Using DMA with Framework Components for ‘C64x+

Murat Karaorman, Gunjan Dang, Ezell Young *Software Development Organization*

ABSTRACT

This application note describes the standard DMA software abstractions and interfaces for TMS320 DSP Algorithm Standard (xDAIS) compliant algorithms designed for the ‘C64x+ EDMA3 controller using DMA Framework Components utilities. The DMA Framework Component utilities described in this document include a standard DMA resource specification and negotiation protocol (IDMA3), a DMA Resource Manager (DMAN3), and a functional DMA interface and library (ACPY3) compliant with IDMA3. This document provides both reference information and guidelines for producers and consumers of IDMA3-compliant algorithms and components.

Contents

1 Introduction 3
2 Fundamental Framework Components DMA Concepts 7
2.1 Logical DMA Channels & DMA Handles 7
2.2 Scratch vs. Persistent DMA Resources 7
2.3 DMA Resource Management using DMAN3 7
2.4 DMA Transfer Submission and Synchronization using ACPY3 8
2.4.1 FIFO Ordering of DMA Transfers and Linked DMA Transfers 8
2.4.2 Channel Privacy and Synchronization 8
2.4.3 DMA Transfer Configuration Settings 9
3 IDMA3: Standard Interface for Negotiating DMA Resources 11
3.1 IDMA3 versus IDMA2 11
3.2 IDMA3 Interface Definition 12
3.2.1 DMA Channel Descriptor: IDMA3_ChannelRec 13
3.2.2 IDMA3 Functions: IDMA3_Fxns 15
3.2.3 IDMA3 Object and Handle Structures: IDMA3_Obj 15
3.2.4 IDMA3 Protocol Object for Channel Environment Memory Management 18
3.2.5 IDMA3 Enumeration Type Documentation 19
4 DMAN3: ‘C64x+ DMA Resource Manager 20
4.1 Using DMAN3 for Algorithm Integration 20
4.2 DMAN3 Configuration 21
4.2.1 Introduction to Configuration Options 21
4.2.2 Configuration Parameters 22
4.2.3 DMAN3 Configuration Examples 26
4.2.4 Configuring DMAN3 Without Using RTSC 27
4.2.5 Configuring DMAN3 Using RTSC Tooling 28
4.3 DMAN3 Functions 29
4.3.1 DMAN3_grantDmaChannels 29
4.3.2 DMAN3_exit 31
4.3.3 DMAN3_init 31
4.3.4 DMAN3_releaseDmaChannels 31
4.3.5 DMAN3_createChannels 32
4.3.6 DMAN3_freeChannels 33

5 ACPY3: Functional DMA Abstraction Layer 34
5.1 ACPY3 Functions and Comparison to ACPY2 34
5.2 ACPY3 Interface 34
5.2.1 Logical Channel Configuration Parameters 34
5.2.2 ACPY3_configure 36
5.2.3 ACPY3_fastConfigure16b 37
5.2.4 ACPY3_fastConfigure32b 38
5.2.5 ACPY3_start 39
5.2.6 ACPY3_wait 39
5.2.7 ACPY3_waitLinked 40
5.2.8 ACPY3_complete 40
5.2.9 ACPY3_completeLinked 41
5.2.10 ACPY3_activate 41
5.2.11 ACPY3_deactivate 42
5.2.12 ACPY3_init 42
5.2.13 ACPY3_exit 42
5.2.14 ACPY3_setFinal 43
6 Cache Coherency Issues for Algorithm Consumers 44
7 For Algorithm Producers: Creating Algorithms that Use DMA 45
7.1 IDMA3 and ACPY3 Related Changes that affect the Algorithm Developers 46
7.2 Rules and Guidelines Summary 46
7.3 Implementing the IDMA3 Interface 47
7.4 Configuring Logical Channels and DMA Transfers 49
7.4.1 Performance Considerations 49
7.5 Scheduling Asynchronous DMA Transfers on Logical Channels 50
7.5.1 Alignment Issues Using ACPY3_start 50
7.6 Synchronizing and Serializing DMA Transfers 51
7.7 Cache Coherency Issues for Algorithm Producers 51
8 The Fast Copy (FCPY) Algorithm Example 52
8.1 IFCPY_Interface Functions 53
8.1.1 Instance Heap Memory Requirements 53
8.1.2 The Use of IDMA3 and ACPY3 Interfaces 53
9 The fastcopytest Example 54
10 References 55
Appendix A. Code for the fastcopytest Example 56
Appendix B: Code for FCPY_TI Algorithm 65
ifcpy.h 65
fcpy_ti.h 66
fcpy_ti_priv.h 67
fcpy_ti_ialg.c 69
fcpy_ti_idma3.c 71
fcpy_ti_idmavt.c 72
fcpy_ti_ifcpy.c 73
fcpy_ti_ialg.c 76

Figures

Figure 1. Client Application and Algorithm Interaction with DMA Resources 4
Figure 2. DMA Transfer Block 10
Figure 3. IDMA3 Function Calling Sequence 12

Figure 4. Algorithm Implementing IALG and IDMA3 Interfaces and Application Using Framework Components 13
Figure 5. IDMA3 Logical Channels 16
Figure 6. DMA Read Access Coherency Problem 44
Figure 7. DMA Write Access Coherency Problem 44
Figure 8. Cache Line Boundaries and the L2 Cache 45
Figure 9. Cache Line Boundaries and the L2 Cache 51
Figure 10. Illustration of FCPY doCopy operation 52
Figure 11. Dependencies in the fastcopytest Example 54

Tables

Table 1. IDMA3 Functions 4
Table 2. IDMA3 Channel Request Descriptor (IDMA3_ChannelRec) 5
Table 3. IDMA3 Channel (IDMA3_Obj) 5
Table 4. IDMA3 Protocol Object (IDMA3_ProtocolObj) 5
Table 5. DMAN3 Functions 6
Table 6. ACPY3 Functions 6
Table 7. ACPY3_Params Structure Fields 6

1 Introduction

The direct memory access (DMA) controller performs asynchronously scheduled data transfers between memory regions without the intervention of the CPU. The parallel operation of the DMA with the execution of the CPU relieves the CPU of the burden of these data transfers. This allows a system to achieve greater throughput.

Algorithms and client applications may want to take advantage of the DMA to overlap data movement with CPU processing. However, the TMS320 DSP Algorithm Standard (also known as xDAIS) does not allow compliant algorithms to *directly* access or control any hardware peripherals, including the DMA. All system DMA resources must be controlled by the client application.

The new Framework Components DMA utilities allow xDAIS algorithms and client applications to utilize DMA resources by providing standard DMA software abstractions and interfaces. Framework Components now includes the following DMA modules and interfaces:

- **IDMA3.** This is the standard interface to algorithms for DMA resource specification and negotiation protocols. This interface allows the client application to query and provide the algorithm its requested DMA resources.
- **DMAN3.** This is the DMA resource manager. It is responsible for managing and granting DMA resources to algorithms and applications based on the IDMA3 interface.
- **ACPY3.** This is the functional DMA interface and library. The ACPY3 interface describes a comprehensive list of DMA operations that an algorithm can perform on the logical DMA channels acquired through the IDMA3 protocol. These functions are implemented as part of the client application and are called by the algorithm.

Figure 1 shows which modules are implemented by client application frameworks and which are implemented by algorithms or components. Arrows indicate which modules use other modules.

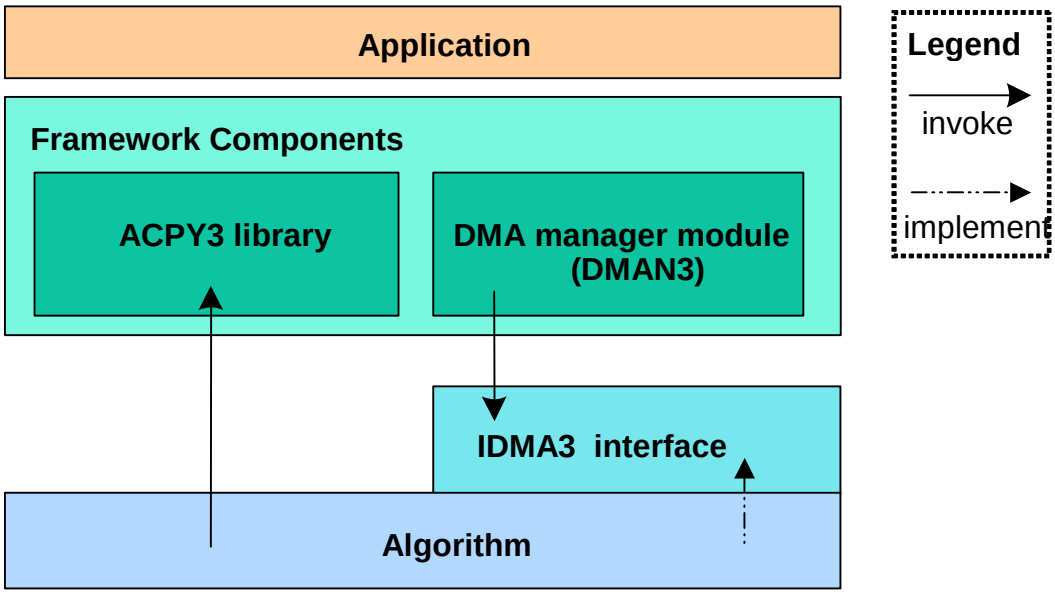


Figure 1. Client Application and Algorithm Interaction with DMA Resources

Client applications use the algorithm's IDMA3 interface to query the algorithm's DMA resource requirements and grant the algorithm logical DMA resources via handles. Each granted handle provides the algorithm a uniform, private logical DMA channel abstraction. Algorithms, upon getting provisioned by the framework with their DMA resource needs, may call ACPY3 functions to schedule DMA transfers on the logical DMA channels. Alternatively, algorithms may provide their own DMA functions to program the physical DMA resources acquired through the IDMA3 protocol.

The basic ideas and objectives described in the "Use of the DMA Resource" chapter of *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352) apply to the design, implementation and use of the ACPY3 and IDMA3 interfaces. Collectively, IDMA3, DMAN3, and ACPY3 provide a flexible and efficient model that greatly simplifies the management of system DMA resources and services by the client application. They also provide a simple and powerful mechanism for algorithms to configure and access DMA services.

The following tables summarize the API functions and structures used by the IDMA3, ACPY3, and DMAN3 interfaces.

Table 1. IDMA3 Functions

Functions	Description
dmaChangeChannels()	Called by an application whenever logical channels are moved at run-time.
dmaGetChannelCnt()	Called by an application to query an algorithm about its number of logical DMA channel requests.
dmaGetChannels()	Called by an application to query an algorithm about its DMA channel requests at initialization time, or to get the current channel holdings.
dmaInit()	Called by an application to grant DMA handle(s) to the algorithm at initialization.

Table 2. IDMA3 Channel Request Descriptor (IDMA3_ChannelRec)

Structure Fields	Description
Handle	Handle to logical DMA channel
numTransfers	Number of DMA transfers that are submitted using this logical channel handle. Single (==1) or Linked (>= 2)
numWaits	Number of individual transfers that can be waited in a linked start. (1 for single transfers or to wait for all transfers to complete)
priority	Relative priority recommendation {Urgent, High, Medium, Low}
protocol	Optional protocol handle for allocating channel environment (“env”) memory and calling custom channel initialization function.
persistent	When persistent is set to TRUE, the granted physical EDMA resources (PaRAMs and TCCs) are for exclusive use by this channel. They cannot be shared with any other IDMA3 channel.

Table 3. IDMA3 Channel (IDMA3_Obj)

Structure Fields	Description
numTccs	The number of TCCs allocated to this channel.
numPaRams	The number of PaRAM entries allocated to this channel.
tccTable	Array of TCCs assigned to this channel
paRamAddr	PaRAMs assigned to channel
qdmaChan	Physical QDMA Channel assigned to handle
transferPending	Run-time channel state that must be maintained by channel owner. Must be set to true when a new transfer is started on this channel. Must be set to false when a wait/sync operation is performed on this channel.
env	When the channel IDMA3_Protocol requires it, “env” points to the private channel memory allocated for the channel by the framework DMA resource manager.
protocol	The IDMA3_Protocol handle used by the DMA manager to determine memory requirements for the “env”.
persistent	Indicates if the channel has been allocated with persistent property.

Table 4. IDMA3 Protocol Object (IDMA3_ProtocolObj)

Structure Fields	Description
name	The name of the optional custom protocol that will be used by the algorithm requesting the IDMA3 channel resources. This object contains function pointers to be used when determining channel “env” memory requirements and for custom channel initialization and finalization.
getEnvMemRec	When not NULL, points to the function called by the DMA manager to query the IDMA3 protocol's memory requirements for the IDMA3 channel’s environment (“env”) pointer.
initHandle	When not NULL, points to the function to be called by DMA manager after allocation of the “env” memory to perform custom IDMA3 protocol initialization of channel state.
deInitHandle	When not NULL, points to the function to be called by DMA manager when a channel is freed to perform any custom IDMA3 protocol de-initialization.

Table 5. DMAN3 Functions

Functions	Description
DMAN3_grantDmaChannels	Grant logical channel resources to one of more algorithm instances sharing a common groupId.
DMAN_exit ()	Finalization method of the DMAN module.
DMAN3_init ()	Initialize the DMAN3 module.
DMAN3_releaseDmaChannels	Remove logical channel resources from one or more algorithm instances.
DMAN3_createChannels	Allocate DMA resources and initialize memory for one or more logical DMA channels.
DMAN3_freeChannels	Free DMA resources and memory allocated to one or more logical DMA channel.

Table 6. ACPY3 Functions

Functions	Description
ACPY3_activate	Activates given channel. Take over shared resources prior to use.
ACPY3_deactivate	Deactivates given channel. Give back shared resources at the end of use.
ACPY3_complete	Check if the data transfers on a specific logical channel have completed
ACPY3_completeLinked	Check if an individual transfer on a specific logical channel have completed
ACPY3_configure	Configure a logical channel
ACPY3_exit	Free resources used by the ACPY3 module ***[FRAMEWORK API]***
ACPY3_setFinal	Dynamically change the number of transfers in a sequence of linked transfers. Sets given transferNo as the last in a sequence of linked transfers
ACPY3_init	Initialize the ACPY3 module ***[FRAMEWORK API]***
ACPY3_fastConfigure16b	Modify a single (16-bit) parameter of the logical DMA Channel
ACPY3_fastConfigure32b	Modify a single (32-bit) parameter of the logical DMA Channel
ACPY3_start	Issue a request for a data transfer using current channel settings
ACPY3_wait	Wait for all data transfers to complete on a specific logical channel
ACPY3_waitLinked	Wait for an individual data transfer to complete on the logical channel

Table 7. ACPY3_Params Structure Fields

Structure Fields	Description
transferType	Transfer type: ACPY3_1D1D, ACPY3_1D2D, ACPY3_2D1D or ACPY3_2D2D
srcAddr	Source Address of the DMA transfer
dstAddr	Destination Address of the DMA transfer
elementSize	Number of consecutive bytes in each 1D transfer vector (ACNT)
numElements	Number of 1D vectors in 2D transfers (BCNT)
numFrames	Number of 2D frames in 3D transfers (CCNT)
srcElementIndex	Offset in number of bytes from beginning of each 1D vector to the beginning of the next 1D vector. (SBIDX)
dstElementIndex	Offset in number of bytes from beginning of each 1D vector to the beginning of the next 1D vector. (DBIDX)
srcFrameIndex	Offset in number of bytes from beginning of the first 1D vector of source frame to the beginning of the first element in the next frame. (SCIDX): signed value between -32768 and32767.
dstFrameIndex	Offset in number of bytes from beginning 1D vector of first element in destination frame to the beginning of the first element in next frame (DCIDX): signed value between -32768 and32767.

waitId	For a linked transfer entry: -1 : no individual wait on this transfer 0 <= waitId < numWaits : this transfer can be waited on or polled for completion. Ignored for single-transfers and for the last transfer in a sequence of linked transfers, which are always synchronized with waitId == (numWaits – 1).

2 Fundamental Framework Components DMA Concepts

The following subsections highlight the fundamental concepts and features supported by the Framework Components associated with DMA resources and services.

2.1 Logical DMA Channels & DMA Handles

The *logical DMA channel* is the fundamental software abstraction for characterizing hardware DMA resources and services. Each logical DMA channel represents a private hardware DMA resource and a private state identified by and accessed through a *DMA handle*. Applications are in charge of the physical DMA resources and grant IDMA3 channel handles to algorithms that request them using the IDMA3 interface.

2.2 Scratch vs. Persistent DMA Resources

The physical EDMA3 resources associated with each logical DMA channel can be can be requested as scratch or persistent, borrowing the concepts from xDAIS/IALG as it applies to memory. Application frameworks or resource managers such as DMAN3 can optimize resource allocation of scarce physical resources by arranging a group of algorithms to share the same physical resources when the channels have been requested as “scratch” (persistent=false).

When more than one processing thread shares a scratch resource (memory or IDMA3 channel), the application framework must ensure that at any given time only one of the processing threads is active. Upon activation, the processing thread may use the scratch resources without interference from other sharing threads. However they must assume that the state of the resource upon activation is undefined and perform necessary initialization. Additionally, at the point of deactivation, they must be completely finished with their use of the scratch resources (for example, by not leave any outstanding DMA transfers or unchecked transfer completion codes) and save any context information they may need for the next activation, as covered under xDAIS DMA Rule 1. For xDAIS-compliant algorithms, the activation and deactivation events correspond to the instance IALG::algActivate and IALG::algDeactivate calls.

2.3 DMA Resource Management using DMAN3

The IDMA3 interface does not specify or mandate the use of a particular framework DMA Resource manager. However, the TI Framework Components package provides DMAN3 as a fully-supported and configurable DMA Resource manager in charge of managing the EDMA3.0 physical resources that the application framework has given exclusively to DMAN3.

In a typical Framework Component based application, DMAN3 grants each algorithm the DMA resources it requests via the IDMA3 interface. The algorithm subsequently may call ACPY3 functions to configure logical channel settings, to request DMA transfers, or to synchronize with on-going transfers.

DMAN3 can be configured using a runtime C interface or statically using XDC tooling. The configuration provides DMAN3 with the physical EDMA3 resources: PaRAMs, TCCs, QDMA channels. DMAN3 configuration dictates how it allocates and manages the memory supplied to each logical DMA channel. DMAN3 additionally supports sharing of physical EDMA3 resources among algorithms created with the same scratch groupId whenever it is possible. It is the responsibility of the application framework to ensure that algorithms created using the same DMAN3 scratch groupIds do not pre-empt each other. See Section 4.2.3, “DMAN3 Configuration Examples” for some common allocation scenarios and tips.

2.4 DMA Transfer Submission and Synchronization using ACPY3

Algorithms or applications can use the physical DMA resources obtained through the IDMA3 interface directly or using any custom DMA library. However, the TI Framework Components package provides a high performance library, ACPY3, which may be used to perform a rich set of DMA operations using the logical DMA channels acquired through the IDMA3 protocol.

The ACPY3 API introduces several DMA transfer-related abstractions highlighted in the following subsections.

2.4.1 FIFO Ordering of DMA Transfers and Linked DMA Transfers

Several outstanding DMA transfer requests may be submitted asynchronously to run concurrently on separate logical DMA channels. Only transfer requests started on the same logical channel are guaranteed to start and complete in a strictly first-in first-out (FIFO) ordering.

In order to start multiple DMA transfers simultaneously but in a strict FIFO order, the IDMA3 interface introduces the notion of logical channels with more than 1 configurable transfer. Each ACPY3_start issued on a logical DMA channel, in effect, issues these as linked DMA transfers, similar to the mechanism provided by the EDMA3.0 hardware. In addition to the enforced FIFO ordering, the ACPY3 library submits linked transfers more efficiently, so their use is encouraged even if FIFO ordering is not strictly required.

ACPY3 additionally allows synchronizing with one or more intermediate transfers within a linked channel. The number of intermediate waits must be indicated in the “numWaits” field when requesting an IDMA3 channel that will be used to wait on intermediate transfers.

2.4.2 Channel Privacy and Synchronization

Algorithms have exclusive ownership of each received logical channel and can operate safely without fear of external components (such as other algorithms or other system code) accessing the channel and issuing transfer requests or changing channel configuration settings. They must, however, follow the ACPY3 API to activate and deactivate each IDMA3 channel during instance activation and deactivation, respectively.

All synchronization calls are issued on a per channel basis, as opposed to a per transfer basis. An algorithm can issue either a blocking wait, or a non-blocking query call to synchronize with a logical channel’s completion status.

2.4.3 DMA Transfer Configuration Settings

The purpose of acquiring logical channel handles is to submit DMA transfer requests. Each submitted DMA transfer request specifies a source and destination memory region. A background DMA activity asynchronously carries out the copying of the contents of the source memory region to the destination.

The configuration setting of a logical channel is similar to the hardware register settings of the underlying EDMA3.0 hardware DMA device. However, each logical channel retains its configured DMA transfer settings. The most recently configured transfer settings at the time the ACPY3_start function is called apply to the asynchronously started DMA transfer.

Two properties of DMA transfers make them desirable and performance critical for algorithms:

- The physical transfer/copy operation takes place in the “background” under the close control of specialized circuitry and controllers. This allows algorithms to issue transfer requests in advance and to perform other useful operations while data is being copied in the background.
- The physical layout of source or destination DMA transfer blocks need not be a single contiguous chunk of memory. By setting a few channel configuration parameters, algorithms can specify complex layout patterns. This can lead to significant performance improvements even if the algorithm cannot take advantage of asynchronous execution and the CPU sits idle while waiting for the transfer to complete.

The unit of DMA transfer is a block composed of frames and elements. Each DMA transfer is submitted on a logical channel via the ACPY3_start function. The *source* and *destination addresses* for the blocks and the *number of elements* in each frame are now part of the channel’s configuration settings. The configuration parameters are intrinsic properties of each logical channel and are set exclusively when the algorithm calls ACPY3 configuration functions. The previously configured properties of each logical channel at the time of an ACPY3_start request determine the actual memory copied from source to destination. Each DMA transfer is characterized by the following list configurable attributes. (Figure 2 illustrates the memory layout of a DMA transfer block characterized by these configuration parameters). Note that the element and frame index parameters can be configured independently for both source and destination.

- **transferType.** 1D-to-1D, 1D-to-2D, 2D-to-1D or 2D-to-2D
- **elementSize.** The number of 8-bit bytes per element. The element size for ACPY3 transfers can be a variable number, 1 ≤ number ≤ 65535, whereas the IDMA2/ACPY2 specification required this to be either 1,2, or 4 **.**
- **numFrames** [1]**.** The number of frames in a block, 1 ≤ number ≤ 65535
- **SrcElementIndex and DstElementIndex.** The offset in 8-bit bytes between the start addresses of two consecutive elements in a frame. Element indexes must be specified only when the source or destination is a 2D transfer. They are ignored for 1D transfers. Element indexes can be signed (i.e. negative) values in the range: -32767 ≤ number ≤ 32768

[1] The current implementation of ACPY3 does not support the use of numFrames or frame indexes due to QDMA-based hardware implementation restrictions. These may be supported in future releases using an EDMA-based implementation.

- **SrcFrameIndex and DstFrameIndex [1].** The offset in 8-bit bytes between the start addresses of the first elements of two consecutive frames. (NOTE: This corresponds to the AB-synched SRC/DST CIDX settings in the EDMA3.0 DMA parameters.) Frame indexes must be specified only when the number of frames > 1, otherwise they are ignored. Frame indexes can be signed (negative or positive) values in the range: -32767 ≤ number ≤ 32768
- **numElements.** The number of elements per frame, 1 ≤ number ≤ 65535
- **srcAddr and dstAddr.** 8-bit byte-addresses

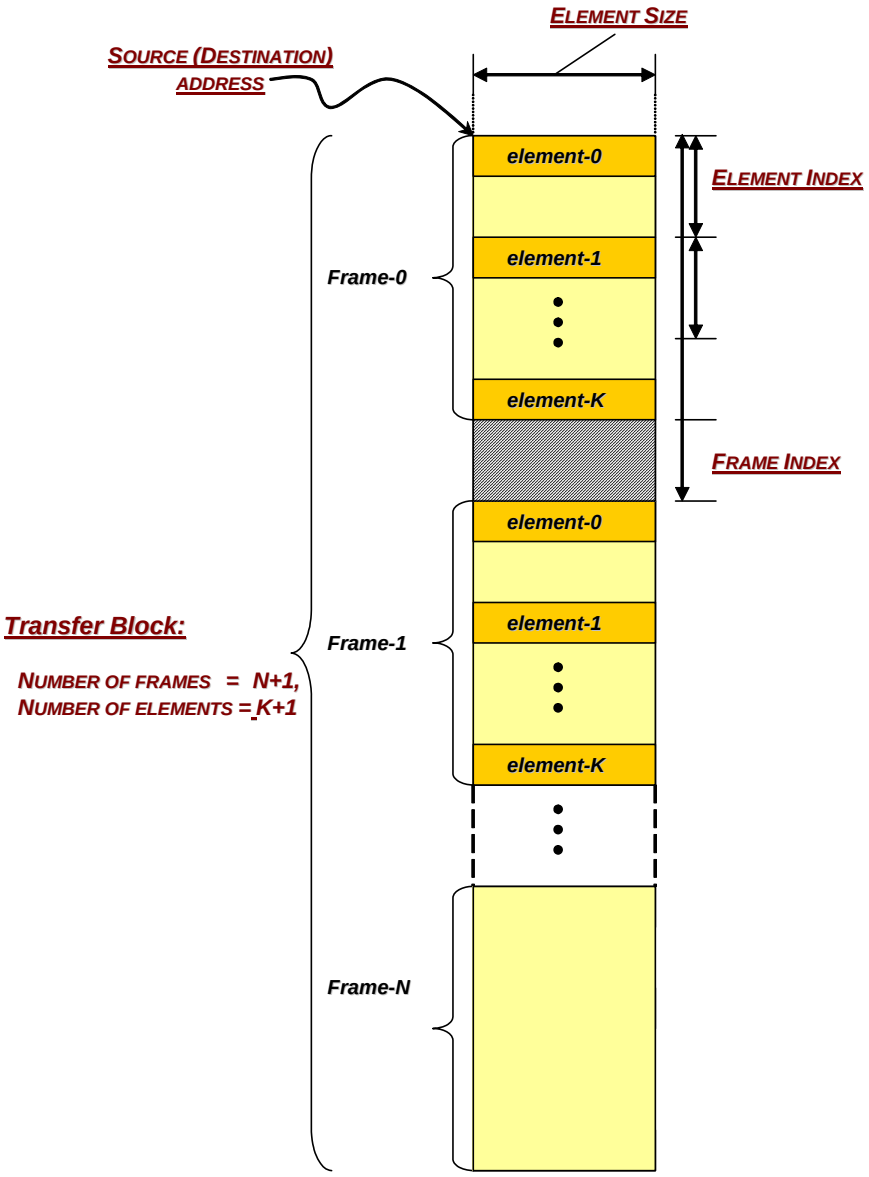


Figure 2. DMA Transfer Block

3 IDMA3: Standard Interface for Negotiating DMA Resources

TMS320 DSP Algorithm Standard (xDAIS) compliant algorithms designed for the 'C64x+ EDMA3 controller must implement the IDMA3 interface to publish and acquire DMA resources they will use. The application framework DMA Resource Manager (DMAN3 is one reference implementation provided by the Framework Components) calls IDMA3 interface functions to query and subsequently allocate and grant the requested DMA resources. The IDMA3 interface is similar to the xDAIS IDMA2 interface in terms of its definition and role. IDMA3 introduces the notion of a logical DMA channels abstraction via a handle similar to that used in IDMA2.

3.1 IDMA3 versus IDMA2

The following key changes have been introduced as compared to the IDMA2/ACPY2 interfaces:

- IDMA3 is no longer a pure hardware abstraction of generic DMA resources. Logical channels obtained through the IDMA3 interface expose some physical EDMA3 resources: Parameter RAM Sets (PaRAMs), Transfer Completion Codes (TCCs), and QDMA Channel ids.
- IDMA3 introduces the notion of *scratch* vs. *persistent* resources for the physical EDMA3 resources assigned to each IDMA3 channel. This approach is similar to the IALG scratch memory concept, which allows frameworks to efficiently share/overlay algorithm instance scratch buffers using instance activation and deactivation. This approach in IDMA3 allows sharing of TCCs and PaRAM entries and nicely hooks with IALG activate/deactivate events.

  Channels that cannot be used in a shared context must be requested with "Persistent=TRUE", otherwise the resource manager is free to arrange the sharing of granted physical DMA resources.
- Each IDMA3 channel can be optionally associated with a custom IDMA3 protocol. When a non-null "protocol" object is provided, the DMA resource manager uses IDMA3_Protocol functions to perform additional memory allocation for the logical DMA channel's environment ("env") field or to call protocol-specific handle initialization and de-initialization functions. This feature allows frameworks to support custom DMA service function libraries (ACPY3 is just one such library) with custom initialization and finalization functions.
- Each logical DMA channel can be assigned a relative channel priority.

For performance reasons, support was added for the following EDMA3.0-centric DMA concepts:

- Hardware linked transfers that can be quickly started through QDMA.
- Waiting on an intermediate transfer's completion in the case of linked transfers.

Finally, based on existing algorithm use cases, the following changes were made to allow ACPY3 to have a more streamlined and high-performance design:

- The queue IDs defined in IDMA2 are no longer needed. This means there is no requirement to enforce inter-channel FIFO ordering of submitted DMA transfers. When FIFO ordering is needed, you must use linked transfers.
- FIFO completion of DMA transfers is supported only on individual logical DMA channels and linked transfers.
- Support has been added for intermediate synchronization points associated with "wait-ids" for individual transfers within a linked transfer chain.

3.2 IDMA3 Interface Definition

The IDMA3 interface is implemented by algorithms that need EDMA3 resources. The application framework DMA Resource Manager calls the algorithm's IDMA3 interface functions to query and subsequently allocate and grant the requested DMA resources. (DMAN3 is a reference implementation of a DMA Resource Manager provided by the Framework Components.) If required, the framework calls IDMA3 channel-specific IDMA3_Protocol functions to allocate, initialize, and free additional channel environment memory, which is part of the logical DMA channel state.

The algorithm implements the IDMA3 interface by defining and initializing a global structure of type IDMA3_Fxns. Every function defined in this structure must be implemented and assigned to the appropriate field in this structure. Figure 3 illustrates the calling sequence for IDMA3 functions and how these functions relate to the IALG functions performed during algorithm instance creation and real-time operation.

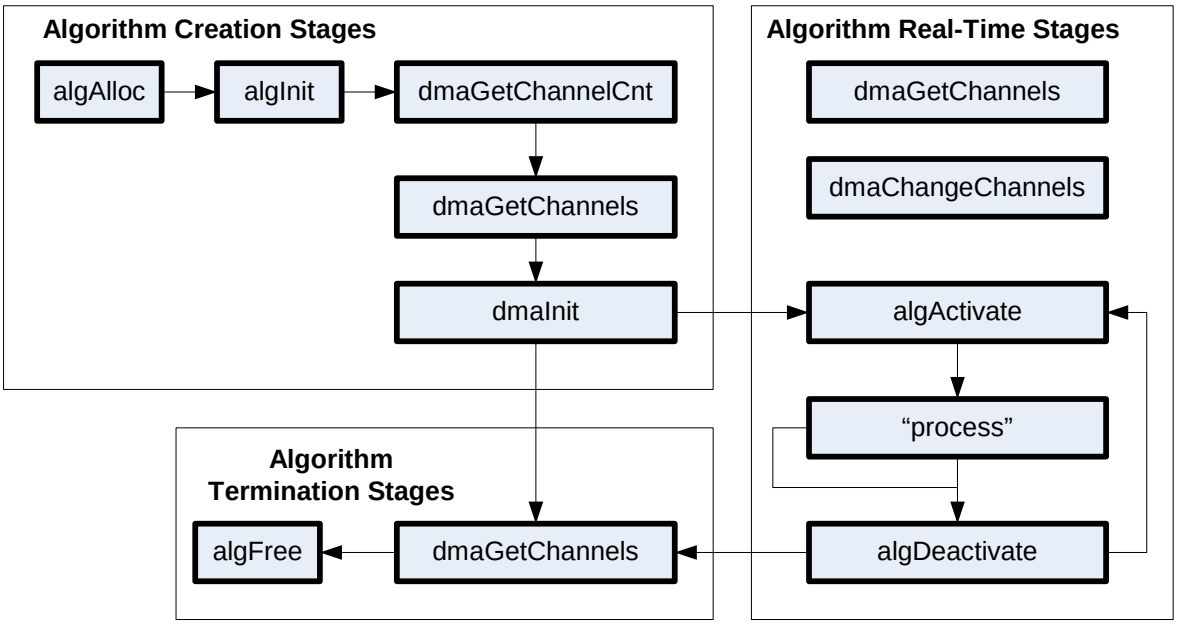


Figure 3. IDMA3 Function Calling Sequence

The dmaChangeChannels() and dmaGetChannels() functions can be called at any time in the algorithm's real-time stages. The algMoved() and algNumAlloc() functions were omitted from this figure for simplicity.

The dmaGetChannels() and dmaInit() functions must be called after algInit() and before algActivate(). The dmaGetChannelCnt( ) function can be called before the algorithm instance object is created if the framework wants to query the algorithm about its DMA resource requirements before creating the instance object.

Note: Framework Components provides a DMA resource manager, DMAN3, which provides functions to perform the IDMA3 operations to create algorithms that implement the IDMA3 interface. This is discussed in Section 4.

Figure 4 illustrates a typical system with an algorithm implementing the IALG and IDMA3 interfaces and the application with a DMA manager. Notice that the algorithm calls the ACPY3 run-time functions, which are implemented by the Framework Components. The ACPY3 interface provides a comprehensive list of DMA functions that an algorithm can call using the IDMA3 handles to program the logical DMA channels obtained through the IDMA3 interface. These functions allow the algorithm to:

- Configure each logical channel's DMA transfer settings
- Submit asynchronous DMA transfer requests
- Synchronize with the completion status of submitted transfers (both blocking and non-blocking).

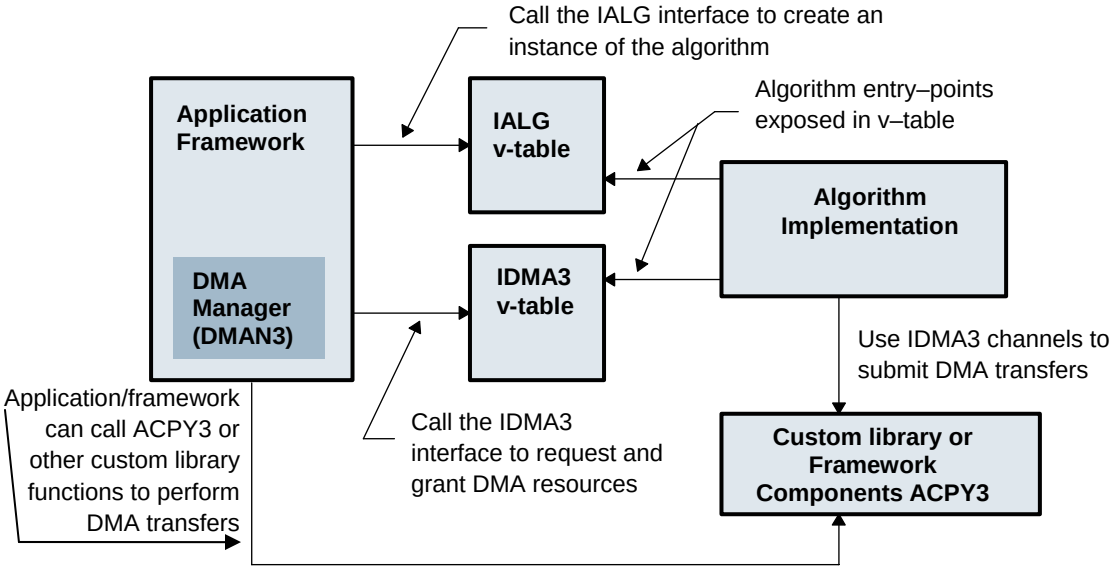


Figure 4. Algorithm Implementing IALG and IDMA3 Interfaces and Application Using Framework Components

3.2.1 DMA Channel Descriptor: IDMA3_ChannelRec

The IDMA3 interface functions use the IDMA3_ChannelRec structure definition to characterize the properties of each logical DMA channel to be granted to the requesting algorithm or module. The DMA Manager utilizes the information passed in a channel descriptor and responds by constructing a logical IDMA3 channel containing the physical DMA resources that are assigned to the channel, and passes the handle of the DMA channel using the same channel descriptor.

This structure has the following data fields:

IDMA3_Handle	handle
Int	numTransfers
Int	numWaits
IDMA3_Priority	priority
IDMA3_ProtocolHandle	protocol
Bool	persistent

The following list describes these fields:

- **IDMA3_Handle IDMA3_ChannelRec::handle**
  The handle to a logical DMA channel.

- **Int IDMA3_ChannelRec::numTransfers**
  The maximum number of linked DMA transfers that will be submitted using this logical channel handle. Single (==1) or Linked (>= 2).

- **Int IDMA3_ChannelRec::numWaits**
  The maximum number of transfers that can be independently waited upon. This includes intermediate transfers of a linked DMA transfer. A wait, with a waitId of (numWaits – 1) is configured to indicate the end of the linked or single transfer on a particular channel. Hence, while requesting a handle with a configured number of numWaits, always count the default wait required to indicate the end of transfer. For example, if only 1 intermediate transfer is to be tracked, IDMA3_ChannelRec::numWaits should be 2. Use a waitId of 0 to track the intermediate transfer and a waitId of 1 (numWaits – 1) to track the end of the entire transfer.

- **IDMA3_Priority IDMA3_ChannelRec::priority**
  The relative priority recommendation for transfers submitted on this channel: High, Medium, or Low. See Section 3.2.5 for constants to use for priorities.

- **IDMA3_ProtocolHandle IDMA3_ChannelRec::protocol**
  When non-null, the protocol object provides an interface for querying and initializing logical DMA channel for use by the given protocol. The protocol can be IDMA3_PROTOCOL_NULL; in this case no “env” is allocated. For example, when requesting a logical channel to be used with ACPY3 functions, the protocol needs to be set to &ACPY3_PROTOCOL.

- **Bool IDMA3_ChannelRec::persistent**
  When persistent is set to TRUE, the PaRAMs and TCCs are allocated exclusively for this channel. They cannot be shared with any other IDMA3 channel.

3.2.2 IDMA3 Functions: IDMA3_Fxns

The application framework calls the following functions to query and grant DMA resources requested by the algorithm at initialization time, and to make changes to these resources at run-time. These IDMA3 functions must be implemented by all algorithms that need to access physical DMA resources of the EDMA3 controller.

Void *implementationId
Void (*dmaChangeChannels)(IALG_Handle, IDMA3_ChannelRec *)
Uns (*dmaGetChannelCnt)(Void)
Uns (*dmaGetChannels)(IALG_Handle, IDMA3_ChannelRec *)
Int (*dmaInit)(IALG_Handle, IDMA3_ChannelRec *)

- **implementationId** – This holds a unique value that identifies the module implementing this interface. This same value must be used in all interfaces implemented by the module. Since all compliant algorithms must implement the IALG interface, it is sufficient for these algorithms to set this field to the address of the module's declared IALG_Fxns structure.
- **dmaChangeChannels()** – The application framework's DMA resource manager calls this function whenever a logical channel is moved at run-time.
- **dmaGetChannelCnt()** – The application framework's DMA resource manager calls this function to query an algorithm about the maximum number of logical DMA channels requested.
- **dmaGetChannels()** – The application framework's DMA resource manager calls this function to query an algorithm about its DMA channel requests at initialization time, or to get the current channel holdings.
- **dmaInit()** – The application framework's DMA resource manager calls this function to grant DMA handle(s) to the algorithm at initialization time. The algorithm uses this function to complete the initialization of its instance object.

3.2.3 IDMA3 Object and Handle Structures: IDMA3_Obj

The IDMA3 channel object holds the private state associated with each logical DMA channel. The application framework DMA manager creates and initializes the logical channel provisioned with the physical EDMA3 resources that are exposed in this structure definition, and passes its handle to the requesting algorithm using the IDMA3 interface.

The holder of a handle to an IDMA3 channel may directly access the physical resources assigned to the channel or use a standard (ACPY3) or custom DMA functional library that recognizes IDMA3 channel handles.

When the channel is created with its persistent field set to “false”, the physical DMA resources assigned to the channel are be considered to be “scratch” memory, as the definition applies to IALG memory attributes. Algorithms must perform initialization of the resource state each time they are put in an “active” state (via an algActivate call) and must save any necessary channel context when they are deactivated (via algDeactivate). When using ACPY3, calling ACPY3_activate and ACPY3_deactivate during instance activation and deactivation, respectively, performs this required context initialization and deinitialization.

Figure 5 shows how the algHandle points to an IALG_Obj object, which in turn points to an IDMA3_Obj object.

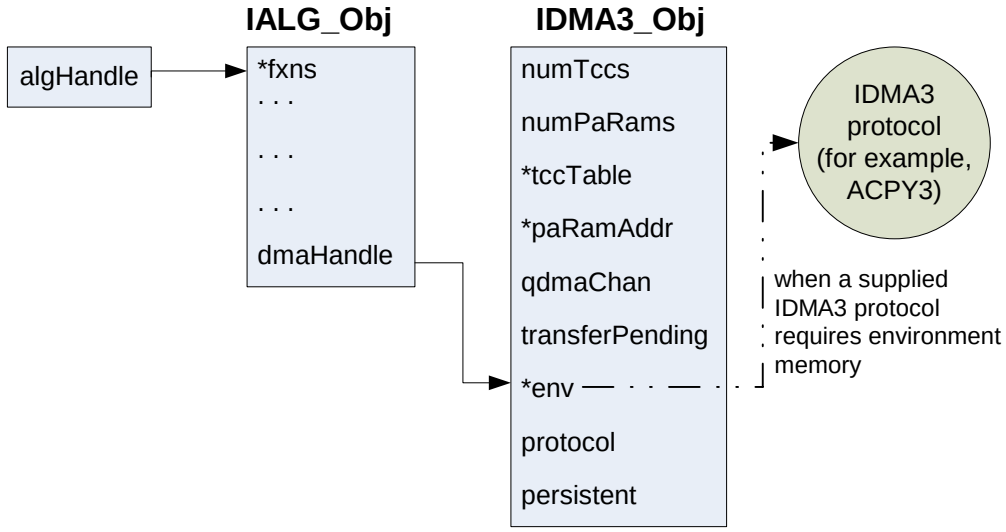


Figure 5. IDMA3 Logical Channels

The IDMA3_Handle data type is a pointer to the IDMA_Obj structure. The IDMA3_Obj structure has the following data fields:

MdUns	numTccs
MdUns	numPaRams
SmUns *	tccTable
Uns *	paRamAddr
MdUns	qdmaChan
Bool	transferPending
Void *	env
IDMA3_ProtocolHandle	protocol
Bool	persistent

The following list describes these fields:

- **MdUns IDMA3_Obj::numTccs**
  The number of TCCs allocated to this channel.

- **MdUns IDMA3_Obj::numPaRams**
  The number of contiguous EDMA3 Parameter RAM (PaRAM Set) register sets allocated to this channel.

- **SmUns* IDMA3_Obj::tccTable**
  The address of the array containing TCCs assigned to this channel.

- **Uns* IDMA3_Obj::paRamAddr**
  The physical address of the first PaRAM assigned to this channel.

- **MdUns IDMA3_Obj::qdmaChan**
  The index of the physical QDMA channel assigned to the handle.

- **Bool IDMA3_Obj::transferPending**
  The channel state. This must be maintained by the channel handle owner. The transferPending state must be set to true each time a new DMA transfer is physically submitted using this DMA channel. The state must be cleared to false before a new physical transfer can be submitted using this channel.

- **Void* IDMA3_Obj::env**
  An optional “environment” memory that may be allocated as a private extension of the channel state. Memory for the “env” must be supplied by the framework prior to calling the IDMA3_Fxns::dmaInit function only if the IDMA3 channel descriptor requires it by providing a non-null “protocol” handle and a non-null getEnvMemRec() function pointer.

  The IDMA3_Protocol object’s getEnvMemRec() function characterizes the size, alignment and space attributes of the “env” memory needed for the channel. The framework is responsible for allocating and reclaiming the “env” memory.

  During channel creation, the “env” pointer must always be created as private and persistent memory assigned to the IDMA3 channel object. However, the framework/resource manager is also allowed to allocate the requested internal “env” memory as “scratch” memory that can only be used when the channel is in an active state.

  In the scratch allocation case, the framework/resource manager must still allocate the “env” as a “persistent” shadow memory, possibly in “external memory”, and then pass the address of the scratch “internal” “env” memory in the first word of the returned IDMA3_Obj's “env” pointer. If a channel’s “env” memory is created as “persistent” with no “scratch” shadow, then the first word of the env memory must be set to null.

- **IDMA3_ProtocolHandle IDMA3_Obj::protocol**
  When non-null, this points to the channel protocol function table used by the DMA manager to interrogate and provision memory for the channel’s “env” area.

- **Bool IDMA3_Obj::persistent**
  This flag indicates whether the channel was allocated with the persistent property.

3.2.4 IDMA3 Protocol Object for Channel Environment Memory Management

The IDMA3 protocol object (the “protocol” field of the channel descriptor) is used only when the requestor of the IDMA3 channel requires:

- additional environment memory (assigned to the channel’s “env” pointer) to be allocated by the framework as part of the channel object, or
- custom initialization or de-initialization functions to be called by the framework upon channel creation and deletion.

If these requirements do not apply, you may request the IDMA3 channel with a null “protocol” field and skip the details of this section.

Algorithms or framework libraries (such as ACPY3) may need some additional persistent and private “environment” memory to be associated with each IDMA3 channel in order to manage channel state or to create shadow copies of registers, data structures, etc. The IDMA3 interface defines a protocol, via the IDMA3_ProtocolObj specification, that can be implemented and used by individual algorithms or functional DMA libraries to request and receive channel environment memory from the resource management framework during channel creation.

Custom DMA libraries may take advantage of this feature and require that IDMA3 channels are requested and created using specific IDMA3 Protocol Objects. An example of this is the ACPY3 library, which supplies its own protocol object, ACPY3_PROTOCOL. Algorithms or applications that will use ACPY3 to submit DMA transfers are simply required to set the “protocol” field to the address of ACPY3_PROTOCOL when they request an IDMA3 channel for ACPY3 use.

When the IDMA3 channel descriptor (**IDMA3_ChannelRec)** contains a non-null IDMA3_ProtocolObj reference, the provided functions are called to determine channel environment memory requirements and to perform initialization and de-initialization of the channel object. If the IDMA3 protocol does not require the functionality associated with any particular function, it may be set to null. The application framework and DMA resource manager are responsible for calling the IDMA3_Protocol functions.

The IDMA3_ProtocolObj contains the following fields:

String name
Void (*getEnvMemRec)(IDMA3_ChannelRec *, IDMA3_MemRec *)
Bool (*initHandle)(IDMA3_Handle)
Bool (*deInitHandle)(IDMA3_Handle)

The following list describes these fields:

- **String IDMA3_ProtocolObj::name**
  The name of the protocol.

- **Void(*IDMA3_ProtocolObj::getEnvMemRec)(IDMA3_ChannelRec *, IDMA3_MemRec *)**
  The function is called by the application framework/DMA manager to obtain the IDMA3 protocol's memory requirements (IDMA3_MemRec) for its environment for the given IDMA3 channel descriptor. This is usually done when creating a logical DMA channel.

- **Bool(* IDMA3_ProtocolObj::initHandle)(IDMA3_Handle)**
  This function is called after allocation. It allows the IDMA3 protocol to do any initialization of its environment. It initializes the “env” memory passed in the IDMA3 channel handle and any other channel state. Returns TRUE on success, FALSE otherwise. If FALSE is returned channel creation fails.

  If the framework/resource manager allocates the requested internal “env” memory as “scratch”, the “env” pointer passed in the IDMA3_Handle points to a persistent and private shadow memory (possibly in “external memory”), which contains the address of the “scratch” allocated “internal” “env” memory in the first word of the “persistent” “env” pointer.

  If the first word of the env memory is NULL, then no separate “scratch” memory has been allocated and the “env” memory itself is “persistent”.

- **Bool(* IDMA3_ProtocolObj::deInitHandle)(IDMA3_Handle)**
  This function is called when a channel is freed. It deinitializes a channel before assigned resources and memory are freed by the DMA manager. It is called so that the IDMA3 protocol can do any required de-initialization or freeing any memory that may have been allocated in initHandle().

3.2.5 IDMA3 Enumeration Type Documentation

The following enumerated types are defined for use by the IDMA3 API:

- **enum IDMA3_MemType**

Constant	Memory Type
IDMA3_INTERNAL	Internal data memory
IDMA3_EXTERNAL	External data memory

- **enum IDMA3_Priority**

Constant	IDMA3 Priority Level
IDMA3_PRIORITY_URGENT	Urgent
IDMA3_PRIORITY_HIGH	High
IDMA3_PRIORITY_MEDIUM	Medium
IDMA3_PRIORITY_LOW	Low

4 DMAN3: 'C64x+ DMA Resource Manager

DMAN3 is the DMA Resource Manager responsible for granting and reclaiming physical DMA resources such as EDMA3.0 PaRAM Register sets and transfer completion codes (TCCs). DMAN3 creates logical DMA channels based on the IDMA3 interface specification and grants these to requesting algorithms or other software components.

The application framework configures DMAN3 during system start-up (prior to its first use) with a dedicated set of EDMA3.0 physical DMA resources: PaRAM Sets, TCCs, and QDMA channels. Algorithms receive DMA handles from DMAN3 and use them to call ACPY3 functions to configure logical channel settings, to request DMA transfers or to synchronize with on-going transfers.

DMAN3 functions are intended to provide application frameworks a convenient and easy-to-use layer to integrate algorithms that request DMA resources.

4.1 Using DMAN3 for Algorithm Integration

The following steps provide a generic and convenient set of instructions for using the DMAN3 module to instantiate algorithm instances that request DMA resources. The code examples are from the fastcopytest.c example provided with this application note.

1. Include the DMAN3 and ACPY3 modules in the application. You can use the DMAN3 module as provided or make changes to it as needed by your application.

```
#include <ti/sdo/fc/dman3/dman3.h>
#include <ti/sdo/fc/acpy3/acpy3.h>
```

2. Create an algorithm instance using the IALG interface and set values for fields in the algorithm-specific params structure, which is declared in i<mod>.h. Use the standard IALG interface to allocate and grant the memory buffers requested by the algorithm and initialize the instance object.

```
FCPY_Params fcpyParams;
FCPY_Handle alg;
IDMA3_Fxns *dmaFxns[NUMALGS];
IALG_Handle alg[NUMALGS];
...
FCPY_init();

fcpyParams = FCPY_PARAMS; /* use the default creation parameters */

if ((alg = FCPY_create(&FCPY_IALG, &fcpyParams)) == NULL) {
    SYS_abort("Could not create algorithm instance");
}
```

3. Call the DMAN3 and ACPY3 module initialization functions.

```
/* Assign internal and external Heaps to DMAN3 module. */
DMAN3_PARAMS.heapInternal = L1DHEAP;
DMAN3_PARAMS.heapExternal = EXTERNALHEAP;

/* Initialize DMA manager & ACPY3 lib for xDAIS algs and grant DMA resources */
DMAN3_init();
ACPY3_init();
```

4. Define a scratch sharing context by associating a group-id to use for DMA resource allocation using DMAN3 module. All subsequent DMAN3 calls using the same group-id will share underlying physical DMA resources.

```
Int groupId = 1;
Int numAlgs = 1;
```

5. Use the DMAN3 module to grant the DMA resources requested by the algorithm.

```
alg[0]     = (IALG_Handle)fcpyAlg;
dmaFxns[0] = &FCPY_IDMA3;

if (DMAN3_grantDmaChannels(groupId, alg, dmaFxns, numAlgs) != DMAN3_SOK) {
    SYS_abort("Problem adding algorithm's dma resources");
}
```

4.2 DMAN3 Configuration

The DMAN3 module manages physical and logical DMA resources for DSP algorithms and drivers. Allocation of following physical EDMA/QDMA resources is needed to ensure interoperability:

- TCCs (Transfer Completion Codes)
- PaRAM entries (Parameter RAM)
- QDMA channels

The DMAN3 functions manage system EDMA/QDMA resources that are pre-allocated to it by initializing the global DMAN3 configuration parameter DMAN3_PARAMS during the initial system configuration. The DMAN3_init() function uses the initial system DMAN3 configuration settings and assumes that it exclusively owns and manages these resources. For example, the system integrator could identify and allocate 48 PaRAM entries to DMAN3. The rest could be distributed between the drivers, power managers on the chip, and possibly other frameworks that have access to the shared DMA resource.

4.2.1 Introduction to Configuration Options

The following section describes the configuration parameters of the DMA Manager module, DMAN3. Each configuration option can be set at design time by the system integrator to ensure optimal sharing of DMA resources for the execution environment.

There are two ways to configure DMAN3 parameters.

- You can use a low-level C language based approach to directly modify an interface-defined global configuration structure, DMAN3_PARAM, as defined in the DMAN3 API specification. The DMAN3_Params structure defines the configurable parameters of the DMAN3 module.
- Alternately, you can use XDC tooling to configure the RTSC module, DMAN3.

The XDC tooling approach results in the generation of the same low-level C based global configuration structures, so the type of configuration technology used does not matter to the underlying DMAN3 library implementation.

4.2.2 Configuration Parameters

The following is a list of the configurable DMAN3 parameters and a description of each.

Two names are given for each configuration parameter in this section. First, the DMAN3 parameters named in the DMAN3_Params struct in the dman3.h header file are listed as DMAN3_Params::<attribute name>. The DMAN3 parameters are also configurable via the RTSC DMAN3 package interface; these names are specified in DMAN3.xdc and are listed in this section as DMAN3.<attribute name>.

- **Uns* DMAN3_Params::qdmaPaRamBase**
  UInt DMAN3.qdmaPaRamBase

  This identifies the physical base address of PARAM0 in the EDMA3/QDMA hardware whose resources are being managed by DMAN3. Set this to the actual base address of the PaRAM entries for the particular device on which DMAN3 is being configured.

- **Uns DMAN3_Params::maxPaRamEntries**
  UInt DMAN3.maxPaRamEntries

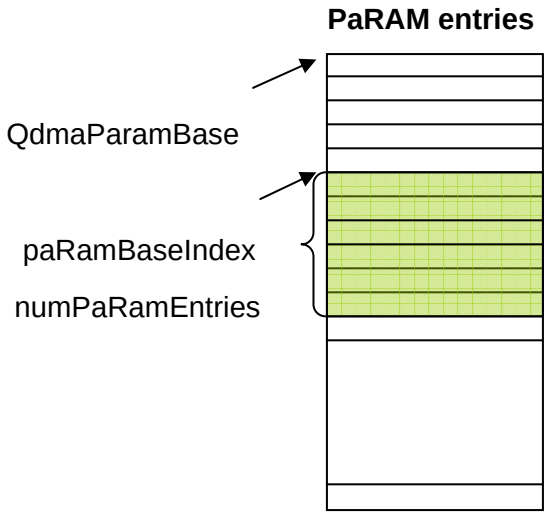
  The total number of PaRAM entries on the target hardware. Set this to the actual number of PaRAM entries available on the particular device on which DMAN3 is being configured.

- **Uns DMAN3_Params::paRamBaseIndex**
  UInt DMAN3.paRamBaseIndex

  (0>value>255) This represents the first PaRAM table entry number that is assigned by configuration for exclusive DMAN3 allocation. Contiguous PaRAM entries will be allocated exclusively for DMAN starting from this index.

- **Uns DMAN3_Params::numPaRamEntries**
  UInt DMAN3.numPaRamEntries

  The number of PaRAM table entries starting at DMAN3_PARAM_BASE_INDEX assigned by configuration for exclusive DMAN3 allocation.



- **Uns DMAN3_Params::maxQdmaChannels**
  UInt DMAN3.maxQdmaChannels

The total number of physical QDMA channels available on the target hardware. Set this to the actual number of QDMA channels that are available on the device for which DMAN3 is being configured.

- **Uns DMAN3_Params::numQdmaChannels**
  UInt DMAN3.numQdmaChannels

  The number of physical QDMA channels that are assigned to DMAN3 via configuration. This number is assigned to DMAN3_NUM_QDMA_CHANNELS.

- **Uns* DMAN3_Params::qdmaChannels**
  UInt DMAN3.qdmaChannels[DMAN3_NUM_QDMA_CHANNELS]

  An array of size DMAN3_NUM_QDMA_CHANNELS that contains the channel numbers of the physical QDMA channels assigned to DMAN3 via configuration. DMAN3 exclusively owns these QDMA channels and assumes that they will not be used by any other resource. These channels need not be contiguous.

- **Uns DMAN3_Params::tccAllocationMaskH**
  UInt DMAN3.tccAllocationMaskH

  A 32-bit bitmask representing a configuration-provided list of TCCs for exclusive DMAN3 allocation in the range 32-63. For example, for TCCs in the range 32-63 the High Mask (tccAllocationMaskH) is configured so that a “1” in bit position “i" indicates the TCC: (32 + i) is assigned to DMAN3.

- **Uns DMAN3_Params::tccAllocationMaskL**
  UInt DMAN3.tccAllocationMaskL

  A 32-bit bitmask representing a configuration-provided list of TCCs for exclusive DMAN3 allocation in the range 0-31. For example, for TCCs in the range 0-31 the Low Mask (tccAllocationMaskL) is configured so that a “1” in bit position “i" indicates the TCC: “i" is assigned to DMAN3.

- **SmUns DMAN3_Params::numTccGroup[DMAN3_MAXGROUPS]**
  Int DMAN3.numTccGroup[DMAN3_MAXGROUPS]

  An array containing the number of TCCs to be assigned to groups for sharing.

  Algorithms in a particular scratch group (with the same group ID), share the same shared scratch pool. DMA channels requested by a given group use TCCs allocated for that group. The numTccGroup array indicates, for each group ID, the number of TCCs allocated to it.

  Depending on the calls you make to create DMA channels in the same scratch group, you may or may not share the same TCCs.

- **MdUns DMAN3_Params::numPaRamGroup[DMAN3_MAXGROUPS]**
  Int DMAN3.numPaRamGroup[DMAN3_MAXGROUPS]

  Arrays containing the list of PaRAM entries that will be assigned to groups for sharing.

  Algorithms in a particular scratch group (with the same group ID), share the same shared scratch pool. DMA channels requested by a given group use TCCs allocated for that group. The numPaRamGroup array indicates, for each group ID, the number of PaRAMs allocated to it.

  Depending on the calls you make to create DMA channels in the same scratch group, you may or may not share the same PaRAMs

- **Bool DMAN3_Params::idma3Internal**
  Bool DMAN3.idma3Internal

  Use the internal memory heap to dynamically allocate IDMA3 objects. If this value is “false”, IDMA3 objects are allocated in the heap specified by heapExternal. If the value of idma3Internal is “true”, IDMA3 objects are allocated in the heap specified by heapInternal.

- **Int DMAN3_Params::heapInternal**
  String DMAN3.heapInternal

  The Memory Heap ID for dynamic allocation of DMAN3 objects that must be allocated in L1D Internal RAM. The internal heap could be used for allocation of memory for IDMA3 objects if indicated by the idma3Internal flag.

  A value of -1 for heapInternal indicates that the heap is NOT DEFINED. If heapInternal is not defined, then any IDMA3 protocol call that requests IDMA3 objects to be created in the internal memory, will fail. When using RTSC module configuration you are required to provide a valid DSP/BIOS heap label.

- **Int DMAN3_Params::heapExternal**
  String DMAN3.heapExternal

  The Memory Heap ID for dynamic allocation of private DMAN3 data structures that can be allocated in external memory. The heapExternal memory space is used by DMAN3 to allocate memory for holding DMA channel descriptors that point to logical DMA channels. This space could also be used to hold IDMA3 objects if the flag idma3Internal is set to false.

  A value of -1 indicates that the heap is NOT DEFINED. If heapExternal is not defined then DMAN3 attempts to use heapInternal. As a consequence, at least one of them must be defined. When using RTSC module configuration you are required to provide a valid DSP/BIOS heap label.

- **DMAN3_ScratchAllocFxn DMAN3_Params::scratchAllocFxn**
  String DMAN3.scratchAllocFxn

  Algorithms, while requesting IDMA3 channels from DMAN3, might specify a particular IDMA3 protocol that provides custom DMA services. This protocol might require additional memory allocation for the channel’s environment (“env” field). The scratchAllocFxn is a function call that when non-null is used to allocate the protocol’s environment memory. If scratchAllocFxn is null, this additional memory is allocated in the internal heap specified by the heapInternal parameter.

If the scratchAllocFxn is provided, but fails to allocate the memory, DMAN3 attempts to allocate the memory dynamically using its heapInternal.

Note: The Framework Component DSKT2 provides a scratchAllocFxn implementation that can be used to allocate, when possible, the IDMA3 protocol handle's environment memory in the shared scratch area, provided that DSKT2 was also used to create the actual algorithm instance using the same scratch group ID.

- **DMAN3_ScratchFreeFxn DMAN3_Params::scratchFreeFxn**
  String DMAN3.scratchFreeFxn

  The scratchFreeFxn is a function call that is used to free memory allocated using the scratchAllocFxn by DMAN3.

  Note: The Framework Component DSKT2 provides a scratchFreeFxn implementation that must be called if memory has been allocated using DSKT2's scratchAllocFxn

- **Uns DMAN3_Params::nullPaRamIndex**
  UInt DMAN3.nullPaRamIndex

  The index of the PaRAM entry to be reserved as a "NULL" PaRAM. Any QDMA channel that is inactive is mapped to the nullPaRamIndex entry by setting up the QCHMAP register accordingly. This PaRAM index must not be used by the application domain for scheduling EDMA3 transfers.

  Note: This index is not counted against the numPaRAMEntries configuration setting.

- **bool DMAN3.debug**

  A value of true for the debug parameter enables the debug profile of the DMAN3 library. This results in a larger and slower version of the library being linked in. It provides extra parameter checking and causes debug trace statements to be generated in the DSP/BIOS SYS trace buffer.

4.2.3 DMAN3 Configuration Examples

In the following subsections we discuss some common DMAN3 configuration scenarios.

4.2.3.1 Configuring DMAN3 for Optimal Sharing of Physical DMA Resources

DMAN3 attempts to maintain a separate sharable pool of physical EDMA3 resources such as PaRAMs and TCCs for each scratch group, identified by the groupId. The groupId argument passed to a DMAN3 function is used to determine the shared scratch resource pool to satisfy the allocation request. However, the shared resource pool for a particular groupId does not get created until the first DMAN3 allocation call using that groupId. The initial size of the shared scratch PaRAM or TCC pool is determined by the "greater" of:

- The size of the resource needed to satisfy the current allocation request.
- The DMAN3 configuration setting for DMAN3::numPaRamGroup[groupId] for the PaRAM pool.
- The DMAN3 configuration setting for DMAN3::numPaRamGroup[groupId] for the TCC pool.

Therefore it may be important to configure the DMAN3 numPaRamGroup and numPaRamGroup properties for optimal allocation to ensure that all allocation requests can be satisfied using the same shared resource pool for that groupId.

For example, if algorithm instances A and B, assigned to the same groupId, i, require 10 PaRAMs for A's channel requirements and 20 PaRAMs for B, then setting:

```
DMAN3::numPaRamGroup[i] = 20
```

ensures that, irrespective of the order in which the A and B instances are created, they can be assigned PaRAMs from the same shared PaRAM pool assigned to group:i.

For the same example, suppose you used a different DMAN3 setting such as:

```
DMAN3::numPaRamGroup[i] = 5
```

If A is created first and then B, then 10 PaRAMs are created for A, and the shared PaRAM pool has size=10 for group:i. This results in not being able to satisfy B's PaRAM allocation from group:i's shared pool. DMAN3 would still try to satisfy B's request by privately allocating 20 PaRAMs to B, in a non-sharable manner, if at the time of the call DMAN3 has enough PaRAMs available for non-shared allocation. If there are not enough such PaRAMs available, B's allocation returns failure, indicating not enough PaRAMs.

With the latter setting, if B is created first, the initial shared PaRAM pool allocation of 20 PaRAMs for group:i would be able to satisfy the subsequent allocation request for A from the shared pool. But, in this scenario, the optimal allocation imposes an order of creation constraint (B first, then A) for the application.

A similar argument applies when configuring the DMAN3 numTccGroup setting.

4.2.3.2 DMA Channel Memory Allocation

The DMAN3 configuration settings, .heapInternal and .heapExternal are used for allocation of IDMA3 channel objects and internal data structures. These heaps can be used to control memory allocation by DMAN3 in conjunction with the DMAN3 Config parameter .idma3Internal (the default is true).

A value of “false” means the IDMA3 objects are allocated in the heap specified by DMAN3_PARAMS.heapExternal. If the value of .idma3Internal is “true”, IDMA3 objects are allocated in the heap specified by DMAN3_PARAMS.heapInternal.

4.2.3.3 IDMA3 Protocol Environment Memory Allocation

In addition to the memory allocated for the IDMA3 channel handle, if the IDMA3 channel is requested with an IDMA3 Protocol, such as ACPY3_PROTOCOL, there may be additional memory allocation for the channel's environment memory (assigned to “env”.) The IDMA3_Protocol may additionally require the environment memory to be allocated in “internal” memory with some size and alignment constraints, as is the case for ACPY3. In this case, the following DMAN3 configuration settings become important:

```
.scratchAllocFxn (default = null)
.scratchFreeFxn  (default = null)
```

When .scratchAllocFxn and .scratchFreeFxn are null, DMAN3 uses .heapInternal to dynamically allocate and free IDMA3 channel env memory. However when not-null, it calls these functions instead to allocate and free the “env” memory.

While there is no dependency between DMAN3 and DSKT2, it is possible to configure DMAN3 by plugging these functions with the specialized DSKT2 APIs DSKT2_allocScratch() and DSKT2_freeScratch() for the DMAN3 config params scratchAllocFxn and scratchFreeFxn respectively. Doing this causes the internal memory requests for the IDMA3_Protocol requested channel “env” memory to be allocated from the DSKT2 managed scratch memory pool for the algorithm instance, when shared scratch DSKT2 memory is available.

4.2.4 Configuring DMAN3 Without Using RTSC

The non-RTSC way to configure the DMAN3 configuration parameters is quite straightforward. To modify the DMAN3 configuration parameters, include the *dman.h* header file in your C file, and over-ride each of the parameters with the modified values. See the code snippet below as an example of how to modify the heapInternal and heapExternal parameters:

```
#include <std.h>
#include <ti/sdo/fc/dman3/dman3.h>
. . .
extern int L1DHEAP;      /* DSP BIOS Heap Label for L1DRAM memory allocation */
extern int EXTERNALHEAP; /* DSP BIOS Heap Label for external memory allocation */

Int main(Void)
{
    DMAN3_PARAMS.heapInternal = L1DHEAP;
    DMAN3_PARAMS.heapExternal = EXTERNALHEAP;
    . . .
    DMAN3_init();
```

4.2.5 Configuring DMAN3 Using RTSC Tooling

In the program configuration file, define the DSP/BIOS heaps that will be assigned to DMAN3 heaps. For example, the following code snippet from the dman3/example/fastcopytest.tcf DSP/BIOS configuration file creates two DSP/BIOS heaps with appropriate attributes and heap labels: EXTERNALHEAP and L1DHEAP:

```
utils.loadPlatform("ti.platforms.evmDM6446");
DDR = prog.module("MEM").instance("DDR2");
/*
 *  Create external memory segment for this (simulated) board
 *  Enable heaps in it and define the label for heap usage.
 */
DDR.base             = 0x83F00000;
DDR.len              = 0x0FFE00;     // may be much bigger -- this is sim
DDR.space            = "code/data";  // code/data so we can place code in it
DDR.createHeap       = true;
DDR.enableHeapLabel  = true;
DDR["heapLabel"]     = prog.extern("EXTERNALHEAP");
DDR.heapSize         = 0x8000;
DDR.comment          = "DDR";

/*
 *  Enable heaps in the L1DSRAM (internal L1 cache ram, fixed size)
 *  and define the label for heap usage.
 */
bios.L1DSRAM.createHeap       = true;
bios.L1DSRAM.enableHeapLabel  = true;
bios.L1DSRAM["heapLabel"]     = prog.extern("L1DHEAP");
bios.L1DSRAM.heapSize         = 0x800;
```

The RTSC module configuration of the application uses and configures the DMAN3 module to make changes to the default DMAN3 configuration settings. For example, the following code snippet from dman3/examples/fastcopytest.cfg assigns the application-defined DSP/BIOS heap “L1DHEAP” to DMAN3.heapInternal, which is defined as the DSP/BIOS heap. It assigns “EXTERNALHEAP” to DMAN3.heapExternal. Then, it assigns the EDMA3 PaRAMs 78 through 125 and TCCs 32 through 63 to the DMAN3 module. It assigns 6 of the 8 available QDMA channels {0, 1, 4, 5, 6, and 7} to DMAN3.

It also sets 16 PaRAMs and 16 TCCs for scratch group 0 and sets 32 PaRAMs and 16 TCCs for scratch group 1.

```
var DMAN3 = xdc.useModule('ti.sdo.fc.dman3.DMAN3');

DMAN3.heapInternal = "L1DHEAP";
DMAN3.heapExternal = "EXTERNALHEAP";

DMAN3.paRamBaseIndex = 78;
DMAN3.numPaRamEntries = 48;
DMAN3.tccAllocationMaskH = 0xffffffff;
DMAN3.tccAllocationMaskL = 0x0;
DMAN3.numTccGroup   = [16, 16, 0, 0, 0, 0];
DMAN3.numPaRamGroup = [16, 32, 0, 0, 0, 0];

DMAN3.qdmaChannels = [0, 1, 4, 5, 6, 7];
DMAN3.maxQdmaChannels = 8;
DMAN3.numQdmaChannels = 6;
```

4.3 DMAN3 Functions

The DMAN3 API provides the following functions:

DMAN3_createChannels	page 32
DMAN3_exit	page 31
DMAN3_freeChannels	page 33
DMAN3_grantDmaChannels	page 29
DMAN3_init	page 31
DMAN3_releaseDmaChannels	page 31

DMAN3 functions perform the following resource management functions:

1. Assign and reclaim PaRAM, QDMA channel, and TCC resources

- DMAN3 *actively* uses the IDMA3 interface to query and grant DMA resources to algorithms (for example, the DMAN3_grantDmaChannels function)
- DMAN3 *passively* provides functions to non-algorithm users to acquire logical and physical DMA resources (for example, the DMAN3_createChannels function)

2. Allocate and free internal memory for the following:

- IDMA3 channel descriptors. Private memory is allocated to hold the handles that are passed back to the algorithm while calling IDMA3::dmaInit().
- Channel private parameter RAM state array to cache persistent DMA transfer settings for each PaRAM entry (memory allocated dynamically in internal RAM)
- Channel private TCC table
- Other channel private state

4.3.1 DMAN3_grantDmaChannels

This function adds one or several algorithms to the DMA Manager. The DMA Manager will grant DMA resources to the algorithms as a result. This function is called when initializing xDAIS algorithm instances. The syntax is:

```
Int DMAN3_grantDmaChannels  (
    Int          groupId,
    IALG_Handle  algHandle[],
    IDMA3_Fxns  *dmaFxns[],
    Int          numAlgs
)
```

Parameters:

The parameters are all input values. The following list describes each one:

groupId The group number for sharing TCCs and PaRAMs. Channels with the same group number get TCCs and PaRAMs allocated from the same scratch group.

Channels created with the same group number in a previous or subsequent call to this function share the same TCCs and PaRAM.

Channels created in this function call are assigned separate TCCs and PaRAMs even though they use the same group ID. When granting DMA resources for several algorithms (that is, when numAlgs is greater than 1), these algorithms do not share TCCs and PaRAMs. If two algorithms can share TCCs and PaRAMs, two calls to DMAN3_grantDmaChannels() are needed.

algHandle[] Array of algorithm handles.

dmaFxns[] Array of IDMA3 Interfaces associated with each algorithm handle.

numAlgs Number of algorithms in algHandle array.

Return values:

DMAN3_SOK	Success.
DMAN3_EOUTOFMEMORY	Failed to allocate logical handle.
DMAN3_EFAIL	Failed to initialize handle, or dmaFxns dmaInit() failed.
DMAN3_EOUTOFTCCS	Not enough TCCs available for channels.
DMAN3_EOUTOFPARAMS	Not enough PaRAMs available for channels.

Preconditions:

- The groupId must be less than DMAN3_MAXGROUPS.
- The IALG_Handle and the IDMA3_Fxns pointer must be valid.
- The IALG_Handle->fxns->implementationId must be the same as the IDMA3_Fxns->implementationId.
- The numberOfChannels returned by the algorithm's dmaGetChannelCnt and dmaGetChannels must not be greater than DMAN3_MAXDMARECS. DMAN3 supports only DMAN3_MAXDMARECS channels.

Postcondition:

- DMAN3_SOK is returned on successful allocation of IDMA3 logical handles.

4.3.2 DMAN3_exit

This is the finalization method for the DMAN module. There are no parameters and no return values. The syntax is:

```
Void DMAN3_exit(Void);
```

4.3.3 DMAN3_init

This is the initialization method for the DMAN module. There are no parameters and no return values. The syntax is:

```
Void DMAN3_init(Void);
```

Parameters:

None

Return Values:

None

Precondition:

- DMAN3_PARAMS must be initialized.

Postcondition:

- DMAN3_SOK is returned on success.

4.3.4 DMAN3_releaseDmaChannels

This function removes logical channel resources from one or several algorithm instances. This function is to be used whenever the DMAN3_grantDMAChannels() function is used to grant DMA resources to the algorithms. The syntax is:

```
Int DMAN3_releaseDmaChannels (
    IALG_Handle  algHandle[],
    IDMA3_Fxns  *dmaFxns[],
    Int          numAlgs
)
```

Parameters:

The parameters are all input values. The following list describes each one:

algHandle[] Array of algorithm handles.

dmaFxns[] Array of IDMA3 Interfaces associated with the algorithm handles.

numAlgs Number of algorithm handles in algHandle[].

Return Values:

DMAN3_SOK	Success.
DMAN3_EOUTOFMEMORY	Failed to allocate logical handle.

Preconditions:

- The algHandle is an array of valid IALG_Handle handles.
- The dmaFxns is an array of valid IDMA3_Fxns pointers.
- The IALG_Handle->fxns->implementationId must be the same as the IDMA3_Fxns->implementationId.
- The numberOfChannels returned by the algorithm's dmaGetChannelCnt and dmaGetChannels must be >= 0 and must not be greater than DMAN3_MAXDMARECS. DMAN3 supports only DMAN3_MAXDMARECS channels.

Postcondition:

- The handle, environment memory, and allocated resources are freed/released.

4.3.5 DMAN3_createChannels

This function allocates and initializes memory for one or several channel handles. This interface is used whenever non-algorithm users would like to request DMA resources from the DMAN3 resource manager. The syntax is:

```
Int DMAN3_createChannels  (
    Int               groupId,
    IDMA3_ChannelRec  dmaTab[],
    Int               numChans
)
```

Parameters:

The parameters are all input values, except dmaTab, which is for both input and output. The following list describes each parameter:

groupId The group number for sharing TCCs and PaRAMs.

Channels created with the same group number in a previous or subsequent call to this function share the TCCs and PaRAMs with the channels created in this call, so they must be used simultaneously.

The channels created in this function call do not share TCCs and PaRAM with each other, even though they use the same group ID. If two channels can share TCCs and PaRAMs, then two calls to DMAN3_createChannels() are needed.

dmaTab *Input:* Array of IDMA3 channel resource descriptors containing the parameters of the requested channels. See Table 2 and Section 3.2.1.

Output: On success, this is assigned the valid and initialized IDMA3 logical DMA handle and environment (env, envSize) fields.

numChans Number of entries in dmaTab (that is, the number of channels to create).

Return Values:

DMAN3_SOK	Success.
DMAN3_EOUTOFMEMORY	Failed to allocate logical handle.
DMAN3_EFAIL	Failed to initialize handle or dmaFxns dmaInit() failed.
DMAN3_EOUTOFTCCS	Not enough TCCs available for channels.
DMAN3_EOUTOFPARAMS	Not enough PaRAMs available for channels.

Preconditions:

- The dmaTab array must have a valid structure.
- The numChans must be greater than or equal to 0.
- The groupId must be less than DMAN3_MAXGROUPS.

Postcondition:

- On success, each handle in dmaTab[] is set to a valid IDMA3_Handle. The “env” field of the IDMA3_Handle is set to newly allocated memory whose size is determined by the IDMA3_InitFxns passed in the dmaTab entry. On failure, the handle of each dmaTab[] entry is set to null.

4.3.6 DMAN3_freeChannels

This function frees memory for an array of channel handles. It is used whenever DMAN3_createChannels was used by a non-algorithm user to request channels. The syntax is:

```
Int DMAN3_freeChannels  (
    IDMA3_Handle  channelTab[],
    Int           numChans
)
```

Parameters:

The parameters are all input values. The following list describes each one:

channelTab	Array of IDMA3 handles representing logical dma channel state.
numChans	Number of entries in channelTab (that is, the number of channels to free).

Return values:

DMAN3_SOK	Success.

Preconditions:

- The numChans must be greater than or equal to 0.
- The channelTab array must provide valid IDMA3 logical channel handles, or numChans must be 0.

Postcondition:

- The “handle” and “env” memory are freed for each channel.

5 ACPY3: Functional DMA Abstraction Layer

The ACPY3 module introduces the notion of a logical DMA channel similar to the ACPY2 interface.

5.1 ACPY3 Functions and Comparison to ACPY2

In summary, the ACPY3 API provides a much lower level of abstraction compared to the ACPY2 interface. ACPY3 is designed to target EDMA3.0/QDMA, while ACPY2 provided a "generic" DMA abstraction layer.

Before submitting a DMA transfer on a logical DMA channel, the channel should be configured for the desired DMA transfer settings via the ACPY3_configure functions.

The fast-configure functions can be used to change a single attribute of the previously configured DMA transfer settings. This configuration approach allows ACPY3 to internally track changed transfer settings and to optimize number of register (PaRAM) writes whenever applicable. Changed settings can be tracked by maintaining a “dirty” bit for each modified field.

The ACPY3_start function is used to submit single or linked EDMA transfers. The ACPY3_start function hides the TCC and PaRAM usage of the QDMA API and provides an easy-to-use interface without sacrificing performance.

5.2 ACPY3 Interface

The ACPY3 API provides the following functions:

ACPY3_activate	page 41
ACPY3_complete	page 40
ACPY3_completeLinked	page 40
ACPY3_configure	page 36
ACPY3_deactivate	page 42
ACPY3_exit	page 42
ACPY3_fastConfigure16b	page 37
ACPY3_fastConfigure32b	page 38
ACPY3_init	page 42
ACPY3_setFinal	page 42
ACPY3_start	page 39
ACPY3_wait	page 39
ACPY3_waitLinked	page 40

5.2.1 Logical Channel Configuration Parameters

Before issuing a DMA transfer, each logical channel must be configured to perform the desired DMA transfer. The ACPY3_Params structure defines the configurable properties of logical DMA channels. Each individual transfer in a Linked Transfer can be configured independently. Assigning a value to the waitId property of a transfer allows the possibility of issuing an ACPY3_waitLinked, which waits until the specified transfer is completed, but does not necessarily wait for the remaining transfers to complete.

The ACPY3_Params structure contains the following DMA transfer parameters. These define the configuration of a logical channel.

ACPY3_TransferType	transferType
Void *	srcAddr
Void *	dstAddr
MdUns	elementSize
MdUns	numElements
MdUns	numFrames
MdInt	srcElementIndex
MdInt	dstElementIndex
MdInt	srcFrameIndex
MdInt	dstFrameIndex
MdInt	waitId

The following list describes these fields:

- **ACPY3_TransferType ACPY3_Params::transferType**
  1D1D, 1D2D, 2D1D or 2D2D
- **Void * ACPY3_Params::srcAddr**
  Source address of the DMA transfer
- **Void * ACPY3_Params::dstAddr**
  Destination address of the DMA transfer
- **MdUns ACPY3_Params::elementSize**
  Number of consecutive bytes in each 1D transfer vector (ACNT)
- **MdUns ACPY3_Params::numElements**
  Number of 1D vectors in 2D transfers (BCNT)
- **MdUns ACPY3_Params::numFrames**
  Number of 2D frames in 3D transfers (CCNT)
- **MdInt ACPY3_Params::srcElementIndex**
  Offset in number of bytes from beginning of each 1D vector to the beginning of the next 1D vector. (SBIDX)
- **MdInt ACPY3_Params::dstElementIndex**
  Offset in number of bytes from beginning of each 1D vector to the beginning of the next 1D vector. (DBIDX)
- **MdInt ACPY3_Params::srcFrameIndex**
  Offset in number of bytes from beginning of 1D vector of current (source) frame to the beginning of next frame's first 1D vector. (SCIDX) Signed value between -32768 and32767.

- **MdInt ACPY3_Params::dstFrameIndex**
  Offset in number of bytes from beginning of 1D vector of current (destination) frame to the beginning of next frame's first 1D vector. (DCIDX) Signed value between -32768 and 32767.

- **MdInt ACPY3_Params::waitId**
  For a single transfer entry, this field is ignored. It is set to 0 (numWaits – 1) internally to track the particular transfer.

  For a linked transfer entry:

  – waitId = -1 : No individual wait on this transfer.

  – 0 < waitId < numWaits : This transfer can be waited on or polled for completion.

  – For the last transfer, this field is ignored. It is internally tracked with a waitId of numWaits – 1.

5.2.2 ACPY3_configure

This function configures all DMA transfer settings for the logical channel. ACPY3_configure must be called at least once for each individual transfer in a logical channel prior to starting the DMA transfer using ACPY3_start. The syntax is:

```
Void ACPY3_configure(
     IDMA3_Handle   handle,
     ACPY3_Params  *params,
     MdInt          transferNo
);
```

Parameters:

The parameters are all input values. The following list describes each one:

handle	IDMA3 channel handle.
params	DMA transfer-specific parameters used to configure this logical DMA channel.
transferNo	Indicates the individual transfer to be configured using the passed “params”.

Return Value:

None

Preconditions:

- The channel must be in an active state.
- The params->waitId must not be numWaits - 0 unless you are configuring the last transfer.
- The IDMA_Obj handle must be valid.
- The handle->protocol field must be set to &ACPY3_PROTOCOL.
- 0 <= transferNo < originally requested number of transfers.

5.2.3 ACPY3_fastConfigure16b

This is a fast configuration function for modifying existing channel settings. Exactly one 16-bit channel transfer property, corresponding to the specified ACPY3_Param field, can be modified. The remaining settings of the channel configuration are unchanged. The syntax is:

```
Void ACPY3_fastConfigure16b(
    IDMA3_Handle         handle,
    ACPY3_ParamField16b  fieldId,
    MdUns                value,
    MdInt                transferNo
);
```

Parameters:

The parameters are all input values. The following list describes each one:

handle	IDMA3 channel handle.
fieldId	Indicates which parameter is to be modified.
value	New value of the parameter to be modified.
transferNo	Indicates which transfer the parameters correspond to. (This is the same value that would be passed to the ACPY3_configure() logical DMA handle and environment (env, envSize) fields.

Enumerated Types:

The following enumerated type is defined for use by the ACPY3 API:

enum ACPY3_ParamField16b

Constant	Value
ACPY3_PARAMFIELD_ELEMENTSIZE	8
ACPY3_PARAMFIELD_NUMELEMENTS	10
ACPY3_PARAMFIELD_ELEMENTINDEX_SRC	16
ACPY3_PARAMFIELD_ELEMENTINDEX_DST	18
ACPY3_PARAMFIELD_FRAMEINDEX_SRC	24
ACPY3_PARAMFIELD_FRAMEINDEX_DST	26
ACPY3_PARAMFIELD_NUMFRAMES	28

Return Value:

None

Precondition:

- The IDMA3_Handle must be valid.
- The algorithm instance must be in an "active" state using the IALG interface.
- The channel must have first been configured with ACPY3_configure.
- The channel must be in an “active” state.

5.2.4 ACPY3_fastConfigure32b

This is a fast configuration function for modifying existing channel settings. Exactly one 32-bit channel transfer property, corresponding to the specified ACPY3_Param field, can be modified. The remaining settings of the channel configuration are unchanged.

Once a channel has been configured once with ACPY3_configure(), ACPY3_fastConfigure32b() can be used to update any of the 32-bit parameter fields, for example, the source address of the data to be transferred.

The syntax is:

```
Void ACPY3_fastConfigure32b(
    IDMA3_Handle         handle,
    ACPY3_ParamField32b  fieldId,
    Uns                  value,
    MdInt                transferNo
);
```

Parameters:

The parameters are all input values. The following list describes each one:

handle IDMA3 channel handle.

fieldId Indicates which of the parameters is to be modified.

value New value of the parameter to be modified.

transferNo Indicates which transfer the parameters correspond to (the same value that would be passed to the ACPY3_configure()). Logical DMA handle and environment (env, envSize) fields.

Enumerated Types:

The following enumerated type is defined for use by the ACPY3 API:

enum ACPY3_ParamField32b

Constant	Value
ACPY3_PARAMFIELD_SRCADDR	4
ACPY3_PARAMFIELD_DSTADDR	12
ACPY3_PARAMFIELD_ELEMENTINDEXES	16
ACPY3_PARAMFIELD_FRAMEINDEXES	24

Return Value:

None

Preconditions:

- The IDMA3_Handle must be valid.
- The algorithm instance must be in an "active" state using the IALG interface.
- The channel must be in an “active” state.

5.2.5 ACPY3_start

This function is called to submit a single or linked DMA transfer. The properties of each individual transfer are the most recent configuration setting for that transfer. The syntax is:

```
Void ACPY3_start(IDMA3_Handle handle);
```

Parameters:

The parameter is an input value. The following list describes it:

handle IDMA3 channel handle.

Return Value:

None

Preconditions:

- The IDMA3_Handle must be valid.
- The channel must be in an “active” state.
- The channel must be in a “configured” state.
- handle->transferPending must be FALSE.

Postcondition:

- handle->transferPending is TRUE.

5.2.6 ACPY3_wait

The ACPY3_wait function performs a polling wait operation, waiting for the completion of *all* DMA transfers issued by the most recent ACPY3_start operation on the given channel handle. ACPY3_wait() uses waitId “numWaits -1” to wait for the completion of all transfers. Therefore, this waitId should not be used to configure any intermediate transfers. This function does not return until all the data transfers on the given channel have completed.

The syntax is:

```
Void ACPY3_wait(IDMA3_Handle handle);
```

Parameters:

The parameter is an input value. The following list describes it:

handle IDMA3 channel handle.

Return Value:

None

Preconditions:

- The IDMA3_Handle must be valid.
- The channel must be in an “active” state.

Postcondition:

- handle->transferPending is FALSE

5.2.7 ACPY3_waitLinked

ACPY3_wait function performs a polling wait operation, waiting for the completion of *an individual* DMA transfer issued by the most recent ACPY3_start operation on the given channel handle. The transfer that gets waited on is the individual transfer that was configured with the associated waitId. This function does not return until the data transfer that was configured with "waited" has completed. The syntax is:

```
Void ACPY3_waitLinked(
    IDMA3_Handle   handle,
    MdUns          waitId
);
```

Parameters:

The parameters are all input values. The following list describes each one:

handle IDMA3 channel handle.

waitId The waitId for the transfer to wait on. This was passed in ACPY3_Params to ACPY3_configure().

Return Value:

None

Preconditions:

- The IDMA3_Handle must be valid.
- The channel must be in an "active" state.
- 0 <= waitId < the originally requested number of waitIds
- The channel must contain one transfer T, which is the i-th transfer, such that i < handle->numPaRams and T is configured with the given waitId.

5.2.8 ACPY3_complete

ACPY3_complete is the non-blocking counterpart of the ACPY3_wait function. It returns true or false depending on whether all transfers have been completed or not, respectively. The online reference API documentation contain more information. The syntax is:

```
Bool ACPY3_complete(IDMA3_Handle handle);
```

Parameters:

The parameter is an input value. The following list describes it:

handle IDMA3 channel handle.

Return Value:

TRUE or FALSE

Preconditions:

- The IDMA3_Handle must be valid.
- The channel must be in an "active" state.

5.2.9 ACPY3_completeLinked

ACPY3_complete is the non-blocking counterpart of the ACPY3_waitLinked function. It returns true or false depending on whether the individual transfer associated with the waitId has been completed or not, respectively. The online reference API documentation contain more information. The syntax is:

```
int ACPY3_completeLinked(IDMA3_Handle handle, MdUns waitId);
```

Parameters:

The parameters are all input values. The following list describes each one:

handle IDMA3 channel handle.

waitId The waitId to check for completion. This was passed in ACPY3_Params to ACPY3_configure().

Return Value:

TRUE or FALSE

Preconditions:

- The IDMA3_Handle must be valid.
- The channel must be in an “active” state.
- 0 <= waitId < the originally requested number of waitIds

5.2.10 ACPY3_activate

ACPY3_activate activates the specified channel. It must be called once before submitting DMA transfers on any IDMA3 channel. The executing task remains the active user of all shared scratch resources, and thus should not be pre-empted by any other task that uses DMA resources created using the same Group ID.

```
Void ACPY3_activate(IDMA3_Handle handle);
```

Parameters:

The parameter is an input value. The following list describes it:

handle IDMA3 channel handle.

Return Value:

None

Precondition:

- The IDMA3_Handle must be valid.
- The algorithm instance must be in an "active" state using the IALG interface.

Postconditions:

- The channel is in an “active” state; any ACPY3 functions can be called using this handle.
- The handle->transferPending is FALSE.

5.2.11 ACPY3_deactivate

ACPY3_deactivate is called to deactivate a channel when a task is ready to relinquish ownership of the shared scratch resources.

```
Void ACPY3_deactivate(IDMA3_Handle handle);
```

Parameters:

The parameter is an input value. The following list describes it:

handle IDMA3 channel handle.

Return Value:

None

Preconditions:

- The IDMA3_Handle must be valid.
- The algorithm instance must be in an "active" state using the IALG interface.
- The channel must be in an “active” state.

Postcondition:

- The channel is in a “deactivated” state.

5.2.12 ACPY3_init

This is the ACPY3 module initialization function.

```
Void ACPY3_init(Void);
```

Parameters:

None

Return Value:

None

5.2.13 ACPY3_exit

This is the ACPY3 module finalization function.

```
Void ACPY3_exit(Void);
```

Parameters:

None

Return Value:

None

5.2.14 ACPY3_setFinal

This function indicates that a given transfer is the last in a sequence of linked transfers. This function can be used to dynamically change the number of transfers in a series of linked transfers. Any waitId previously associated with the transfer "transferNo" is replaced with the waitId “numWaits – 1” since ACPY3_wait() uses this waitId to check for transfer completion.

This function can be used if a channel was created to transfer numTransfers linked transfers, but at some point, it may be that fewer transfers than numTransfers should be started.

```
Void ACPY3_setFinal(
    IDMA3_Handle  handle,
    MdInt         transferNo
);
```

Parameters:

The parameters are all input values. The following list describes each one:

handle IDMA3 channel handle.

transferNo Indicates which transfer will be the last in a set of linked transfers. (This is the same value that was passed to ACPY3_configure().)

Return Value:

None

Preconditions:

- The IDMA3_Handle must be valid.
- The channel must be in an “active” state.

Postcondition:

- The transferNo's waitId is set to “numWaits – 1”.
- Any previously associated waitId with the old "final" transfer is cleared. As a consequence, applications may need to call ACPY3_configure() to restore the appropriate waited.

6 Cache Coherency Issues for Algorithm Consumers

On several C6x1x devices, data that is in both external memory and the L2 cache can cause problems with DMA transfers in several ways.

In Figure 6, memory corresponding to location x has been brought into the L2 cache. The copy in the cache has been modified, but has not yet been written back to external memory. If a DMA transfer copies the data from location x to another location, it would be reading stale data. To avoid this problem, the cache must be flushed before the DMA read proceeds.

Figure 6. DMA Read Access Coherency Problem

In Figure 7, the location x has been brought into the L2 cache. Suppose a DMA transfer writes new data to location x in external memory. In this case, the CPU would access the old cached data in a subsequent read, unless the cached copy is invalidated.

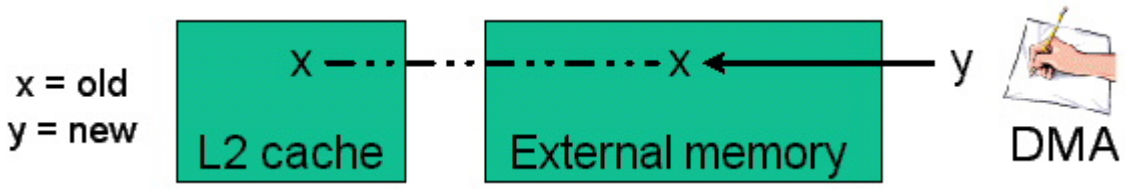


Figure 7. DMA Write Access Coherency Problem

To deal with these coherency problems, several new guidelines and rules have been added to the TMS320 DSP Algorithm Standard. Previously, the xDAIS rules and guidelines for using DMA applied only to algorithms. In Version 2.5 of the TMS320 DSP Algorithm Standard Developer's Kit, the following new guidelines and rules have been added that apply to client applications.

DMA Guideline 3

To ensure correctness, all C6000 algorithms using IDMA2 or IDMA3 need to be supplied with the internal memory they request from the client.

This guideline has been added in conjunction with a new requirement in the xDAIS specification that the client application must inform the algorithm of the type of memory (for example, internal or external) used by each buffer it allocates to the algorithm. The client application does this via the IALG_MemRec structure passed to the algorithm using algInit(). The algorithm can use this information to decide how to respond if it does not receive the type of memory it requests.

DMA Rule 7

If a C6000 algorithm has implemented the IDMA2 or IDMA3 interface, all input and output buffers residing in external memory, and passed to this algorithm through its function calls, should be allocated on a cache line boundary and be a multiple of caches lines in size. The application must also clean the cache entries for these buffers before passing them to the algorithm. (This rule applies to client applications.)

This rule is targeted at the application or client application writer. It ensures that cached entries for buffers passed to the algorithm are flushed to avoid the coherency problem shown in Figure 7. For example, the fastcopytest.c example described in the SPRA789 application note uses the following CSL macro to clean the cache before initializing the data arrays and before performing an algorithm that uses DMA transfers.

```
CACHE_clean(. . .);
```

It is important that input and output buffers be allocated on a cache line boundary and be a multiple of the cache line length in size. As shown in Figure 8, if location x is accessed by the DMA but other data (v) shares the same cache line, the entire cache line may be brought into the cache when v is accessed. Location x would then end up in the cache, which violates the reason behind DMA Rule 6.

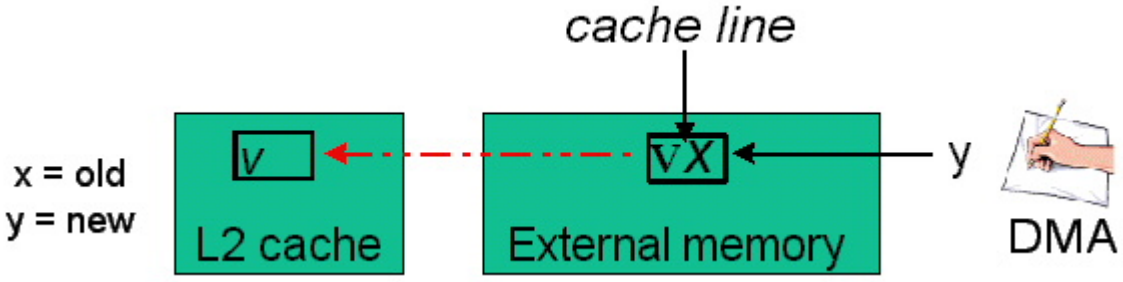


Figure 8. Cache Line Boundaries and the L2 Cache

7 For Algorithm Producers: Creating Algorithms that Use DMA

This section is intended for developers of xDAIS-compliant algorithms that use DMA. After a brief summary and references for the changes introduced by the IDMA3 and ACPY3 requirements, we discuss how algorithm producers do the following:

- Implement the required interfaces to request and receive DMA resources
- Configure DMA channels
- Schedule DMA transfers
- Synchronize with completion status of scheduled transfers

7.1 IDMA3 and ACPY3 Related Changes that affect the Algorithm Developers

Algorithm producers should be aware of a number of changes to the ways they need to develop algorithms that use DMA. These new C6000 DMA guidelines apply to both IDMA2 and IDMA3.

- **IDMA3 interface.** All ‘C64x algorithms that intend to run on ‘C64x+ DSP core devices that require DMA resources must implement the IDMA3 interface. Non-‘C64x+ algorithms may continue to use IDMA2.
- **Configuring channels.** Algorithms should optimize DMA transfers by configuring each channel a single time. Optimized fast configuration functions for configuring channels have been added to the ACPY3 interface.
- **Scratch Group ID.** All physical EDMA3 resources that are granted to different processing threads using the same DMAN3 scratch groupId are **potentially** shared. Such processing threads must not pre-empt each other since the underlying physical resource allocations may be shared.
- **Scheduling.** The ACPY3_start() function is used to schedule a transfer. This function takes a single channel handle argument. This is a change from the deprecated ACPY2_start function.
- **Synchronization.** All linked transfers issued on the same channel start and complete in the same order they were issued. It is possible to wait for all transfers or just an individual transfer in the link chain associated with the configured waitIds.
- **Cache coherency.** Data that is stored in both external memory and the L2 cache can cause problems with DMA transfers in several ways.

7.2 Rules and Guidelines Summary

The TMS320 DSP Algorithm Standard specifies the following rules and guidelines for all C6000 algorithms that request and use DMA resources and for applications that integrate such algorithms. All algorithms designed for ‘C64x+ EDMA3 controller must implement the IDMA3 interface.

- **DMA Rule 1.** All data transfer must be completed before return to caller.
- **DMA Rule 2.** All algorithms using the DMA resource must implement the IDMA2 or IDMA3 interface.
- **DMA Rule 3.** Each of the IDMA2 or IDMA3 methods implemented by an algorithm must be independently relocatable.
- **DMA Rule 4.** [DEPRECATED] All algorithms must state the maximum number of concurrent DMA transfers for each logical channel.
- **DMA Rule 5.** All algorithms must characterize the average and maximum size of the data transfers per logical channel for each operation. Also, all algorithms must characterize the average and maximum frequency of data transfers per logical channel for each operation.

- **DMA Rule 6.** An algorithm using IDMA2 or IDMA3 must not directly access buffers in external memory involved in DMA transfers. This includes the input buffers passed to the algorithm through its function interface.
- **DMA Rule 7.** If an algorithm has implemented the IDMA2 or IDMA3 interface, all input and output buffers residing in external memory and passed to this algorithm through its function calls should be allocated on a cache line boundary and be a multiple of caches lines in size. The application must also clean the cache entries for these buffers before passing them to the algorithm. (Applies to client applications.)
- **DMA Rule 8.** All buffers residing in external memory involved in a DMA transfer should be allocated on a cache line boundary and be a multiple of cache lines in size.
- **DMA Rule 9.** Algorithms should not use stack allocated buffers as source or destination of any DMA transfer.
- **DMA Guideline 1.** The data transfer should complete before the CPU operations executing in parallel.
- **DMA Guideline 2.** Algorithms should request a distinct IDMA2 or IDMA3 logical channel for distinct type of DMA transfer it issues.
- **DMA Guideline 3**. To ensure correctness, all algorithms using IDMA2 or IDMA3 need to be supplied with the internal memory they request from the client application using algAlloc(). (Applies to client applications.)

7.3 Implementing the IDMA3 Interface

A ‘C64x+ algorithm that requests DMA resources must implement the IDMA3 interface. The FCPY_TI algorithm discussed in this application note provides an example implementation of the IDMA3 interface. The following list describes each function that must be implemented and shows an example.

- **dmaChangeChannels.** This function should update the algorithm instance object's persistent memory using the channel descriptors table.

```
/*
 *  ======== FCPY_TI_dmaChangeChannels ========
 *  Update instance object with new logical channel.
 */
Void FCPY_TI_dmaChangeChannels(IALG_Handle handle, IDMA3_ChannelRec dmaTab[])
{
    FCPY_TI_Obj *fcpy = (Void *)handle;

    fcpy->dmaHandle1D1D8B = dmaTab[CHANNEL0].handle;
    fcpy->dmaHandle1D2D8B = dmaTab[CHANNEL1].handle;
    fcpy->dmaHandle2D1D8B = dmaTab[CHANNEL2].handle;
}
```

- **dmaGetChannels.** This function should fill the channel descriptors table (passed by the client application) with the channel characteristics for each logical channel required by the algorithm.

```
/*
 *  ======== FCPY_TI_dmaGetChannels ========
 *  Declare DMA resource requirement/holdings.
 */

Int FCPY_TI_dmaGetChannels(IALG_Handle handle, IDMA3_ChannelRec dmaTab[])
{
    FCPY_TI_Obj *fcpy = (Void *)handle;
    int i;

    /* Initial values on logical channels */
    dmaTab[0].handle = fcpy->dmaHandle1D1D8B;
    dmaTab[1].handle = fcpy->dmaHandle1D2D8B;
    dmaTab[2].handle = fcpy->dmaHandle2D1D8B;

    /*   */
    dmaTab[0].numTransfers = 1;
    dmaTab[0].numWaits = 1;

    dmaTab[1].numTransfers = 1;
    dmaTab[1].numWaits = 1;

    dmaTab[2].numTransfers = 1;
    dmaTab[2].numWaits = 1;

    /*
     * Request logical DMA channels for use with ACPY3
     * AND with environment size obtained from ACPY3 implementation
     * AND with low priority.
     */
    for (i=0; i<NUM_LOGICAL_CH; i++) {
        dmaTab[i].priority = IDMA3_PRIORITY_LOW;
        dmaTab[i].protocol = &ACPY3_PROTOCOL;
        dmaTab[i].persistent = FALSE;    }

    return (NUM_LOGICAL_CH);
}
```

- **dmaGetChannelCnt.** This function should return the number of channels requested by the dmaGetChannels() function.

```
#define NUM_LOGICAL_CH 3

/*
 *  ======== FCPY_TI_dmaGetChannelCnt ========
 *  Return max number of logical channels requested.
 */
Int FCPY_TI_dmaGetChannelCnt(Void)
{
    return(NUM_LOGICAL_CH);
}
```

- **dmaInit.** This function should save the handles for the logical channels granted by the framework in the algorithm instance object's persistent memory.

```
/*
 *  ======== FCPY_TI_dmaInit========
 *  Initialize instance object with granted logical channel.
 */
Int FCPY_TI_dmaInit(IALG_Handle handle, IDMA3_ChannelRec dmaTab[])
{
    FCPY_TI_Obj *fcpy = (Void *)handle;

    fcpy->dmaHandle1D1D8B = dmaTab[CHANNEL0].handle;
    fcpy->dmaHandle1D2D8B = dmaTab[CHANNEL1].handle;
    fcpy->dmaHandle2D1D8B = dmaTab[CHANNEL2].handle;

    return (IALG_EOK);
}
```

7.4 Configuring Logical Channels and DMA Transfers

For every logical DMA channel, before any DMA transfer requests can be submitted to the channel, the algorithm must configure the channel's transfer parameters. Each configurable property characterizes the layout of each DMA transfer block as depicted in Figure 2 (page 10) and corresponds to one of the fields of the ACPY3_Params structure in Table 7 (page 6).

Logical channels always "remember" the most recently applied configuration settings, so additional reconfiguration is unnecessary unless a different type of transfer setting is needed. When a transfer request is submitted, the current channel transfer parameters are recorded and applied when the memory transfer is carried out.

There are several ACPY3 functions to configure the transfer parameters of the logical channel for the type of DMA transfer:

- **Configure-all function:** ACPY3_configure. It takes an ACPY3_Params argument, and replaces the entire channel settings with the new configuration.
- **Fast configuration functions:** ACPY3_fastConfigure16b and ACPY3_fastConfigure32b. Each function selectively updates exactly one parameter of the current configuration.

7.4.1 Performance Considerations

For algorithms that rely heavily on the speed of completion of DMA transfers, minimizing the configuration overhead is extremely critical. Indeed, the addition of new fast configuration functions, the change to the transfer request submission function, ACPY3_start, and the new DMA Guideline 2 all result from performance requirements that were not adequately addressed by the deprecated ACPY specifications.

If a logical channel is configured at the same time a DMA transfer request is submitted, the function incurs a substantial amount of overhead. A straightforward optimization technique is to break the work into two parts: one for channel configuration and the other for transfer request submission. Therein lies the motivation for one of the new guidelines:

DMA Guideline 2

Algorithms should minimize channel (re)configuration overhead by requesting a dedicated logical DMA channel for each distinct type of DMA transfer it issues, and should avoid calling ACPY3_configure. Algorithms should use the fast configuration functions where possible.

DMA Guideline 2 is a useful guideline to follow when different types of DMA transfers are needed in a critical loop of an algorithm. By defining different IDMA3 logical channels for each transfer type, ACPY3_configure() can be called on each channel at the beginning of the algorithm code. Then, transfer requests can be rapidly submitted on these pre-configured channels in the critical loop using ACPY3_start() calls, eliminating re-configuration overhead. By following DMA Guideline 2, the algorithm code can be structured as follows to minimize channel configuration overhead:

```
Void MYALG_TI_process(...) {

    /* Configure the logical channels */
    ACPY3_configure(dma1D1D8bit_handle,  &params1, transferId);
    ACPY3_configure(dma2D2D32bit_handle, &params2, transferId);
    ACPY3_configure(dma1D1D16bit_handle, &params2, transferId);
    ...

    /* Critical loop in the algorithm */
    for (...) {
        ...
        ACPY3_start(dma1D1D8bit_handle); /* channel is configured once */
        ...
   }
    ...
}
```

7.5 Scheduling Asynchronous DMA Transfers on Logical Channels

Algorithms call the ACPY3_start function to submit a request for an asynchronous transfer of a memory block copy from the specified source to the destination memory block. The source and destination addresses need to be properly aligned with respect to the configured element size.

ACPY3_start returns to the caller (the algorithm) as soon as the transfer request is submitted to a logical or physical hardware queue or devices that will asynchronously perform the copy operation.

The exact source and destination memory layout that gets physically copied is determined by the transfer parameters at the time the ACPY3_start call has been issued.

7.5.1 Alignment Issues Using ACPY3_start

ACPY3_start places no restriction on address alignment.

7.6 Synchronizing and Serializing DMA Transfers

An algorithm can submit several transfer requests on each logical channel it owns. Actual physical DMA transfers are started and completed by the hardware resources available to the ACPY3 module.

When an algorithm needs to check the completion status of submitted transfers, it can call a blocking "wait" function, ACPY3_wait(), or a non-blocking "completion status" check function, ACPY3_complete(). The wait or completion status is for the entire channel. That is, the status returned is for the last submitted transfer on that logical channel.

ACPY3 specifications ensure that all transfers issued on the same channel start and complete in the same real-time order as they were issued. The need for synchronizing DMA transfers can be reduced because of these new requirements. This provides additional opportunities to optimize algorithms that use DMA resources.

7.7 Cache Coherency Issues for Algorithm Producers

Algorithms must enforce coherence and alignment/size constraints for internal buffers they request through the IALG interface. In Section 6, Cache Coherency Issues for Algorithm Consumers, page 44, the figures show problems that can occur if coherency between the cache and the external memory are not observed.

To deal with these coherency problems, the following new rules have been added:

DMA Rule 6

C6000 algorithms using IDMA2 or IDMA3 must not directly access buffers in external memory involved in DMA transfers. This includes the input buffers passed to the algorithm through its function interface.

DMA Rule 6 ensures that xDAIS algorithms operate correctly without the need to issue cache clean and flush operations, which are low-level operations that should be dealt with at the client application level. With the introduction of DMA Rule 6, no external buffers involved in DMA transfers end up in the L2 cache, therefore no external coherency problems should occur.

Also remember that the DMA Rule 7, which is targeted at the client application writer, ensures that cached entries for buffers passed into the algorithm are flushed to avoid the coherency problem shown in Figure 7.

It is important that these buffers are allocated on a cache line boundary and are a multiple of caches lines in size. As shown in Figure 9, if for some location x that is accessed by the DMA, there is other data v sharing the same cache line, the entire cache line may be brought into the cache when v is accessed. Location x would then end up in the cache, which violates the purpose of DMA Rule 6.

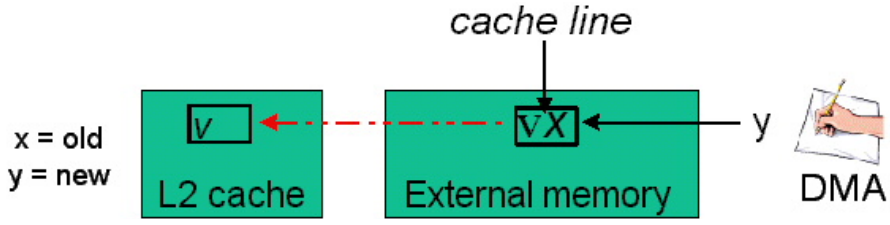


Figure 9. Cache Line Boundaries and the L2 Cache

DMA Rule 8

For C6000 algorithms, all buffers residing in external memory involved in a DMA transfer should be allocated on a cache line boundary and be a multiple of cache lines in size.

DMA Rule 8 is added for algorithm writers who divide buffers supplied to them through their function interface into smaller buffers, and then use the smaller buffers in DMA transfers. In this case, the transfer must also occur on buffers aligned on a cache line boundary. Note that this does not mean the transfer size needs to be a multiple of the cache line length in size. Instead, the “buffer” containing memory locations involved in the transfer must be considered a single buffer; the algorithm must not directly access part of the buffer as per DMA Rule 6.

DMA Rule 9

C6000 algorithms should not use stack allocated buffers as source or destination for any DMA transfer.

DMA Rule 9 is necessary since buffers allocated on the stack are not aligned on cache line boundaries, and there is no mechanism to force alignment. Furthermore, this rule is good practice, as it helps to minimize an algorithm's stack size requirements.

8 The Fast Copy (FCPY) Algorithm Example

In this section we present a toy algorithm, FCPY_TI, used to illustrate how to implement the IDMA3 interface and use DMA via ACPY3 calls. Sample code is provided in Appendix B.

The FCPY algorithm’s doCopy() function illustrates a contrived scenario of copying a 2D buffer from one location in memory to another using DMA. In the process, doCopy() does a 2D to 1D transfer from the source to a work buffer using the parameters [srcLineLen, srcNumLines, srcStride], copies the contents of the work buffer to a second work buffer using a 1D to 1D transfer, and finally copies the contents of the second work buffer to the destination using the parameters [dstLineLen, dstNumLines, dstStride].

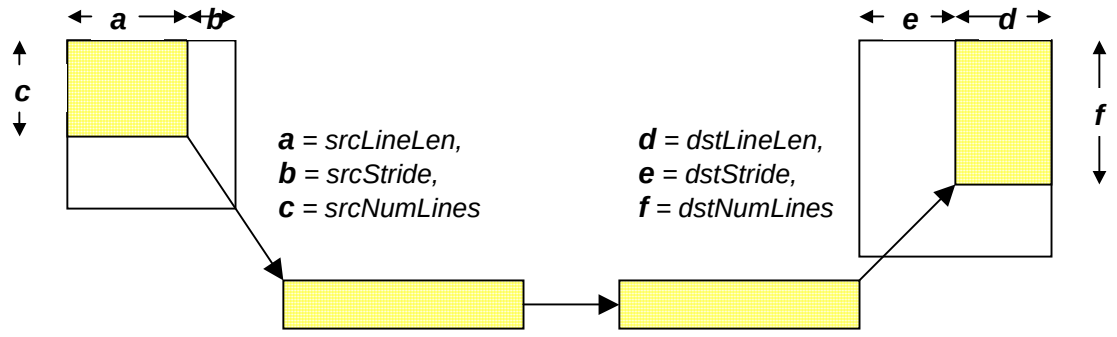


Figure 10. Illustration of FCPY doCopy operation

The FCPY instance object can be configured using the following structure:

```
typedef struct IFCPY_Params {
    Int  size;         /* Size of this structure */
    Int  srcLineLen;   /* Source line length */
    Int  srcNumLines;  /* Number of lines for source */
    Int  srcStride;    /* Stride between lines for source */
    Int  dstLineLen;   /* Destination line length */
    Int  dstNumLines;  /* Number of lines for destination */
    Int  dstStride;    /* Stride between lines for destination */
} IFCPY_Params;
```

Note that srcLineLen * srcNumLines = dstLineLen * dstNumLines must hold true for the algorithm to operate correctly. Otherwise the behavior is undefined.

8.1 IFCPY_Interface Functions

The function table for the algorithm is shown below:

```
typedef struct IFCPY_Fxns {
    IALG_Fxns   ialg;    /* IFCPY extends IALG */
    XDAS_Bool  (*control)(IFCPY_Handle handle, IFCPY_Cmd cmd, IFCPY_Status *status);
    Void       (*doCopy)(IFCPY_Handle handle, Int in[], Int out[]);
} IFCPY_Fxns;
```

In addition to implementing the ialg interface, this algorithm also implements a control (FCPY_TI_control) function that has two commands:

- IFCPY_GETSTATUS: returns the IFCPY_Params structure's non-size parameters
- IFCPY_SETSTATUS: sets the IFCPY_Params structure's non-size parameters

The doCopy function (FCPY_TI_doCopy) is the process function of the algorithm.

8.1.1 Instance Heap Memory Requirements

This algorithm requests three buffers:

Buffer	***Size***	***Alignment***	***Space***	***Attrs***
0	Sizeof(FCPY_TI_Obj)	0	External	Persist
1	(srcLineLen * srcNumLines) * sizeof(Char)	128	Internal	Scratch
2	(srcLineLen * srcNumLines) * sizeof(Char)	128	Internal	Scratch

The alignment of these buffers is set at 128 to align at cache boundaries, so that cache coherence issues do not arise.

8.1.2 The Use of IDMA3 and ACPY3 Interfaces

The IDMA3 interface has been implemented for fcpy_ti to request three different logical channels for the three types of transfers required in the algorithm, following the new DMA performance guideline.

In the algorithm's processing function, FCPY_TI_doCopy, ACPY3 run-time functions are used to show their invocation procedures.

9 The fastcopytest Example

To show how the ACPY3 functions can be used in an actual application, an example, fastcopytest, has been developed. It uses the FCPY_TI algorithm, which follows the new guidelines for achieving high performance. This section describes the example application. Sample code is provided in Appendix A. Code for the fastcopytest Example page 56.

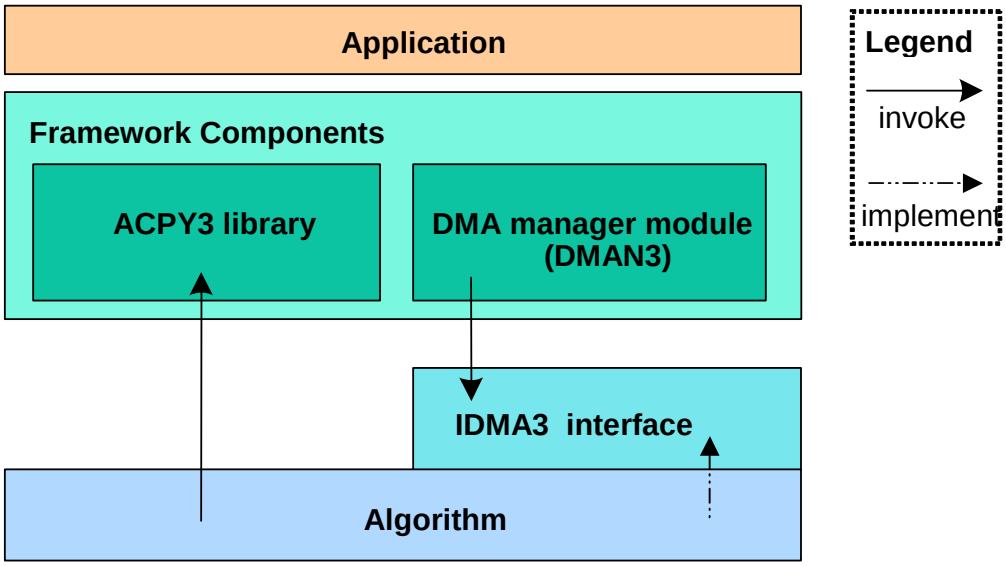


Figure 11. Dependencies in the fastcopytest Example

As Figure 11 shows, to simplify interaction between the IDMA3 interface and the ACPY3 library, a DMA manager module (DMAN3) is used as an extra layer between the algorithm and the application. This module queries the algorithm for its DMA resource needs through the algorithm's IDMA3 interface, allocates the necessary memory for logical channels, and grants the memory to the algorithm.

The example uses the FCPY_TI algorithm, which copies a 2D buffer with a frame index value—the number of 8-bit bytes between the last byte of a row in the 2D buffer and the first byte of its next row—of x into another buffer with a frame index value of y.

Note that this algorithm is also part of the example. It is not performance-driven and aims to show the use of the ACPY3 functions. It first does a 2D to 1D transfer of the data in the input buffer into an internal buffer, then transfers the data to a second internal buffer using a 1D to 1D transfer. Finally, it does a 1D to 2D transfer from the second buffer to the output buffer.

You may also refer to the *TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609), for more details about the DMA-related cache coherence issues.

The fastcopytest application does the appropriate module initializations and uses an instance of the FCPY_TI algorithm to copy a 64x64 block from an input buffer to an output buffer. It divides the block into four quadrants, and makes four separate calls to FCPY_TI to copy the data one quadrant at a time. It prints the word “Pass” in the DSP/BIOS Message Log when all data transfers complete successfully.

10 References

1. *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352)
2. *TMS320 DSP Algorithm Standard API Reference* (SPRU360)
3. *TMS320 DSP Algorithm Standard Developer's Guide* (SPRU424)
4. *TMS320C6000 Peripherals Reference Guide* (SPRU190)
5. *TMS320C621x/C671x EDMA Queue Management Guidelines* (SPRA720)
6. *TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609)

Appendix A. Code for the fastcopytest Example

The following example (fastcopytest.c) uses the FCPY_TI algorithm and the 'C6x1x ACPY3 implementation to illustrate how IDMA3 functions are implemented and how ACPY3 functions are used to perform DMA transfers.

```
/*  ======== fastcopytest.c ========
 *  Test application for FCPY algorithm.  Copies a 2D block from one
 *  location to another in memory, one quadrant at a time.
 */
/* External data sections */
#pragma DATA_SECTION(fcpyInput,".image:ext_sect1")
#pragma DATA_SECTION(fcpyOutput,".image:ext_sect2")
#pragma DATA_ALIGN(fcpyInput,128)   /* aligned on cache boundary */
#pragma DATA_ALIGN(fcpyOutput,128)  /* aligned on cache boundary */

#pragma DATA_ALIGN(srcArr,128)  /* aligned on cache boundary */
#pragma DATA_ALIGN(dstArr,128)  /* aligned on cache boundary */
#pragma DATA_ALIGN(tempMem,128)  /* aligned on cache boundary */
#pragma DATA_ALIGN(scratchMem,128)  /* aligned on cache boundary */

#include <std.h>

#ifndef _RTS_MODE
#include <sys.h>
#include <log.h>
#include <mem.h>
#endif
//#include <csl_cache.h>

#ifndef IDMA3_USEFULLPACKAGEPATH
#define IDMA3_USEFULLPACKAGEPATH
#endif

#include <ti/sdo/fc/utils/api/alg.h>
#include <ti/xdais/ialg.h>
#include <ifcpy.h>

#include <ti/xdais/idma3.h>
#include <ti/sdo/fc/acpy3/acpy3.h>
#include <ti/sdo/fc/dman3/dman3.h>

#include <fcpy.h>

#ifdef _RTS_MODE
#include <bios_rts.h>
#endif

#define SLINELEN  32    /* in bytes */
#define SNUMLINES 32    /* in bytes */
#define SSTRIDE   32    /* in bytes */
#define DLINELEN  32    /* in bytes */
#define DNUMLINES 32    /* in bytes */
#define DSTRIDE   32    /* in bytes */

#define INPUTSIZE   1024        /* in words */
#define OUTPUTSIZE  INPUTSIZE   /* in words */
#define BUFSIZE                         0x800
```

```
/* 2D 64x64 Input and output data buffers */
int fcpyInput[INPUTSIZE];
int fcpyOutput[OUTPUTSIZE];

#ifndef _RTS_MODE

#define LOG_printf0(a, b)       LOG_printf(a, b)
#define LOG_printf1(a, b, c)    LOG_printf(a, b, c)
#define LOG_printf2(a, b, c, d) LOG_printf(a, b, c, d)
extern far Int INTERNALHEAP;
extern far Int EXTMEMHEAP;
extern LOG_Obj LOG_myLog;
extern int L1DHEAP;        /* Heap Label for L1DRAM memory allocation */
extern int EXTMEMHEAP;     /* Heap Label for external memory allocation */

#else

#define LOG_printf0(a, b)
#define LOG_printf1(a, b, c)
#define LOG_printf2(a, b, c, d)

int INTERNALHEAP = 0;
int EXTMEMHEAP =  0;
int EXTERNALHEAP =  0;
int L1DHEAP = 0;

#endif

#define NUMALGS 1

extern far IFCPY_Fxns FCPY_IFCPY;      /* FCPY algorithm's v-table */
extern far IDMA3_Fxns FCPY_IDMA3;      /* FCPY algorithm's IDMA2 v-table */

char srcArr[BUFSIZE];
char dstArr[BUFSIZE];
char tempMem[BUFSIZE];
char scratchMem[BUFSIZE];

void sortInput();
void activateChannels();
void deactivateChannels();
void processInput();
void sendOutput();

/*
 * An example of the use of a non-algorithm user of Framework Components to
 * create IDMA3 channels by directly calling DMAN3 API and subsequently
 * using ACPY3 API to configure and submit DMA transfers.
 */
IDMA3_Handle h;
```

```
void example_nonAlgorithmDMAN3Usage()
{
    int i;

    /* Initialize source and destination arrays for test */
    for (i = 0; i<BUFSIZE; i++) {
         srcArr[i] = i >> 5;
         dstArr[i] = 0;
         tempMem[i] = 0;
         scratchMem[i] = 0;
    }

    activateChannels();

    /* Obtain data from the srcArr sorted into the tempMem array */
    sortInput();
    processInput();
    sendOutput();
    deactivateChannels();
}

void activateChannels()
{

    IDMA3_ChannelRec dmaTab;
    Int status;

    /* Set up the DMAN3 Configurable parameters */
    /* the heap settings of DMAN3 should be set to valid heap descriptors */
    DMAN3_PARAMS.heapInternal = L1DHEAP;
    DMAN3_PARAMS.heapExternal = EXTMEMHEAP;

    /* Set up the DMA Channel descriptor with the transfer parameters  */
    dmaTab.numTransfers = 4;
    dmaTab.numWaits = 4;
    dmaTab.priority = IDMA3_PRIORITY_LOW;

    /* The ACPY3 transfer protocol will be used as the IDMA3 protocol object
     * This object defines the memory requirements and the initialization and
     * de-initialization functions for the protocol's environment */
    dmaTab.protocol = &ACPY3_PROTOCOL;
    dmaTab.persistent = FALSE;

    /*
     * On success this call will return a valid DMA channel handle with the
     * attributes defined above
     */
    status = DMAN3_createChannels(0, &dmaTab, 1);

    if (status == DMAN3_SOK ) {
        h = dmaTab.handle;

        /* Put the channel in active state */
        /* Now other ACPY3 APIs can be called on this handle */
        ACPY3_activate(h);
    }
    else {
        SYS_abort("Channel creeate failed. Status: %d\n", status);
    }
}
```

```
void processInput()
{
    ACPY3_Params p;

    /* Wait for trasnfer with waitId 0 to complete */
    ACPY3_waitLinked(h,0);
    /* Can go ahead and process channel One's data here */

    /* Wait for trasnfer with waitId 1 to complete */
    ACPY3_waitLinked(h,1);

    /* Process channel 2's data here etc.. */

    /* Wait for trasnfer with waitId 2 to complete */
    ACPY3_waitLinked(h,2);

    /* Wait for the entire trasnfer to complete */
    ACPY3_wait(h);

    /* Can process the entire data now */
    /* We simply DMA it from one array to another */

    /* Setting up the parameters for the  transfer (data grp 1) */
    p.transferType = ACPY3_1D1D;
    p.dstAddr = (void *)scratchMem;
    p.srcAddr = (void *)tempMem ;
    p.elementSize = 16 * 16;
    p.numElements = 1;
    p.numFrames = 1;

    /* waitId of 0 implies wait after the first transfer */
    p.waitId = 0;

    /*
     * Configure transfer number 0 on the active DMA handle with the
     * parameters set up above
     */
    ACPY3_configure(h, &p, 0);

    /* Submit the transfer configured on the logical channel handle */
    ACPY3_start(h);

    /* wait for it to finish */
    ACPY3_wait(h);

}
```

```
void sendOutput()
{
    ACPY3_Params p;

    /* Wait for the previous transfer to finish */
    ACPY3_wait(h);

    /* Revert the data back to the form it was received */

    /* Setting up the parameters for the first transfer (data grp 1) */
    p.transferType = ACPY3_2D1D;
    p.dstAddr = (void *)dstArr;
    p.srcAddr = (void *)scratchMem ;
    p.elementSize = 16;
    p.numElements = 4;
    p.numFrames = 1;
    /* No need to set p.srcElementIndex for a 1D2D transfer */
    p.srcElementIndex = 16 * 4;
    p.srcFrameIndex = 1;
    p.dstFrameIndex = 1;

    /* waitId of 0 implies wait after the first transfer */
    p.waitId = 0;

    /*
     * Configure transfer number 0 on the active DMA handle with the
     * parameters set up above
     */
    ACPY3_configure(h, &p, 0);

    /* Setting up the parameters for the second transfer (data grp 2) */
    p.dstAddr = (void *) &dstArr[16 * 4];
    p.srcAddr = (void *) &scratchMem[16];
    p.waitId = 1;

    /* Configure transfer number 1 */
    ACPY3_configure(h, &p, 1);

    /* Setting up the parameters for the third transfer (data grp 3 )*/
    p.dstAddr = (void *) &dstArr[16 * 4 * 2];
    p.srcAddr = (void *) &scratchMem[16 * 2];
    p.waitId = 2;

    /* Configure transfer number 2 */
    ACPY3_configure(h, &p, 2);


    /* Setting up the parameters for the third transfer (data grp 4) */
    p.dstAddr = (void *) &dstArr[16 * 4 * 3];
    p.srcAddr = (void *) &scratchMem[16 * 3];
    p.waitId = -1;

    /* Configure transfer number 2 */
    ACPY3_configure(h, &p, 3);

    /* Submit the transfer configured on the logical channel handle */
    ACPY3_start(h);
}
```

```
/*
 * Deactivate the logical DMA channel
 */
void deactivateChannels()
{
    Int status;

    /* Wait for the transfer to complete */
    ACPY3_wait(h);

    /* deactivate */
    ACPY3_deactivate(h);

    /* Free the channel */
    if ((status = DMAN3_freeChannels(&h, 1)) != DMAN3_SOK ) {
        SYS_abort("DMAN3_freeChannels failed. Status: %d\n", status);
    }
}

/*
 * Assume data in the srcArray is multichannel input data and Sort it.
 * Uses temporary buffer tempArr.
 * srcArray contents of format: 1234123412341234 are sorted
 * tempArray 111122223333444 where 1,2,3,4 represents 16 byte channel data  */
void sortInput() {

    ACPY3_Params p;
    /* Setting up the parameters for the first transfer (data grp 1) */
    p.transferType = ACPY3_1D2D;
    p.dstAddr = (void *)tempMem;
    p.srcAddr = (void *)srcArr ;
    p.elementSize = 16;
    p.numElements = 4;
    p.numFrames = 1;
    /* No need to set p.srcElementIndex for a 1D2D transfer */
    p.dstElementIndex = 16 * 4;
    p.srcFrameIndex = 1;
    p.dstFrameIndex = 1;

    /* waitId of 0 implies wait after the first transfer */
    p.waitId = 0;

    /*
     * Configure transfer number 0 on the active DMA handle with the
     * parameters set up above
     */
    ACPY3_configure(h, &p, 0);

    /* Setting up the parameters for the second transfer (data grp 2) */
    p.transferType = ACPY3_1D2D;
    p.dstAddr = (void *) &tempMem[16];
    p.srcAddr = (void *) &srcArr[16 * 4];
    p.waitId = 1;

    /* Configure transfer number 1 */
    ACPY3_configure(h, &p, 1);
```

```
    /* Setting up the parameters for the third transfer (data grp 3 )*/
    p.transferType = ACPY3_1D2D;
    p.dstAddr = (void *) &tempMem[16 * 2];
    p.srcAddr = (void *) &srcArr[16 * 4 * 2];
    p.waitId = 2;

    /* Configure transfer number 2 */
    ACPY3_configure(h, &p, 2);


    /* Setting up the parameters for the third transfer (data grp 4) */
    p.transferType = ACPY3_1D2D;
    p.dstAddr = (void *) &tempMem[16 * 3];
    p.srcAddr = (void *) &srcArr[16 * 4 * 3];
    p.waitId = -1;

    /* Configure transfer number 2 */
    ACPY3_configure(h, &p, 3);

    /* Submit the transfer configured on the logical channel handle */
    ACPY3_start(h);
}

#ifdef NON_RTSC_CONFIGURATION
static Uns CFG_QDMA_CHANNELS[8] = { 0, 1, 2, 3, 4, 5, 6, 7 };
#endif


Int main(Void)
{
    Int i;
    FCPY_Params fcpyParams;
    FCPY_Handle fcpyAlg;
    IDMA3_Fxns *dmaFxns[NUMALGS];
    IALG_Handle alg[NUMALGS];
    Bool errorFlag = FALSE;
    Int status;


    IFCPY_Fxns * fxns = (IFCPY_Fxns *)&FCPY_IFCPY;

#ifdef NON_RTSC_CONFIGURATION

    DMAN3_PARAMS.heapInternal = L1DHEAP;
    DMAN3_PARAMS.heapExternal = EXTMEMHEAP;

    DMAN3_PARAMS.paRamBaseIndex = 78;
    DMAN3_PARAMS.numPaRamEntries = 48;
    DMAN3_PARAMS.tccAllocationMaskH = 0xffffffff;
    DMAN3_PARAMS.tccAllocationMaskL = 0x0;

    DMAN3_PARAMS.qdmaChannels = CFG_QDMA_CHANNELS;
    DMAN3_PARAMS.maxQdmaChannels = 8;
    DMAN3_PARAMS.numQdmaChannels = 4;
#endif
```

```
    /* Initialize DMA manager and ACPY3 library for XDAIS algorithms
     * and grant DMA resources
     */
    DMAN3_init();
    ACPY3_init();

    example_nonAlgorithmDMAN3Usage();

    FCPY_init();        /* Initialize the framework  */

    /* Set up FCPY Module  param structure */
    fcpyParams = FCPY_PARAMS;
    fcpyParams.srcLineLen = SLINELEN;
    fcpyParams.srcNumLines = SNUMLINES;
    fcpyParams.srcStride = SSTRIDE;
    fcpyParams.dstLineLen = DLINELEN;
    fcpyParams.dstNumLines = DNUMLINES;
    fcpyParams.dstStride = DSTRIDE;

    /* Use the ALG interface to create a new algorithm instance */
    if ((fcpyAlg = FCPY_create(fxns, &fcpyParams)) == NULL) {
        SYS_abort("Could not create algorithm instance");
    }

    alg[0] = (IALG_Handle)fcpyAlg;
    dmaFxns[0] = &FCPY_IDMA3;

    status = DMAN3_grantDmaChannels(0, alg, dmaFxns, NUMALGS);
    if (status != DMAN3_SOK) {
        SYS_abort("Problem adding algorithm's dma resources");
    }

    // CACHE_clean(CACHE_L2ALL, NULL, NULL);

    /* Initialize data arrays */
    for (i = 0; i < INPUTSIZE; i++)
    {
        fcpyInput[i] = i;
        fcpyOutput[i] = 0xDEADBEEF;
    }

    // CACHE_clean(CACHE_L2ALL, NULL, NULL);

    /*
     * Copy input to the output one quadrant at a time
     */

    /* Quadrant 2 */
    FCPY_apply(fcpyAlg, fcpyInput, fcpyOutput);

    /* Quadrant 1 */
    FCPY_apply(fcpyAlg, fcpyInput + (SSTRIDE/4), fcpyOutput + (DSTRIDE/4));

    /* Quadrant 3 */
    FCPY_apply(fcpyAlg, fcpyInput + (INPUTSIZE/2), fcpyOutput + (OUTPUTSIZE/2));

    /* Quadrant 4 */
    FCPY_apply(fcpyAlg, fcpyInput + (INPUTSIZE/2) + (SSTRIDE/4),
                   fcpyOutput + (OUTPUTSIZE/2) + (DSTRIDE/4));
```

```
    /* Verify output */
    for (i = 0; i < OUTPUTSIZE; i++)
    {
        if (fcpyOutput[i] != i) {
            LOG_printf2(&LOG_myLog, " %d th elem in output should not be %d.\n"
                        , i, fcpyOutput[i]);
            errorFlag = TRUE;
        }
    }

    if (errorFlag == FALSE) {
        LOG_printf0(&LOG_myLog, "Pass \n");
    }

    /*
     * Reclaim DMA resources from algorithm and deinitialize the DMA
     * manager and ACPY3 library
     */
    if (DMAN3_releaseDmaChannels(alg, dmaFxns, NUMALGS) != DMAN3_SOK) {
        SYS_abort("Problem removing algorithm's dma resources");
    }

    /* delete the algorithm instance  */
    ALG_delete((IALG_Handle)fcpyAlg);

    /* module finalization */
    DMAN3_exit();
    ACPY3_exit();
    FCPY_exit();    /* Deinitialize the framework */

    return (0);
}
```

Appendix B: Code for FCPY_TI Algorithm

The FCPY_TI algorithm follows the guidelines for achieving high performance.

ifcpy.h

```
/*
 *  ======== ifcpy.h ========
 *  This header defines all types, constants, and functions shared by all
 *  implementations of the FCPY algorithm.
 */
#ifndef IFCPY_
#define IFCPY_

#include <ti/xdais/ialg.h>
#include <ti/xdais/xdas.h>

#ifdef __cplusplus
extern "C" {
#endif /*__cplusplus*/

/*
 *  ======== IFCPY_Obj ========
 *  This structure must be the first field of all FCPY instance objects.
 */
typedef struct IFCPY_Obj {
    struct IFCPY_Fxns *fxns;
} IFCPY_Obj;

/*
 *  ======== IFCPY_Handle ========
 *  This handle is used to reference all FCPY instance objects.
 */
typedef struct IFCPY_Obj *IFCPY_Handle;

/*
 *  ======== IFCPY_Cmd ========
 *  This structure defines the control commands for the FCPY module.
 */
typedef enum IFCPY_Cmd {
    IFCPY_GETSTATUS,
    IFCPY_SETSTATUS
} IFCPY_Cmd;

/*
 *  ======== IFCPY_Params ========
 *  This structure defines the creation parameters for all FCPY instance
 *  objects.
 */
typedef struct IFCPY_Params {
    Int  size;        /* Size of this structure */

    /* The following two parameters are read-only */
    Int  srcLineLen;  /* Source line length (# of 8-bit elements) */
    Int  srcNumLines; /* Number of lines for source */
```

```
    /* The following parameters are read/write */
    Int  srcStride;   /* Stride between lines for source */
    Int  dstLineLen;  /* Destination line length (# of 8-bit elements) */
    Int  dstNumLines; /* Number of lines for destination */
    Int  dstStride;   /* Stride between lines for destination */
} IFCPY_Params;

extern const IFCPY_Params IFCPY_PARAMS; /* default params */

/*
 *  ======== IFCPY_Status ========
 *  This structure defines the parameters that can be changed at runtime
 *  (read/write), and the instance status parameters (read-only).
 */
typedef struct IFCPY_Status {
    Int  size;            /* Size of this structure */

    /* The following two parameters are read-only */
    Int  srcLineLen;      /* Source line length (# of 8-bit elements) */
    Int  srcNumLines;     /* Number of lines for source */

    /* The following parameters are read/write */
    Int  srcStride;       /* Stride between lines for source */
    Int  dstLineLen;      /* Destination line length (# of 8-bit elements) */
    Int  dstNumLines;     /* Number of lines for destination */
    Int  dstStride;       /* Stride between lines for destination */
} IFCPY_Status;

/*
 *  ======== IFCPY_Fxns ========
 *  This structure defines all of the operations on FCPY objects.
 */
typedef struct IFCPY_Fxns {
    IALG_Fxns   ialg;    /* IFCPY extends IALG */
    XDAS_Bool  (*control)(IFCPY_Handle handle, IFCPY_Cmd cmd,
        IFCPY_Status *status);
    Void       (*doCopy)(IFCPY_Handle handle, Void * in, Void * out);
} IFCPY_Fxns;

#ifdef __cplusplus
}
#endif /*__cplusplus*/

#endif  /* IFCPY_ */
```

fcpy_ti.h

```
/*
 *  ======== fcpy_ti.h ========
 *  Interface header for the FCPY_TI module.
 */
#ifndef FCPY_TI_
#define FCPY_TI_

#ifndef IDMA3_USEFULLPACKAGEPATH
#define IDMA3_USEFULLPACKAGEPATH
#endif

#include <ti/xdais/ialg.h>
#include <ifcpy.h>
#include <ti/xdais/idma3.h>
```

```
#ifdef __cplusplus
extern "C" {
#endif /*__cplusplus*/

/*  ======== FCPY_TI_Handle ========
 *  FCPY algorithm instance handle
 */
typedef struct FCPY_TI_Obj *FCPY_TI_Handle;

/*  ======== FCPY_TI_exit ========
 *  Required module finalization function.
 */
extern Void FCPY_TI_exit(Void);

/*  ======== FCPY_TI_init ========
 *  Required module initialization function.
 */
extern Void FCPY_TI_init(Void);

/*  ======== FCPY_TI_IALG ========
 *  TI's implementation of FCPY's IALG interface
 */
extern IALG_Fxns FCPY_TI_IALG;

/*  ======== FCPY_TI_IFCPY ========
 *  TI's implementation of FCPY's IFCPY interface
 */
extern IFCPY_Fxns FCPY_TI_IFCPY;

/*  ======== FCPY_TI_IDMA3 ========
 *  TI's implementation of FCPY's IDMA2 interface
 */
extern IDMA3_Fxns FCPY_TI_IDMA3;

#ifdef __cplusplus
}
#endif /*__cplusplus*/

#endif /* FCPY_TI_ */
```

fcpy_ti_priv.h

```
/*
 *  ======== fcpy_ti_priv.h ========
 *  Internal vendor specific (TI) interface header for FCPY
 *  algorithm. Only the implementation source files include
 *  this header; this header is not shipped as part of the algorithm.
 *
 *  This header contains declarations that are specific to
 *  this implementation and which do not need to be exposed
 *  in order for an application to use the FCPY algorithm.
 */
#ifndef FCPY_TI_PRIV_
#define FCPY_TI_PRIV_
```

```
#include <ti/xdais/ialg.h>
#include <ti/xdais/xdas.h>
#include <ifcpy.h>
#include <ti/xdais/idma3.h>

#ifdef __cplusplus
extern "C" {
#endif /*__cplusplus*/

typedef struct FCPY_TI_Obj {
    IALG_Obj    ialg;        /* MUST be first field of all DAIS algs */
    Int         *workBuf1;   /* on-chip scratch */
    Int         *workBuf2;   /* on-chip scratch */
    Int         srcLineLen;  /* Source line length (# of 8-bit elements) */
    Int         srcNumLines; /* Number of lines for source */
    Int         srcStride;   /* Stride between lines for source */
    Int         dstLineLen;  /* Destination line length (# of 8-bit elements) */
    Int         dstNumLines; /* Number of lines for destination */
    Int         dstStride;   /* Stride between lines for destination */
    IDMA3_Handle dmaHandle1D1D8B;  /* DMA logical channel for 1D to 1D xfers */
    IDMA3_Handle dmaHandle1D2D8B;  /* DMA logical channel for 1D to 2D xfers */
    IDMA3_Handle dmaHandle2D1D8B;  /* DMA logical channel for 2D to 1D xfers */ }
FCPY_TI_Obj;

/* IALG fxn declarations */
extern Void FCPY_TI_activate(IALG_Handle handle);
extern Void FCPY_TI_deactivate(IALG_Handle handle);
extern Int FCPY_TI_alloc(const IALG_Params *algParams, IALG_Fxns **parentFxns,
                         IALG_MemRec memTab[]);
extern Int FCPY_TI_free(IALG_Handle handle, IALG_MemRec memTab[]);

extern Int FCPY_TI_initObj(IALG_Handle handle,
                           const IALG_MemRec memTab[], IALG_Handle parent,
                           const IALG_Params *algParams);

extern Void FCPY_TI_moved(IALG_Handle handle,
                           const IALG_MemRec memTab[], IALG_Handle parent,
                           const IALG_Params *algParams);

/* IFCPY fxn declarations */
extern Void FCPY_TI_doCopy(IFCPY_Handle handle, Void * in, Void * out); extern XDAS_Bool
FCPY_TI_control(IFCPY_Handle handle, IFCPY_Cmd cmd,
                                     IFCPY_Status *status);

/* IDMA3 fxn declarations */
extern Void FCPY_TI_dmaChangeChannels(IALG_Handle handle,
                         IDMA3_ChannelRec dmaTab[]);

extern Uns FCPY_TI_dmaGetChannelCnt(Void);

extern Uns FCPY_TI_dmaGetChannels(IALG_Handle handle,
                         IDMA3_ChannelRec dmaTab[]);

extern Int FCPY_TI_dmaInit(IALG_Handle handle, IDMA3_ChannelRec dmaTab[]);

#ifdef __cplusplus
}
#endif /*__cplusplus*/

#endif  /* FCPY_TI_PRIV_ */
```

fcpy_ti_ialg.c

```
/*
 *  ======== fcpy_ti_ialg.c ========
 *  FCPY Module - TI implementation of the FCPY module.
 *  This file contains the implementation of the required IALG interface.
 */
#pragma CODE_SECTION(FCPY_TI_alloc, ".text:algAlloc")
#pragma CODE_SECTION(FCPY_TI_free, ".text:algFree")
#pragma CODE_SECTION(FCPY_TI_initObj, ".text:algInit")
#pragma CODE_SECTION(FCPY_TI_moved, ".text:algMoved")

#ifndef IDMA3_USEFULLPACKAGEPATH
#define IDMA3_USEFULLPACKAGEPATH
#endif

#include <std.h>
#include <fcpy_ti_priv.h>
#include <ifcpy.h>
#include <ti/xdais/ialg.h>

#define OBJECT 0
#define WORKBUF1 1
#define WORKBUF2 2
#define NUMBUFS 3
#define ALIGN_FOR_CACHE 128  /* alignment on cache boundary */

/*
 *  ======== FCPY_TI_alloc ========
 *  Request memory.
 */
Int FCPY_TI_alloc(const IALG_Params *algParams,
                IALG_Fxns **parentFxns, IALG_MemRec memTab[]) {
    const IFCPY_Params *params = (Void *)algParams;

    if (params == NULL) {
        params = &IFCPY_PARAMS;  /* Use interface default params */
    }

    /* Request memory for FCPY object */
    memTab[OBJECT].size = sizeof (FCPY_TI_Obj);
    memTab[OBJECT].alignment = 0;    /* No alignment required */
    memTab[OBJECT].space = IALG_EXTERNAL;
    memTab[OBJECT].attrs = IALG_PERSIST;

    /* Request memory for working buffer 1 */
    memTab[WORKBUF1].size = (params->srcLineLen) * (params->srcNumLines) *
        sizeof (Char);
    memTab[WORKBUF1].alignment = ALIGN_FOR_CACHE;
    memTab[WORKBUF1].space = IALG_DARAM0;
    memTab[WORKBUF1].attrs = IALG_SCRATCH;

    /* Request memory for working buffer 2 */
    memTab[WORKBUF2].size = (params->srcLineLen) * (params->srcNumLines) *
        sizeof (Char);
    memTab[WORKBUF2].alignment = ALIGN_FOR_CACHE;
    memTab[WORKBUF2].space = IALG_DARAM0;
    memTab[WORKBUF2].attrs = IALG_SCRATCH;

    return (NUMBUFS);
}
```

```
/*  ======== FCPY_TI_free ========
 *  Return a complete memTab structure.
 */
Int FCPY_TI_free(IALG_Handle handle, IALG_MemRec memTab[]) {
    FCPY_TI_Obj *fcpy = (Void *)handle;

    FCPY_TI_alloc(NULL, NULL, memTab);

    memTab[OBJECT].base = handle;

    memTab[WORKBUF1].base = fcpy->workBuf1;
    memTab[WORKBUF1].size = (fcpy->srcLineLen) * (fcpy->srcNumLines) * sizeof (Char);

    memTab[WORKBUF2].base = fcpy->workBuf2;
    memTab[WORKBUF2].size = (fcpy->srcLineLen) * (fcpy->srcNumLines) * sizeof (Char);

    return (NUMBUFS);
}

/*
 *  ======== FCPY_TI_initObj ========
 *  Initialize instance object.
 */
Int FCPY_TI_initObj(IALG_Handle handle,
                const IALG_MemRec memTab[], IALG_Handle parent,
                const IALG_Params *algParams) {
    FCPY_TI_Obj *fcpy = (Void *)handle;
    const IFCPY_Params *params = (Void *)algParams;

    if (params == NULL) {
        params = &IFCPY_PARAMS;  /* Use interface defult params */
    }

    /* Set addresses of internal buffers */
    fcpy->workBuf1 = memTab[WORKBUF1].base;
    fcpy->workBuf2 = memTab[WORKBUF2].base;

    /* Configure the instance object */
    fcpy->srcLineLen  = params->srcLineLen;
    fcpy->srcStride   = params->srcStride;
    fcpy->srcNumLines = params->srcNumLines;
    fcpy->dstLineLen  = params->dstLineLen;
    fcpy->dstStride   = params->dstStride;
    fcpy->dstNumLines = params->dstNumLines;

    return (IALG_EOK);
}

/*  ======== FCPY_TI_moved ========
 *  Re-initialize buffer ptrs to new location.
 */
Void FCPY_TI_moved(IALG_Handle handle,
                const IALG_MemRec memTab[], IALG_Handle parent,
                const IALG_Params *algParams) {
    FCPY_TI_Obj *fcpy = (Void *)handle;

    fcpy->workBuf1 = memTab[WORKBUF1].base;
    fcpy->workBuf2 = memTab[WORKBUF2].base;
}
```

fcpy_ti_idma3.c

```
/*  ======== fcpy_ti_idma3.c ========
 *  FCPY Module - TI implementation of a FCPY algorithm
 *
 *  This file contains an implementation of the IDMA3 interface  */

#pragma CODE_SECTION(FCPY_TI_dmaChangeChannels, ".text:dmaChangeChannels")
#pragma CODE_SECTION(FCPY_TI_dmaGetChannelCnt, ".text:dmaGetChannelCnt")
#pragma CODE_SECTION(FCPY_TI_dmaGetChannels, ".text:dmaGetChannels")
#pragma CODE_SECTION(FCPY_TI_dmaInit, ".text:dmaInit")

#include <std.h>

#include <fcpy_ti_priv.h>
#include <ti/xdais/ialg.h>
#include <ti/xdais/idma3.h>
#include <ti/sdo/fc/acpy3/acpy3.h>

#define NUM_LOGICAL_CH 3

/*
 *  ======== FCPY_TI_dmaChangeChannels ========
 *  Update instance object with new logical channel.
 */
Void FCPY_TI_dmaChangeChannels(IALG_Handle handle, IDMA3_ChannelRec dmaTab[]) {
    FCPY_TI_Obj *fcpy = (Void *)handle;

    fcpy->dmaHandle1D1D8B = dmaTab[0].handle;
    fcpy->dmaHandle1D2D8B = dmaTab[1].handle;
    fcpy->dmaHandle2D1D8B = dmaTab[2].handle;
}

/*
 *  ======== FCPY_TI_dmaGetChannelCnt ========
 *  Return max number of logical channels requested.
 */
Uns FCPY_TI_dmaGetChannelCnt(Void)
{
    return(NUM_LOGICAL_CH);
}

/*
 *  ======== FCPY_TI_dmaGetChannels ========
 *  Declare DMA resource requirement/holdings.
 */
Uns FCPY_TI_dmaGetChannels(IALG_Handle handle, IDMA3_ChannelRec dmaTab[]) {
    FCPY_TI_Obj *fcpy = (Void *)handle;
    int i;

    /* Initial values on logical channels */
    dmaTab[0].handle = fcpy->dmaHandle1D1D8B;
    dmaTab[1].handle = fcpy->dmaHandle1D2D8B;
    dmaTab[2].handle = fcpy->dmaHandle2D1D8B;

    /*   */
    dmaTab[0].numTransfers = 1;
    dmaTab[0].numWaits = 1;

    dmaTab[1].numTransfers = 1;
    dmaTab[1].numWaits = 1;
```

```
    dmaTab[2].numTransfers = 1;
    dmaTab[2].numWaits = 1;

    /*
     * Request logical DMA channels for use with ACPY3
     * AND with environment size obtained from ACPY3 implementation
     * AND with low priority.
     */
    for (i=0; i<NUM_LOGICAL_CH; i++) {
        dmaTab[i].priority = IDMA3_PRIORITY_LOW;
        dmaTab[i].protocol = &ACPY3_PROTOCOL;
        dmaTab[i].persistent = FALSE;
    }

    return (NUM_LOGICAL_CH);
}

/*
 *  ======== FCPY_TI_dmaInit========
 *  Initialize instance object with granted logical channel.
 */
Int FCPY_TI_dmaInit(IALG_Handle handle, IDMA3_ChannelRec dmaTab[]) {
    FCPY_TI_Obj *fcpy = (Void *)handle;

    fcpy->dmaHandle1D1D8B = dmaTab[0].handle;
    fcpy->dmaHandle1D2D8B = dmaTab[1].handle;
    fcpy->dmaHandle2D1D8B = dmaTab[2].handle;

    return (IALG_EOK);
}
```

fcpy_ti_idmavt.c

```
/*
 *  ======== fcpy_ti_idma3vt.c ========
 *  This file contains the function table definitions for the
 *  IDMA3 interface implemented by the FCPY_TI module.
 */
#include <std.h>

#include <ti/xdais/idma3.h>
#include <fcpy_ti.h>
#include <fcpy_ti_priv.h>

/*
 *  ======== FCPY_TI_IDMA3 ========
 *  This structure defines TI's implementation of the IDMA2 interface
 *  for the FCPY_TI module.
 */
IDMA3_Fxns FCPY_TI_IDMA3 = {      /* module_vendor_interface */
    &FCPY_TI_IALG,              /* IALG functions */
    FCPY_TI_dmaChangeChannels,  /* ChangeChannels */
    FCPY_TI_dmaGetChannelCnt,   /* GetChannelCnt */
    FCPY_TI_dmaGetChannels,     /* GetChannels */
    FCPY_TI_dmaInit             /* initialize logical channels */
};
```

fcpy_ti_ifcpy.c

```
/*
 *  ======== fcpy_ti_ifcpy.c ========
 *  FCPY Module - TI implementation of a FCPY algorithm
 *
 *  This file contains the implementation of the IFCPY abstract interface.
 */

#pragma CODE_SECTION(FCPY_TI_doCopy, ".text:doCopy")
#pragma CODE_SECTION(FCPY_TI_control, ".text:control")

#include <std.h>

#include <ti/xdais/xdas.h>
#include <ti/xdais/idma3.h>
#include <ti/sdo/fc/acpy3/acpy3.h>

#include <ifcpy.h>
#include <fcpy_ti_priv.h>
#include <fcpy_ti.h>

/*
 *  ======== FCPY_TI_doCopy ========
 */
Void FCPY_TI_doCopy(IFCPY_Handle handle, Void * in, Void * out) {
    FCPY_TI_Obj *fcpy = (Void *)handle;
    ACPY3_Params params;

    /*
     * Activate Channel  scratch DMA channels.
     */
    ACPY3_activate(fcpy->dmaHandle1D1D8B);
    ACPY3_activate(fcpy->dmaHandle1D2D8B);
    ACPY3_activate(fcpy->dmaHandle2D1D8B);

    /* Configure the logical channel */
    params.transferType = ACPY3_1D1D;

    params.srcAddr = (void *)(fcpy->workBuf1);
    params.dstAddr = (void *)(fcpy->workBuf2);
    params.elementSize = (Uns)(fcpy->srcLineLen) * (fcpy->srcNumLines);
    params.numElements = 1;
    params.numFrames = 1;
    params.srcElementIndex = 0;
    params.dstElementIndex = 0;
    params.srcFrameIndex = 0;
    params.dstFrameIndex = 0;
    params.waitId = 0;

    /* Configure logical dma channel */
    ACPY3_configure(fcpy->dmaHandle1D1D8B, &params, 0);

    /* Configure the logical channel */

    params.transferType = ACPY3_2D1D;
```

```
    params.srcAddr = (void *)in;
    params.dstAddr = (void *)(fcpy->workBuf1);
    params.elementSize = (Uns)(fcpy->srcLineLen);
    params.numElements = fcpy->srcNumLines;
    params.numFrames = 1;
    params.srcElementIndex = fcpy->srcStride + fcpy->srcLineLen ;
    params.dstElementIndex = fcpy->srcLineLen;
    params.srcFrameIndex = 0;
    params.dstFrameIndex = 0;
    params.waitId = 0;

    //params.numFrames = fcpy->srcNumLines;
    //params.srcFrameIndex = fcpy->srcStride;
    //params.dstFrameIndex = 0;

    /* Configure logical dma channel */
    ACPY3_configure(fcpy->dmaHandle2D1D8B, &params, 0);

    /* Configure the logical channel */
    params.transferType = ACPY3_1D2D;
    //params.elemSize = IDMA3_ELEM8;
    //params.numFrames = 0;
    //params.srcFrameIndex = 0;
    //params.dstFrameIndex = 0;

    params.srcAddr = (void *)(fcpy->workBuf2);
    params.dstAddr = (void *)out;
    params.elementSize = (Uns)(fcpy->dstLineLen);
    //params.numElements = fcpy->dstNumLines;
    params.numFrames = 1;
    params.srcElementIndex = fcpy->dstLineLen ;
    //params.dstElementIndex = fcpy->dstLineLen + fcpy->dstStride;

    /* Configure logical dma channel */
    ACPY3_configure(fcpy->dmaHandle1D2D8B, &params, 0);
    ACPY3_fastConfigure16b(fcpy->dmaHandle1D2D8B,
            ACPY3_PARAMFIELD_NUMELEMENTS,
            fcpy->dstNumLines, 0);

    ACPY3_fastConfigure16b(fcpy->dmaHandle1D2D8B,
            ACPY3_PARAMFIELD_ELEMENTINDEX_DST,
            fcpy->dstLineLen + fcpy->dstStride, 0);

    //ACPY2_setNumFrames(fcpy->dmaHandle1D2D8B, fcpy->dstNumLines);
    //ACPY2_setDstFrameIndex(fcpy->dmaHandle1D2D8B, fcpy->dstStride);

    /* Use DMA to fcpy input buffer into working buffer */
    ACPY3_start(fcpy->dmaHandle2D1D8B);
    //ACPY2_start(fcpy->dmaHandle2D1D8B, (Void *)in,
    //    (Void *)(fcpy->workBuf1),
    //    (Uns)(fcpy->srcLineLen));

    /* Check that dma transfer has completed before finishing "processing" */
    while (!ACPY3_complete(fcpy->dmaHandle2D1D8B)) {
        ;
    };
```

```
    /* Use the DMA to copy data from working buffer 1 to working buffer 2 */
    ACPY3_start(fcpy->dmaHandle1D1D8B);
    //ACPY2_start(fcpy->dmaHandle1D1D8B, (Void *)(fcpy->workBuf1),
    ///   (Void *)(fcpy->workBuf2),
    //    (Uns)((fcpy->srcLineLen) * (fcpy->srcNumLines)));

    /* wait for transfer to finish  */
    ACPY3_wait(fcpy->dmaHandle1D1D8B);

    /* Quickly configure NumFrames and FrameIndex values for dmaHandle1D2D8B */
    //ACPY2_setNumFrames(fcpy->dmaHandle1D2D8B, fcpy->dstNumLines);
    //ACPY2_setDstFrameIndex(fcpy->dmaHandle1D2D8B, fcpy->dstStride);

    /* Use the DMA to copy data from working buffer 2 to output buffer */
    ACPY3_start(fcpy->dmaHandle1D2D8B);
    //ACPY2_start(fcpy->dmaHandle1D2D8B, (Void *)(fcpy->workBuf2),
    //    (Void *)out, (Uns)(fcpy->dstLineLen));

    /* wait for all transfers to complete before returning to the client */
    ACPY3_wait(fcpy->dmaHandle1D2D8B);

    /*
     * DeActivate Channel  scratch DMA channels.
     */
    ACPY3_deactivate(fcpy->dmaHandle1D1D8B);
    ACPY3_deactivate(fcpy->dmaHandle1D2D8B);
    ACPY3_deactivate(fcpy->dmaHandle2D1D8B);
}

/*
 *  ======== FCPY_TI_control ========
 */
XDAS_Bool FCPY_TI_control(IFCPY_Handle handle, IFCPY_Cmd cmd, IFCPY_Status *status)
{
    FCPY_TI_Obj *fcpy = (FCPY_TI_Obj *)handle;

    if (cmd == IFCPY_GETSTATUS) {
        status->srcLineLen = fcpy->srcLineLen;
        status->srcNumLines = fcpy->srcNumLines;
        status->srcStride = fcpy->srcStride;
        status->dstLineLen = fcpy->dstLineLen;
        status->dstNumLines = fcpy->dstNumLines;
        status->dstStride = fcpy->dstStride;
        return (XDAS_TRUE);
    }
    else if (cmd == IFCPY_SETSTATUS) {
        /*
         * Note that srcLineLen and srcNumLines cannot be changed once the
         * algorithm has been instantiated, as they determine the sizes of
         * the internal buffers used in FCPY_TI
         */
        fcpy->srcStride = status->srcStride;
        fcpy->dstLineLen = status->dstLineLen;
        fcpy->dstNumLines = status->dstNumLines;
        fcpy->dstStride = status->dstStride;
        return (XDAS_TRUE);
    }

    /* Should not happen */
    return (XDAS_FALSE);
}
```

fcpy_ti_ialg.c

```
/*
 *  ======== fcpy_ti_ialg.c ========
 *  FCPY Module - TI implementation of the FCPY module.
 *  This file contains the implementation of the required IALG interface.
 */
#pragma CODE_SECTION(FCPY_TI_alloc, ".text:algAlloc")
#pragma CODE_SECTION(FCPY_TI_free, ".text:algFree")
#pragma CODE_SECTION(FCPY_TI_initObj, ".text:algInit")
#pragma CODE_SECTION(FCPY_TI_moved, ".text:algMoved")

#ifndef IDMA3_USEFULLPACKAGEPATH
#define IDMA3_USEFULLPACKAGEPATH
#endif

#include <std.h>
#include <fcpy_ti_priv.h>
#include <ifcpy.h>
#include <ti/xdais/ialg.h>

#define OBJECT 0
#define WORKBUF1 1
#define WORKBUF2 2
#define NUMBUFS 3
#define ALIGN_FOR_CACHE 128  /* alignment on cache boundary */

/*  ======== FCPY_TI_alloc ========
 *  Request memory.
 */
Int FCPY_TI_alloc(const IALG_Params *algParams,
                IALG_Fxns **parentFxns, IALG_MemRec memTab[]) {
    const IFCPY_Params *params = (Void *)algParams;

    if (params == NULL) {
        params = &IFCPY_PARAMS;  /* Use interface default params */
    }

    /* Request memory for FCPY object */
    memTab[OBJECT].size = sizeof (FCPY_TI_Obj);
    memTab[OBJECT].alignment = 0;    /* No alignment required */
    memTab[OBJECT].space = IALG_EXTERNAL;
    memTab[OBJECT].attrs = IALG_PERSIST;

    /* Request memory for working buffer 1 */
    memTab[WORKBUF1].size = (params->srcLineLen) * (params->srcNumLines) *
        sizeof (Char);
    memTab[WORKBUF1].alignment = ALIGN_FOR_CACHE;
    memTab[WORKBUF1].space = IALG_DARAM0;
    memTab[WORKBUF1].attrs = IALG_SCRATCH;

    /* Request memory for working buffer 2 */
    memTab[WORKBUF2].size = (params->srcLineLen) * (params->srcNumLines) *
        sizeof (Char);
    memTab[WORKBUF2].alignment = ALIGN_FOR_CACHE;
    memTab[WORKBUF2].space = IALG_DARAM0;
    memTab[WORKBUF2].attrs = IALG_SCRATCH;

    return (NUMBUFS);
}
```

```
/*  ======== FCPY_TI_free ========
 *  Return a complete memTab structure.
 */
Int FCPY_TI_free(IALG_Handle handle, IALG_MemRec memTab[]) {
    FCPY_TI_Obj *fcpy = (Void *)handle;

    FCPY_TI_alloc(NULL, NULL, memTab);

    memTab[OBJECT].base = handle;
    memTab[WORKBUF1].base = fcpy->workBuf1;
    memTab[WORKBUF1].size = (fcpy->srcLineLen) * (fcpy->srcNumLines)
        * sizeof (Char);
    memTab[WORKBUF2].base = fcpy->workBuf2;
    memTab[WORKBUF2].size = (fcpy->srcLineLen) * (fcpy->srcNumLines)
        * sizeof (Char);

    return (NUMBUFS);
}

/*  ======== FCPY_TI_initObj ========
 *  Initialize instance object.
 */
Int FCPY_TI_initObj(IALG_Handle handle,
                const IALG_MemRec memTab[], IALG_Handle parent,
                const IALG_Params *algParams) {
    FCPY_TI_Obj *fcpy = (Void *)handle;
    const IFCPY_Params *params = (Void *)algParams;

    if (params == NULL) {
        params = &IFCPY_PARAMS;  /* Use interface defult params */
    }
    /* Set addresses of internal buffers */
    fcpy->workBuf1 = memTab[WORKBUF1].base;
    fcpy->workBuf2 = memTab[WORKBUF2].base;

    /* Configure the instance object */
    fcpy->srcLineLen  = params->srcLineLen;
    fcpy->srcStride   = params->srcStride;
    fcpy->srcNumLines = params->srcNumLines;
    fcpy->dstLineLen  = params->dstLineLen;
    fcpy->dstStride   = params->dstStride;
    fcpy->dstNumLines = params->dstNumLines;

    return (IALG_EOK);
}

/*  ======== FCPY_TI_moved ========
 *  Re-initialize buffer ptrs to new location.
 */
Void FCPY_TI_moved(IALG_Handle handle,
                const IALG_MemRec memTab[], IALG_Handle parent,
                const IALG_Params *algParams) {
    FCPY_TI_Obj *fcpy = (Void *)handle;
    fcpy->workBuf1 = memTab[WORKBUF1].base;
    fcpy->workBuf2 = memTab[WORKBUF2].base;
 }


asm("_FCPY_TI_IALG  .set _FCPY_TI_IFCPY");
```

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
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
Low Power Wireless	www.ti.com/lpw	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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