 20MHz LTE application with 8 receive antennas, if one FFTC is dedicated for UL front end IFFT processing, then a symbol-based ping-pong descriptor allocation for AIF2 Rx FDQ will guarantee no starvation and no overwriting because FFTC can process 8 size 2048 IFFT in less than a OFDM symbol time.

It is important to note that the FFTC output queue will be in the same order as the AIF2 output queue which is not guaranteed to be in antenna order. However as AxC number is available in the PS words of FFTC output packets, the application can do the processing accordingly based on AxC number.

6.3.2.2 Multi-Queue Data Path

If the AIF2 PS word is not copied to the end of FFTC PS word's buffer, then it is necessary to have separate AIF2 Rx FDQs and output queues for different antenna carriers, as well as separate FFTC Tx FDQs so that they have a one to one correspondence. For example, for antenna carrier 1, the reconstructed packet in FFTC Tx FDQ 1 would point to the payload of the descriptor of AIF2 Rx FDQ 1 which will appear in the AIF2 output queue 1 when the packet is received by AIF2. The queue set up is shown in Figure 10.

Figure 10 Multi-Queue DSP Intervention Free Path



If the accumulator is used to generate the synchronization event, then multiple high priority queues are needed as there are multiple AIF2 output queues. Since each accumulator queue corresponds to one antenna carrier, the accumulator event can be triggered when the packet of the antenna carrier for one ODFM symbol is received. As a result, multiple EDMA channels need to be used with one EDMA channel triggered by one accumulator event. Steps 2 to 5 above need to be done with each EDMA channel only handling one antenna. It is important to note here that although it is known at the output of FFTC which output packet belongs to which antenna, the order of the multiple antennas within a symbol is not guaranteed. So it is necessary to use some field that can be defined by the application to indicate to which antenna the packet belongs. For example, *source_tag_hi*, *dest_tag_hi,* and *dest_tag_lo*. FFTC will pass those fields from the Tx packet to Rx packet if the flow table is configured accordingly.

If AIF2 timer is used to generate the synchronization event, then the trigger can be set to happen when all antenna packets of one OFDM symbol are received. Therefore only one EDMA channel is needed to do the FFTC en-queue transfer traversing all the FFTC Tx FDQs. Similarly, one single EDMA channel is enough to do the recycling transfer for AIF2 output queue. It is important to note that in order to use a single EDMA channel to do the FFTC en-queue or AIF2 output queue recycling transfer, the multiple queues used for FFTC Tx queues, AIF2 output queues and AIF2 Rx FDQs need to be continuous or equally spaced queues so that EDMA can be programmed to jump from one queue to another queue. Upon the AIF2 timer event, steps 2 and 3 above need to be done. If the AIF2 timer is set to generate the event when all antenna packets of one OFDM symbol are received, the EDMA can en-queue them to FFTC in the antenna order so that it is known at the output which packet belongs to which antenna.

6.3.2.3 Comparisons

When accumulator events are used as the synchronization events, subject to the latency of accumulator events, the EDMA can be triggered as soon as the packets arrive in the AIF2 output queue. When AIF2 timer is used as the synchronization event, since the packet arrival time may be jittery due to various factors, some buffer time needs to be added between antenna packet arrival and the AIF2 timer trigger point to guarantee the data has arrived before the trigger is generated.

When accumulator events are used as the synchronization events, if the multi-queue data path is used, then more system resources are needed including high priority accumulator queues and EDMA channels. When AIF2 timer is used as the synchronization event, regular queues can be used and a single EDMA channel is enough.

When accumulator events are used as the synchronization events, it is necessary to clear the interrupt pending registers of CP_INTC and INTD. When AIF2 timer is used as the synchronization event, there is no need to clear the interrupt pending registers of CP_INTC and INTD.

When AIF2 timer is used as synchronization event, since the AIF2 output queue status is irrelevant provided that AIF2 timer event is offset to guarantee that it is only generated when the packets have arrived, the AIF2 output queue can the same as AIF2 Rx FDQ. Therefore the recycling step for the AIF2 output queue can be saved.

Other options and different variants of the methods described in this section can also be used to achieve the seamless connection between AIF2 and FFTC.

Since the descriptors in the Tx completion queue do not contain the descriptor size information, when DSP intervention free method is used, the Tx completion queue can not be used directly as the source queue, namely Tx FDQ here, for EDMA transfer. There are two ways to work around this issue:

1. Use two separate queues for the Tx FDQ, and the Tx completion queue and DSP needs to pop descriptors from the Tx completion queue and push descriptors with size information to the Tx FDQ. This can be done in post FFTC processing at an appropriate time depending on the application. EDMA will still read from the Tx FDQ for DSP intervention free connection between AIF2 and FFTC.
2. Use an array to replace the Tx FDQ. The array will contain the descriptor addresses with size information in the 4 LSB. EDMA will always read from the array and push to the Tx queue. Although the packets will still be recycled to Tx completion queue, they will not be read from there. The Tx completion queue needs to be flushed appropriately so that no descriptors will be present simultaneously in both the Tx queue and the Tx completion queue. Since pushing a NULL descriptor flushes a queue, one way to flush the Tx completion queue is to do an extra EDMA transfer of a NULL descriptor to the Tx completion queue at an appropriate time depending on the application. Using an array provides a DSP intervention-free connection between AIF2 and FFTC without post FFTC handling of the queues, however extra care needs to be taken when using this method as using an array assumes fixed packet order at all times.

6.4 Example LTE PUSCH and PUCCH Usage

Figure 11 Example Usage of FFTC for LTE PUSCH and PUCCH

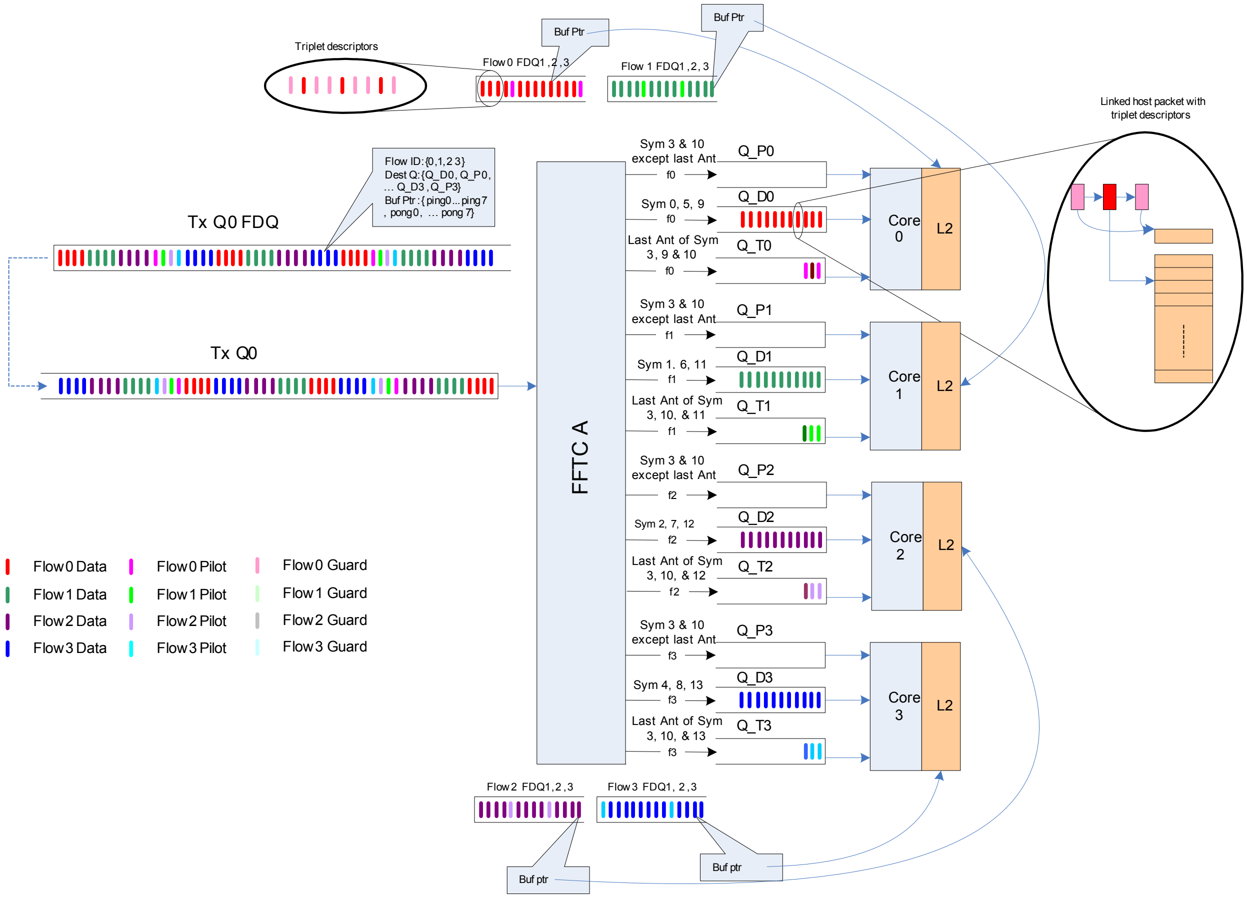


Figure 11 shows a complete picture of an example usage of LTE PUSCH and PUCCH around FFTC. It uses the single queue data path idea as described in section "6.3 Advanced Usage" on page 25 to reconstruct the packets so as to connect AIF2 and FFTC on inbound traffic.

As AIF2 part is the same as Figure 9, it is omitted in the picture. For simplicity of drawing, it shows a four receive antenna system. The same design idea can be applied to other system configurations, for example 2 receive antennas or 8 receive antennas. The load balancing, system partitioning, scheduling and memory usage are all achieved by using PKTDMA with different settings of FFTC PS words on a packet by packet basis.

There is one Tx FDQ for FFTC and one input queue of FFTC is used. The partition and scheduling information is contained in the Tx FDQ descriptors by using the destination queue in the local configuration words of FFTC. Each core will only process packets contained in 3 queues assigned to it. The partition is a round robin based method. For data symbols, it is symbol based round robin such that each core processes all antennas of 3 data symbols each. For pilot symbol, it is antenna based round robin so that ¼ of antennas of each pilot symbol is processed on each core.

For example, core 0 processes data symbol 0, 5, and 9, and antenna 0 of two pilot symbols. This is achieved by setting different output queues based on symbol or antenna number. For example, data symbol 0, 5 and 9 go to output queue Q_D0. Since the pilot symbols need to be processed first, data symbols and pilot symbols are routed to separate output queues so that there is no need to pop data symbols in order to get to pilot symbols. This is shown in the picture by Q_Px and Q_Dx.

Since the DSP may need to do other processing, it is preferred to interrupt the DSP when certain events happen instead of letting the DSP poll the output queue status. In addition, it is also preferred to only interrupt the DSP which will process the job. Therefore an interrupt queue is used per core which is shown in the picture by Q_Tx. In this example, three interrupts will be generated per subframe, once when pilot symbol 3 is done by FFTC, once when pilot symbol 10 is done by FFTC and once when the last data symbol is done by FFTC. Since the fastest way to trigger interrupt is by using the QPEND signal, all four Q_Tx queues are queues supporting QPEND event. However QPEND event is triggered only based on non-empty status. So only the output of FFTC of the last antenna of each pilot symbol goes to the Q_Tx queue, and the output of FFTC of the last antenna of the last data symbol goes to Q_Tx queue.

For example, last antenna of data symbol 9, 11, 12, and 13 goes to Q_T0, Q_T1, Q_T2 and Q_T3 respectively. Since there is only 1 antenna per pilot per core in this example, this pilot symbol output also goes to Q_Tx respectively and Q_Px is not used. If there is more than 1 pilot antenna processed per core, then except for the last antenna per pilot symbol, the FFTC output goes to Q_Px while the last antenna per pilot symbol goes to Q_Tx. Since Q_Tx queues are the queues with QPEND event, it can generate interrupts to the DSP to trigger processing. Channel estimation is triggered when the last antenna of each pilot symbol is processed by FFTC, data symbol processing can start immediately after channel estimation is done without having to wait the last data symbol per core. When the last antenna of last data symbol has arrived in the Q_Tx queue, DSP will be notified by interrupt.

Four flows of FFTC are used with each flow assigned to each destination core. Because Rx FDQ is associated with a flow, then each core can prepare its own Rx FDQ with local L2 memory addresses for FFTC output. Each flow uses 3 Rx FDQs with them all being the same queue. And the descriptors in the Rx FDQ have a triplet pattern. The first one has a buffer big enough to hold the left guard tones, the second one has a buffer big enough to hold the desired data tones, and the third one has a buffer big enough to hold the right guard tones. Since the guard tones are discarded after FFTC processing, it is not necessary to allocate separate memory for different symbols. They can all point to the same buffer to save memory usage.

For each FFT packet, the Rx PKTDMA will need to pop Rx FDQ 1, 2 and 3 to put the entire output packet in. Because Rx FDQ 1, 2 and 3 are set to the same queue with the triplet descriptor pattern, the output of the packet will be a linked host packet with three descriptors. It is important to note that the triplet descriptors in the Rx FDQs are 3 separate packet descriptors, while they become one linked host packet in the output queue in which there is only the first descriptor of the triplet descriptors.

The FFTC Tx FDQ can be prepared at initialization time with one or multiple subframes worth of descriptors, each with its own PS words set to achieve the load balancing, partitioning, scheduling and memory usage model described above. This doesn't add significant overhead in memory usage as host packet mode is used and only packet descriptors are allocated for one or multiple subframes. For AIF2 output, only symbol based ping-pong descriptors need to be allocated. The FFTC Tx FDQ needs to be set up such that the packet descriptors point to the correct payload buffers of the AIF2 output ping-pong descriptors.

7 Connecting FFTC and AIF2 for Outbound Antenna Traffic

On the outbound traffic, since FFTC is capable of outputting packets with protocol specific words for AIF2 by using the pass through field, FFTC can be connected to AIF2 directly via queues without any loss of flexibility.

On the outbound path, AIF2 requires that each antenna carrier has a separate input queue. If the number of transmit antennas to support is not more than available FFTC flows, it is acceptable to use different FFTC flows to direct different antenna symbols to different AIF2 input queues. Otherwise the destination queue can be specified by using the local configuration words of protocol specific words for FFTC.

Example 5: Using FFTC PS words to connect FFTC and AIF2 directly

In this example, FFTC protocol specific words are used to control the output queue for each packet, and to fill in the protocol specific words in the output packet for AIF2.

- FFTC input packet

```
HOST_DESCRIPTOR *pdPtr;

/* host packet is used */
pdPtr -> packet_id = 0;

/* protocol specific data is located in SOP buffer */
pdPtr -> ps_region_location = 1;

/* packet length in host packet doesn't include descriptor header or the protocol
specific words */
pdPtr -> packet_length  = (number of data subcarriers)*size of sample in bytes;

/* the flow used for FFTC can be set to any number between 0~7 depending on the
application, this does not necessarily have a one to one correspondence with the
output queue as output queue can be specified in the FFTC PS words */
pdPtr -> src_tag_lo = FFTC flow for this packet;

/* the number of PS words in this example is set to 7*4=28 bytes, which include FFTC
control header, local configuration words, and AIF2 PS word */
pdPtr -> psv_word_count = 7;

/* this field indicates FFTC control header is present in protocol specific words */
pdPtr -> ps_flags  = 1;

/* this field sets up the return of the packet including buffer descriptor will be
returned as a whole still linked together to the tail of the queue */
pdPtr -> ret_policy = 0;
pdPtr -> ret_push_policy = 0;

/* this is needed for FFTC Tx Packet DMA to recycle the Tx packet */
pdPtr -> pkt_return_qmgr = (FFTC_TX_FDQ >> 12) & 0x3;
pdPtr -> pkt_return_qnum = FFTC_TX_FDQ & 0x0fff;

/* buffer 0 length is set to the same as packet length */
pdPtr -> buffer_len = (number of data subcarriers)*size of sample in bytes;

/* buffer 0 points to input data buffer */
pdPtr -> buffer_ptr = fft_input_data_buffer
pdPtr -> next_desc_ptr  = NULL;
```

Depending on the application, the protocol specific words can be set to choose which destination queue each packet is sent and to fill in the AIF2 protocol specific words including AxC number, symbol number and egress. The control header of the protocol specific words should indicate the pass through fields are present so that FFTC will pass the protocol specific words set for AIF2 to output packets.

Since in this example the Tx packet is configured to contain only the useful data subcarriers, FFTC needs to be configured to pad 0s. If the input data is arranged in the subcarrier order, then FFTC needs to be configured to do the input left / right shift. In addition, as FFTC output is given to AIF2 directly, FFTC needs to be configured to add cyclic prefix.

- FFTC flow setup

```
FLOW_TBL *flowPtr;

/* there is PS info in the Rx packet, as we want the PS words of AIF2 to be carried
to the output of FFTC */
flowPtr -> Reg_A.rx_psinfo_present = 1;

/* Rx packet is a monolithic packet, so the Rx FDQ packet descriptors need to be
monolithic packet type */
flowPtr -> Reg_A.rx_desc_type = 2;

/* payload offset is 16bytes in the output monolithic packet */
flowPtr -> Reg_A.rx_sop_offset = 16;

/* this flow uses FFTC_RX_FDQ as the FDQ */
flowPtr -> Reg_E.rx_fdq0_sz0_qmgr = (FFTC_RX_FDQ >> 12) & 0x3;
flowPtr -> Reg_E.rx_fdq0_sz0_qnum = FFTC_RX_FDQ & 0x0fff;
```

- FFTC_RX_FDQ packet descriptor setup

```
MONOLITHIC_DESCRIPTOR *pdPtr;

/* the following information is used by AIF2 Tx Packet DMA to recycle the descriptor
*/
pdPtr -> ret_push_policy = 0;
pdPtr -> pkt_return_qmgr = (FFTC_RX_FDQ >> 12) & 0x3;
pdPtr -> pkt_return_qnum = FFTC_RX_FDQ & 0x0fff;
```

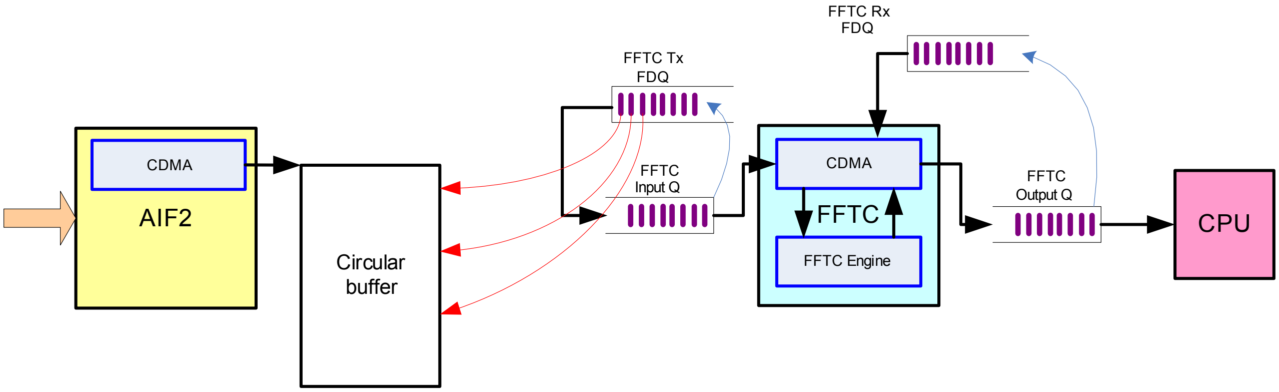
- The output packet of the FFTC looks like:

```
Packet descriptor:
packet_id: 2
packet_type: 0
data_offset: 16
packet_length: (OFDM symbol length + CP length)*size of sample in bytes
source_tag_hi: 0
source_tag_lo: 0
dest_tag_hi: 0
dest_tag_lo: 0
extended_packet_info_block_present: 0
protocol_specifc_valid_word_count: 1
error_flags: 0
protocol_specific_flag: 0
ret_push_policy: 0
pkt_return_qmgr: (FFTC_RX_FDQ) >> 12) & 0x3
pkt_return_qnum: FFTC_RX_FDQ & 0x0fff
protocol_specifc_word: AxC number, Symbol number within a frame/subframe, egress
payload: cyclic prefix and symbol
```

8 Special Notes on Using AIF2 DIO Mode

Other than packet mode, AIF2 also supports DIO mode. DIO mode is a simple circular buffer write to memory or read from memory, so it doesn't have the concept of symbols or packets. The advantage of using DIO mode is that the output data buffer can be contiguous.

Figure 12 Constructing Packets in DIO Mode



When DIO mode is used, as there is no queue involved, packets need to be constructed before sending to FFTC. This is shown in Figure 12 with the red arrows indicating the buffer pointer to the circular buffer for each packet. If there is no memory usage limitation, it is desirable to keep the entire subframe of antenna output data or at least one whole slot of the antenna data as it is simpler to calculate pointers for reconstructing packets. However in most cases, it is not possible to store an entire subframe or one whole slot of antenna data.

As DIO is a simple read / write circular buffer, it cannot be configured to handle different lengths of LTE symbols. So when the output buffer size can only be a fraction of a slot, the data offset per OFDM symbol or per PRACH packet needs to be calculated carefully depending on the buffer size chosen. It is better to choose the buffer size to be divisible by the number of samples in a subframe or in a slot so that the pointers align on a subframe or a slot basis.

When DIO mode is used, as there is no queue involved, AIF2 timer can be used to generate the synchronizing event to trigger the EDMA transfer to do the en-queue for FFTC.

9 References

C66x CorePac User Guide	SPRUGW0
Fast Fourier Transform Coprocessor (FFTC) for KeyStone Devices User Guide	SPRUGS2

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Communications and Telecom	www.ti.com/communications
DSP	dsp.ti.com	Computers and Peripherals	www.ti.com/computers
Clocks and Timers	www.ti.com/clocks	Consumer Electronics	www.ti.com/consumer-apps
Interface	interface.ti.com	Energy	www.ti.com/energy
Logic	logic.ti.com	Industrial	www.ti.com/industrial
Power Mgmt	power.ti.com	Medical	www.ti.com/medical
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video and Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless-apps

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265