DMA Guide for eXpressDSP-Compliant Algorithm Producers and Consumers

Murat Karaorman, Vincent Wan, and Jagadeesh Sankaran

Texas Instruments, Santa Barbara

ABSTRACT

This application note provides an overview of the DMA architecture specified by the TMS320 DSP Algorithm Standard (also known as XDAIS). It also describes two sets of APIs used for accessing DMA resources: the IDMA2 abstract interface and the ACPY2 library.

In addition to providing an overview of fundamental DMA abstractions for eXpressDSP-compliant algorithms, this application note highlights DMA-related enhancements to the Algorithm Standard in Version 2.5 of the TMS320 DSP Algorithm Standard Developer's Kit.

Sections sectios of this application note are provided for producers and consumers of eXpressDSP-compliant algorithms.

Contents

Trademarks are the property of their respective owners.

List of Figures

List of Tables

1 Introduction

The direct memory access (DMA) controller performs asynchronously scheduled data transfers between memory regions without intervention by the CPU. The DMA controller allows movement to and from internal memory, internal peripherals, and external devices to occur in the background, while the CPU continues to execute other instructions in parallel. Algorithms and applications can achieve greater throughput by using DMA to overlap data movement with processing.

However, eXpressDSP-compliant algorithms are not allowed to *directly* access or control the hardware or peripherals, which include the DMA. All system DMA resources are controlled by the client application. The TMS320 DSP Algorithm Standard (also known as XDAIS) specifies interfaces, that when implemented, allow the client application and algorithm to negotiate DMA resources which in turn grants algorithms controlled access to DMA services. To allow an algorithm to schedule DMA transfers, the client application must inquire, from the algorithm during instance creation, about its DMA resource requirements and grant it handles for accessing the DMA. Each granted handle provides the algorithm a uniform, private “logical” DMA channel abstraction. Table 1 summarizes these interfaces. Section 1.2 provides more information.

Table 1. Standard Interfaces Related to DMA Use

	Algorithm	Client Application
Interfaces Implemented	IDMA2 or deprecated IDMA abstract interfaces	ACPY2 or deprecated ACPY concrete interfaces
Implementations Called	ACPY2 (ACPY) to access the logical DMA channel to configure, request, and synchronize data transfers	IDMA2 (IDMA) interface to query and grant logical DMA channels ACPY2 for module initialization and logical channel object size query

The support for DMA was introduced to the XDAIS specification during its first revision in 2000 through the introduction of new DMA rules and two standard interfaces: IDMA and ACPY. TI recently introduced additional rules and guidelines and new enhanced APIs: IDMA2 and ACPY2. These new APIs deprecate the original IDMA and ACPY APIs. Throughout the rest of this application note we will exclusively use the IDMA2 and ACPY2 APIs. The differences between the two sets of APIs are highlighted where applicable and summarized in section 1.2.

Note that the original interface header files: idma.h and acpy.h are still present in the XDAIS Developer’s Kit (Version 2.5) for backward compatibility with existing frameworks and applications. However, these will be gradually phased out as algorithm vendors transition to the higher performance and functionality ACPY2 and IDMA2 interfaces.

1.1 Using This Application Note

In addition to providing an overview of DMA access for eXpressDSP-compliant algorithms, this application note highlights DMA-related enhancements to the Algorithm Standard in Version 2.5 of the TMS320 DSP Algorithm Standard Developer’s Kit. In a separate application note, *Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6x1x* (SPRA789), the authors present a C6x1x-specific (C6211, C6711, and C6416) implementation of the ACPY2 APIs along with an example algorithm and application (including source code) to demonstrate an end-to-end system with algorithms that use DMA.

This application note is intended to assist both of the following groups:

- **Consumers**. Application developers and system integrators who use algorithms that require DMA should read section 2, which includes several sections with information for algorithm consumers.
- **Producers**. Algorithm developers who implement the IDMA2 interface and use ACPY2 calls to perform DMA operations should read the following sections:
  - Section 3, which includes several sections with information for algorithm producers.
  - Appendix A, which presents a complete algorithm to illustrate how to implement IDMA2 functions and use ACPY2 functions to perform DMA transfers.

1.2 Overview of Standard DMA Interfaces

Algorithms must access DMA hardware via the “logical” DMA channel handles which they request and receive from the client application. Algorithms submit DMA transfer requests on these logical channels through the functions provided by the client application.

Two sets of API interfaces are required for accessing DMA resources: IDMA2 and ACPY2.

- **IDMA2.** All algorithms that use DMA resources must implement the IDMA2 interface. This interface allows the algorithm to request and receive handles representing private logical DMA resources.
- **ACPY2**. These functions are implemented as part of the client application and called by the algorithm (and possibly the client application). A client application must implement the ACPY2 interface (or integrate a provided ACPY2 interface) in order to use algorithms that use the DMA resource. The ACPY2 interface describes the comprehensive list of DMA operations an algorithm can access through the logical DMA channels which it acquires through the IDMA2 protocol. The ACPY2 functions allow:
  - Configuring channel DMA transfer parameters
  - Scheduling asynchronous DMA transfers
  - Synchronizing with scheduled transfers (both blocking and non-blocking)

Chapter 6, *Use of the DMA Resource*, in *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352) describes the use of these interfaces. The *TMS320 DSP Algorithm Standard API Reference* (SPRU360) provides details on each function.

Collectively, IDMA2 and ACPY2 describe a flexible and efficient model that greatly simplifies management of system DMA resources and services by the client application and a simple and powerful mechanism for the algorithm to configure and access DMA services.

1.3 Fundamental DMA Abstractions

1.3.1 Logical DMA Channels and DMA Handles

The logical DMA channel is the fundamental DMA abstraction defined in a hardware-independent manner through the introduction of ACPY2 and IDMA2 API specifications. 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. They grant the DMA handles to algorithms, which have requested them using the IDMA2 interface. Algorithms use the DMA handles to call ACPY2 functions in order to configure logical channel settings, to request DMA transfers, or to synchronize with ongoing transfers. The configuration setting of a logical channel is similar to the hardware register settings of a particular physical DMA device. The most recently configured channel settings get applied to each DMA transfer request.

While logical channels provide a uniform DMA resource and service abstraction, in reality systems come with vastly different physical DMA architectures, limitations, and hardware characteristics. It is up to the client application to provide an implementation of the ACPY2 APIs on the target hardware to match the expected performance and functional requirements. It is also up to the client application to seamlessly arrange the sharing of the physical DMA resources among algorithms that request logical channels on which to operate. For example, the ACPY2 library in software can implement queuing behavior even when the underlying physical DMA devices do not have hardware queuing capability.

1.3.2 Queuing of DMA Transfers and Queue IDs

Several outstanding DMA transfer requests may be submitted to a logical channel due to the asynchronous specification of the submit (`ACPY2_start` and `ACPY2_startAligned`) functions. An important new property of the logical channel specification is the strict first-in, first-out (FIFO) ordering by which submitted DMA transfers are carried out.

Therefore, a logical channel can also be seen as implementing a queue for DMA transfer requests. Each logical channel can be seen as an independent queue; however, with the newly introduced *queueId* attribute of a logical channel's descriptor that gets set by the requesting algorithm, the FIFO ordering property further extends to multiple logical channels all sharing the same *queueId*.

1.3.3 Channel Privacy and Synchronization

Algorithms have exclusive ownership of each received logical channel. They can operate safely without fear of external components (other algorithms or other system code) accessing the channel and issuing transfer requests or changing channel configuration settings.

All synchronization calls are issued on a per channel, as opposed to 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.

1.3.4 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 copies the contents of the source memory region to the destination.

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

1. 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 sufficiently in advance, and to perform other useful operations while data is being copied in the background.
2. The physical layout of the source or destination DMA transfer blocks does not have to 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 when the algorithm cannot take advantage of the 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 `ACPY2_start` or `ACPY2_startaligned` function. The *source* and *destination addresses* for the blocks and the *number of elements* in each frame are passed as function arguments. The remaining configuration parameters are the intrinsic properties of the logical channel and are set exclusively by the algorithm calling the ACPY2 configuration functions. The previously configured properties of the logical channel at the time of the transfer request determine the actual memory that gets copied from source to destination. Each DMA transfer is characterized by the following list of configurable attributes. (Figure 1 illustrates the memory layout of a DMA transfer block characterized by these configuration parameters.)

Transfer Type:	1D-to-1D, 1D-to-2D, 2D-to-1D or 2D-to-2D
Element Size:	number of 8-bit bytes per element $\in$ {1, 2, 4},
Number of Frames:	the number of frames in a block, $1 \leq$ number ≤ 65535
Source/Destination Element Index:	the size of the gap between two consecutive elements within a frame plus the element size in 8-bit bytes. When element index is set to zero (0) element indexing is not used
Source/Destination Frame Index:	the size of the gap in 8-bit bytes between two consecutive frames within a block. Defined for 2D transfers only.
Number of Elements:	the number of elements per frame, $1 \leq$ number ≤ 65535
Source/Destination Addresses:	8-bit byte-addresses

The element and frame index parameters are defined independently for both source and destination. If the hardware does not support setting these independently for source and destination, as is the case for the C6x1x EDMA architecture, they must be configured with the same value. Configure functions should indicate error status when any configuration settings are not supported by the client implementation.

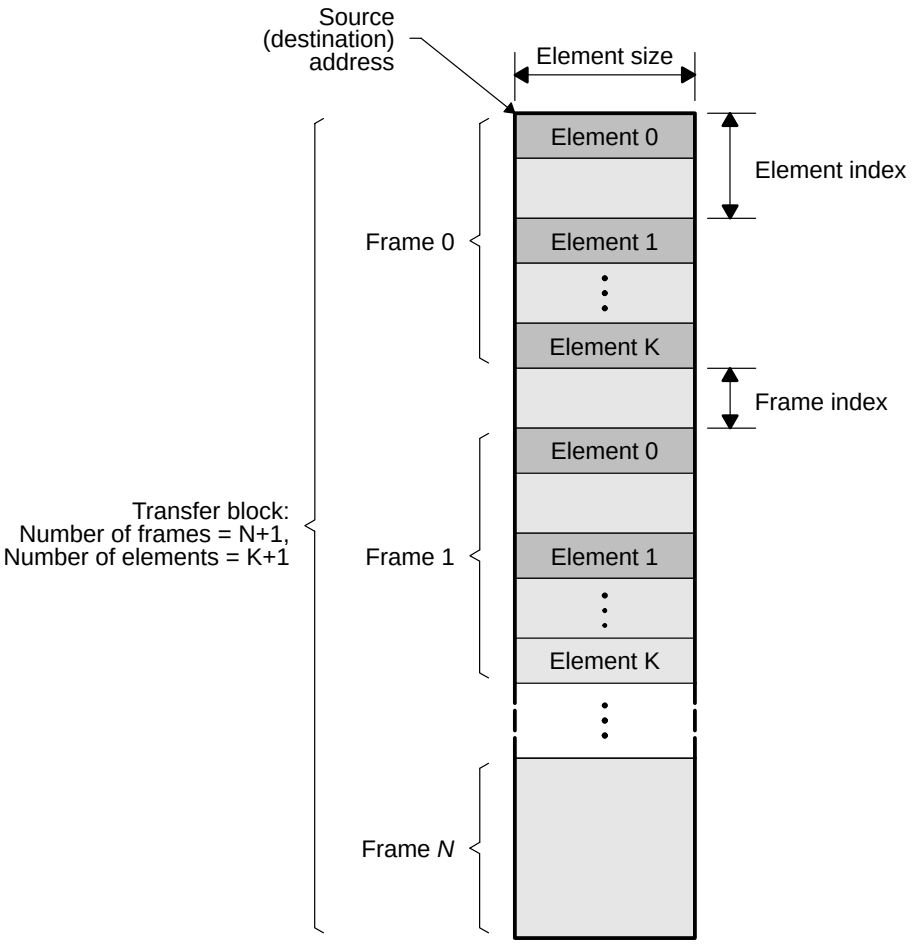


Figure 1. DMA Transfer Block

1.4 Interface Summary

Two sets of standard interfaces are required for accessing DMA resources: IDMA2 and ACPY2. Figure 2 shows which modules are implemented by the client application and which by the algorithm. Arrows indicate which modules use other modules.

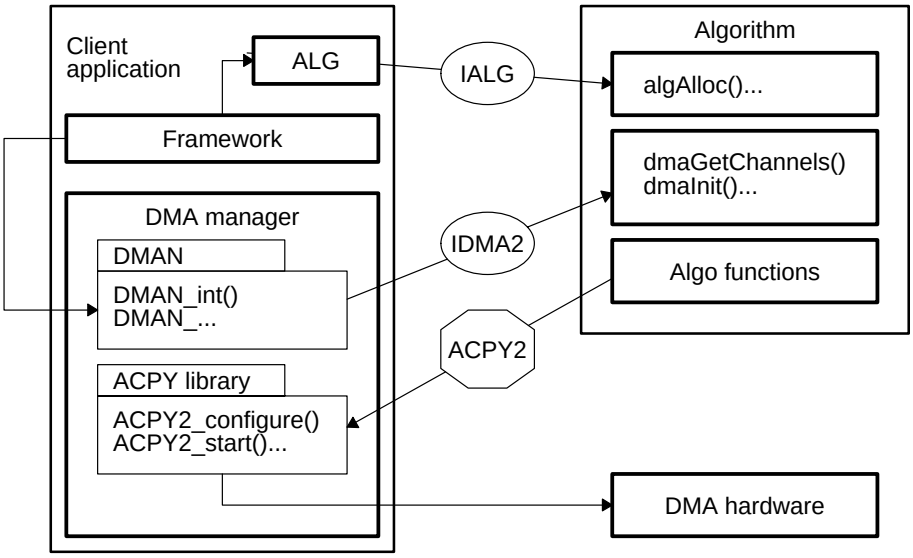


Figure 2. Client Application and Algorithm Interaction with DMA Resources

The client uses the IDMA2 interface to query and grant handles representing private “logical” DMA resources. The algorithm functions call ACPY2 module functions that are provided by the client application to schedule DMA transfers. Like IALG, IDMA2 is an abstract interface accessed through the algorithm module’s IDMA2 functions table. ACPY2, on the other hand, is a concrete interface whose functions are referenced directly.

Client applications may provide and use a general purpose DMA manager module (depicted as DMAN in Figure 2) for granting DMA resources to algorithms, designed as a wrapper around the IDMA2 interface. A reference DMAN module has been developed in the companion application note, SPRA789, and may be used for this purpose.

The following tables summarize the API functions and structures used by the IDMA2 and ACPY2 interfaces and identify items that are new in v2.5 of the XDAIS Developer’s Kit. The new items are described in more detail in this application note. The first column in each table indicates how the specific function relates to the corresponding deprecated APIs.

Table 2. IDMA Functions

vs. IDMA	Functions	Description
	dmaChangeChannels	Called by an application whenever logical channels are moved at runtime.
	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 handles to the algorithm during initialization.

Table 3. ACPY Functions

vs. ACPY	Functions	Description	
	ACPY2_complete	Check if the data transfers on a specific logical channel have completed	
	ACPY2_configure	Configure a logical channel	
New	*ACPY2_exit*	Free resources used by the ACPY2 module	***FRAMEWORK API***
New	*ACPY2_getChanObjSize*	Get the size of the IDMA2 channel object	***FRAMEWORK API***
New	*ACPY2_init*	Initialize the ACPY2 module	***FRAMEWORK API***
New	*ACPY2_initChannel*	Initialize the IDMA2 channel object passed in	***FRAMEWORK API***
New	*ACPY2_setNumFrames*	Rapidly configure the numFrames parameter of an IDMA2 channel	
New	*ACPY2_setSrcFrameIndex*	Rapidly configure the source frame index parameter of IDMA2 channel	
New	*ACPY2_setDstFrameIndex*	Rapidly configure the destination frame index parameter of IDMA2 channel	
Change	*ACPY2_start*	Issue a request for a data transfer using current channel settings	
New	*ACPY2_startAligned*	Issue a request for a data transfer using current channel settings (assumes aligned addresses)	
	ACPY2_wait	Wait for all data transfers to complete on a specific logical channel	

Table 4. IDMA2_Params Structure Fields

vs. IDMA	Structures	Description
	xType	Transfer type: 1D1D, 1D2D, 2D1D or 2D2D
	elemSize	Element transfer size {1,2, or 4 bytes}
	numFrames	Num of frames for 2D
New	*srcElementIndex*	Gap + elemSize between consecutive elements in source data (in 8-bit bytes)
New	*dstElementIndex*	Gap + elemSize between consecutive elements in destination data (in 8-bit bytes)
New	*srcFrameIndex*	Jump between source data frames for 2D transfers (in 8-bit bytes)
New	*dstFrameIndex*	Jump between destination data frames for 2D transfers (in 8-bit bytes)

Table 5. IDMA2_ChannelRec Structure Fields

vs. IDMA	Structures	Description
Removed	*depth*	
Removed	*dimension*	
	handle	Handle to logical DMA channel
New	*queueId*	Selects the serialization queue

1.5 What's New in IDMA2 and ACPY2?

This application note highlights the following DMA-related enhancements that were introduced to the Algorithm Standard in Version 2.5 of the TMS320 DSP Algorithm Standard Developer's Kit:

- New enhanced IDMA2 and ACPY2 APIs deprecate the current IDMA and ACPY APIs.
- New guidelines for high performance. These guidelines allow vendors to extract maximum performance benefits when developing algorithms that use the DMA heavily. (See sections 3.2, 3.3, 3.4, 3.5, and 3.6.)
- Submitted transfers on each logical channel are now performed in FIFO/serial order. This diminishes the need for `ACPY2_wait()` synchronization when scheduling back-to-back transfers from and to the same buffer. (See sections 2.3.3 and 3.6.)
- A new "*queueId*" property for logical channels. In addition to the FIFO rule above, transfers submitted to separate channels sharing the same queueId must also complete sequentially. The ACPY2 functions can take advantage of queueIds to map logical channels to separate physical DMA hardware devices or queues. (See sections 2.3.3 and 3.6.)
- New ACPY2 functions for optimized configuration and optimized data transfer requests. (See section 2.2.)
- New rules and guidelines for external memory access and buffer alignments for specific C6000 devices where cache coherence between the L2 cache and external memory is not directly supported in hardware when simultaneous CPU and DMA accesses exist. (See sections 2.3.2 and 3.7.)
- New "*frame index*" (*formerly stride*) for supporting separate source and destination strides for 2D transfer blocks.
- New "*element index"* field for source and destination for DMA transfer blocks.
- Removed *depth* and *dimension* attributes from DMA channel descriptors.

1.5.1 C6000 Specific Changes

- New rules and guidelines for external memory access and buffer alignments for devices where cache coherence between the L2 cache and external memory is not directly supported in hardware, when simultaneous CPU and DMA accesses exist.

1.5.2 C5000 Specific Changes

- Changed to using 8-bit byte addressing for source & destination addresses.
- Support for *extended memory* addressing capability for DMA transfers.
- [C55x only] New rules for addressing automatic endianism conversion issues.
- [C55x only] New guidelines for enhancing performance through supporting packed/burst modes of operation.

2 For Algorithm Consumers: Integrating Algorithms that Use DMA

Algorithm consumers should be aware of a number of changes to the ways they need to integrate algorithms that use DMA:

- **ACPY2 Interface**. Several new ACPY2 functions should be implemented as part of the ACPY2 module. See section 2.2.
- **Rules and Guidelines**. A rule and a guideline that apply to algorithm consumers have been added to the XDAIS specification. See section 2.3.2.
- **Serialization and Queueids**. All transfers submitted to the same channel or queueId must now start and complete in the same order. See section 2.3.3.
- **C6000 Specific Issue: Cache Coherency**. Data that is stored in both external memory and the L2 cache can cause problems with DMA transfers in several ways. See section 2.3.2.
- **C5000 Specific Issue**. New alignment, size, and data access rules for C55x algorithms are introduced in section 2.3.

2.1 Integrating Algorithms that Use DMA Resources

The steps for integrating an algorithm that uses DMA resources will differ depending upon your system needs and specific DMA management policy. The following steps offer a simple and generic example that can be used to set up and use an algorithm that requests DMA resources. These steps can be incorporated into the client's DMA manager module that is used to grant DMA resources to algorithms.

1. Implement or include an ACPY2 library in the application. The ACPY2 module **must** implement all the functions specified by the XDAIS specification. This is a non-trivial step and is beyond the scope of this app note to discuss how to implement ACPY2 APIs. The companion application note presents the details of the design and implementation of an ACPY2 library for C6x1x which is shipped along with that application note.
2. Initialize the algorithm by calling its initialization function, and set values for fields in the algorithm-specific params structure, which is declared in i<mod>.h.

```
IMOD_Params modParams;
...
MOD_init();
/*
 * Use default instance creation parameters
 */
modParams = IMOD_PARAMS;
```

3. Use the standard IALG interface to allocate and grant the memory buffers requested by the algorithm and initialize the instance object.

```
IALG_Handle algHandle;
...
if ((algHandle=ALG_create((IALG_Fxns *)MOD_VEND_IALG, NULL, (IALG_Params
*)&modParams)) == NULL) {
    SYS_abort("could not create algorithm instance");
}
```

4. Obtain a reference to the module's IDMA2 functions table. Call module's dmaGetChannelCnt() to get the number of logical DMA channels needed by the algorithm.

```
IDMA2_Fxns *dmaFxns = &MOD_VEND_IDMA2;
Int numChan;
...
numChan = dmaFxns->dmaGetChannelCnt();
```

5. Allocate space on a heap or stack for the channel descriptors used in IDMA2 calls.

```
IDMA2_ChannelRec *dmaTab = (IDMA2_ChannelRec *) malloc(numChan *
sizeof(IDMA2_ChannelRec) );
```

6. Call dmaGetChannels() to get the DMA channel properties requested by the algorithm.

```
numChan = dmaFxns->dmaGetChannels(algHandle, dmaTab);
```

7. Call ACPY2_getChanObjSize() to get the size of the channel object (a structure representing the logical channels). The size of each channel object is implementation-dependent and part of the implementation of the ACPY2 library. The library must provide the size information to the client application.

```
Int chanObjSize = ACPY2_getChanObjSize();
```

8. Allocate the memory for each logical channel object.

```
dmaTab[i].handle = (IDMA2_Handle)malloc(chanObjSize);
```

9. Call ACPY2_initChannel() to initialize each channel object.

```
ACPY2_initChannel(dmaTab[i].handle, dmaTab[i].queueId);
```

10. Call dmaInit() to pass the DMA channels to the algorithm.

```
if (dmaFxns->dmaInit(algHandle, dmaTab) == IALG_EOK) {
    return (TRUE);  // DMA init success
}
```

2.2 New ACPY2 Module APIs for Frameworks

A number of functions have been added to the ACPY2 interface specification for the client application's use. These functions are intended to provide application frameworks a way to standardize generation of logical DMA channel handles.

In a nutshell all that the application needs to do to generate a DMA handle is to allocate and assign the memory for DMA handle and call the channel initialization function. The memory assigned to the handle represents the private channel state data and is managed entirely by the DMA manager's ACPY2 library.

- `extern Void ACPY2_init(Void);`

  Initialize the ACPY2 module.

- `Void ACPY2_initChannel(IDMA2_Handle handle, Uns qid);`

  Initialize the IDMA2 channel object passed in (`handle`). The queue id(`qid`) parameter is used by the ACPY2 library to ensure FIFO completion of DMA transfers submitted to all channels sharing the same id. Depending on the DMA Resource Manager policy, either the same value from the algorithm, or some other consistent internal mapping of that value can be passed in `APCY2_initiChannel`.

- `Uns ACPY2_getChanObjSize(Void);`

  Get the size of the IDMA2 channel object. Application uses the size information to allocate the space for the DMA handle.

- `extern Void ACPY2_exit(Void);`

  Free resources used by the ACPY2 module.

Refer to the *TMS320 DSP Algorithm Standard API Reference* (SPRU360) for details about these functions.

2.3 6000 Specific Issues for Algorithm Consumers

2.3.1 ACPY2 Implementation Provided for C6x1x

A C6x1x-specific implementation of the ACPY2 library is provided with the companion application note. This application note provides a function-by-function description of its implementation. The attachment provides the optimized library.

By retaining configuration and resource states with each logical channel descriptor, ACPY2 functions can be implemented very efficiently. This C6x1x implementation demonstrates how this can be done.

2.3.2 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. Please refer to *TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609), for more details. In this section we summarize the relavant issues that might affect algorithm consumers.

In Figure 3, 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.

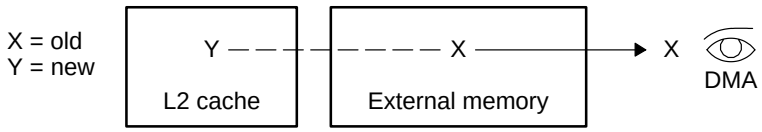


Figure 3. DMA Read Access Coherency Problem

In Figure 4, the location x has been brought into the L2 cache. Suppose a DMA transfer writes new data to location x. In this case, the CPU would access the old cached data in a subsequent read, unless the cached copy is invalidated.

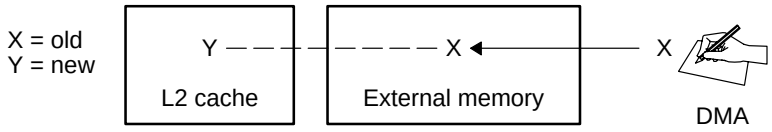


Figure 4. DMA Write Access Coherency Problem

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

DMA Guideline 3

To ensure correctness, All C6000 algorithms that implement IDMA2 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 states 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 interface, the client must allocate all the required at cache line boundary. These buffers must be a multiple of cache line length in size. The client must also clean the cache entries for these buffers before passing them to the algorithm.

This rule is targeted to the application or client application writer. It ensures that cached entries for buffers passed into the algorithm are flushed to avoid the coherency problem shown in Figure 4. For example, the fastcopytest.c example described in the companion application note, *Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6x1x* (SPRA789), 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 5, 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.

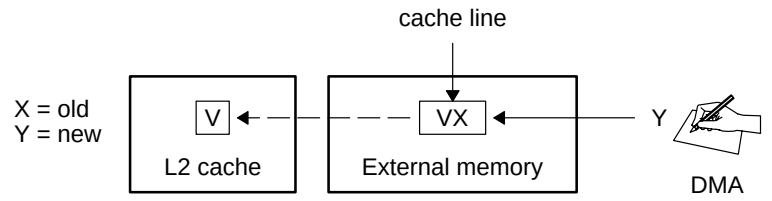


Figure 5. Cache Line Boundaries and the L2 Cache

2.3.3 Serialization and QueueIds for ACPY2

A new requirement is that all transfers submitted to the same channel by `ACPY2_start()` or `ACPY2_startAligned()` must start and complete in the same real-time order as they were issued. Additionally, transfers submitted to separate channels sharing the same queueId must also complete sequentially. For existing C6000 DMA devices based on EDMA, all DMA transfers are submitted to a hardware queue and this requirement is implicitly met. For the same stated reasons, implementing queueId policy is also easily met. For sophisticated DMA managers/ACPY2 libraries, the queueId field can be utilized transparent to the algorithms, to assign different priority levels to improve hardware parallelism.

2.4 C55x Specific Issues for Algorithm Consumers

2.4.1 Supporting packed/burst mode DMA Transfers

The performance studies on the C55/OMAP 1510 platform identified the need to perform DMA transfers in burst enabled/packed transfer modes as much as possible. In this mode it is possible to achieve speedup by a factor of 3 to 4 for transfers from internal to external memory and a factor of 2 to 3 speedup for external to internal transfers. DMA Guideline 4 below is introduced to transparently assist ACPY2 library implementations on the OMAP C55x platforms to perform transfers with burst enabled/packed mode. As a corollary, client applications are advised also to follow the same guideline both for application side data, and any application buffers that are passed as arguments to the algorithm's processing functions.

DMA Guideline 4

To facilitate high performance, C55x algorithms should request DMA transfers with source and destinations aligned on 32-bit byte addresses.

Additionally, where configuration options exist, the DMA manager should be set to operate in packed/burst mode.

2.4.2 Addressing Automatic Endianism Conversion Issues

New alignment, size, and data access rules govern C55x algorithms to ensure correct operation in the presence of possible automatic endianism conversion. The issue is treated in greater detail for the algorithm producers in a later section. Application developers should also follow the same rules and guidelines.

3 For Algorithm Producers: Creating Algorithms that Use DMA

This section is intended for developers of eXpressDSP-compliant algorithms that use DMA. Following a brief summary and references for the changes introduced by the IDMA2 and ACPY2 requirements, we discuss how algorithm producers implement the required interfaces to request and receive DMA resources, configure DMA channels, schedule DMA transfers, and synchronize with completion status of scheduled transfers.

3.1 IDMA2 and ACPY2 Related Changes that Affect the Algorithm Developers

Algorithm producers should be aware of a number of changes to the ways they must develop algorithms that use DMA:

- **Rules and guidelines**. A number of rules and guidelines have been added for algorithms that use DMA resources. For a summary, see section 3.2.
- **IDMA2 interface**. All algorithms that require DMA resources must implement the IDMA2 interface. See section 3.3.
- **Configuring channels**. Algorithms should optimize DMA transfers by configuring each channel a single time. Optimized functions for configuring channels have been added to the ACPY2 interface. See section 3.4.
- **Scheduling**. The `ACPY2_start()` or `ACPY2_startAligned()` function is used to schedule a transfer. The ability to configure transfer parameters as an optional argument to the same scheduling function is not supported by this API function. This is a change from the deprecated ACPY_start API. See section 3.5.
- **Synchronization**. All transfers issued on the same channel must now start and complete in the same order they were issued. Additionally, separate channels sharing the same *queueId* must also complete sequentially. See section 3.6.
- **C6000 Specific: Cache Coherency**. Data that is stored in both external memory and the L2 cache can cause problems with DMA transfers in several ways. See section 3.7.1.
- **C5000 Specific: Extended and 8-Bit Byte Addressing**. The source and destination addresses in ACPY2_start or ACPY2_startAligned APIs are defined using a newly introduced type, IDMA2_AddrPtr, with a 32-bit representation on the C5000 targets. See section 3.8.
- **C55x Specific: Endianism Support**. New alignment, size, and data access rules help to ensure correct operation in the presence of possible automatic endianism conversion. Rules allow general-purpose C55x algorithms to be deployable in OMAP based target DSPs.

3.2 Rules and Guidelines Summary

The previous version of the TMS320 DSP Algorithm Standard specified the following rules and guidelines for algorithms that request and use DMA resources.

- **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 interface.
- **DMA Rule 3**. Each of the IDMA2 methods implemented by an algorithm must be independently relocateable.
- **DMA Rule 4**. 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 Guideline 1**. The data transfer should complete before the CPU operations executing in parallel.

The following rules and guidelines are new in version 2.5 of the TMS320 DSP Algorithm Standard Developer's Kit for algorithms that use DMA resources and for applications that integrate such algorithms.

- **DMA Guideline 2**. Algorithms should request a distinct IDMA2 logical channel for a distinct type of DMA transfer it issues.
- **DMA Guideline 3**. To ensure correctness, all algorithms using IDMA2 need to be supplied with the internal memory they request from the client application using algAlloc(). (Applies to client applications.)
- **DMA Rule 6**. An algorithm using IDMA2 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 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 Rule 10.** C55x algorithms must request all data buffers in external memory with 32-bit alignment and sizes in multiples of 4 bytes.
- **DMA Rule 11.** C55x algorithms must use the same data type and access mode when reading, writing, or transferring data that is stored in external memory or in application-passed data buffers.

3.3 Implementing the IDMA2 Interface

An algorithm that requests DMA resources must implement the IDMA2 interface. The FCPY_TI algorithm discussed in this application note provides an example implementation of the IDMA2 interface. The following list describes each function that must be implemented and shows an example.

- **dmaChangeChannels() function**. 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, IDMA2_ChannelRec
dmaTab[])
{
    FCPY_TI_Obj *fcpy = (Void *)handle;
    fcpy->dmaHandle1D1D8B = dmaTab[CHANNEL0].handle;
    fcpy->dmaHandle1D2D8B = dmaTab[CHANNEL1].handle;
    fcpy->dmaHandle2D1D8B = dmaTab[CHANNEL2].handle;
```

```
}
```

- **dmaGetChannels() function**. 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, IDMA2_ChannelRec dmaTab[])
{
    FCPY_TI_Obj *fcpy = (Void *)handle;
    /* Initial values on logical channels */
    dmaTab[CHANNEL0].handle = fcpy->dmaHandle1D1D8B;
    dmaTab[CHANNEL1].handle = fcpy->dmaHandle1D2D8B;
    dmaTab[CHANNEL2].handle = fcpy->dmaHandle2D1D8B;
    /* Want all transfers to be serialized (to simplify debugging) */
    dmaTab[CHANNEL0].queueId = 0;
    dmaTab[CHANNEL1].queueId = 0;
    dmaTab[CHANNEL2].queueId = 0;

    return (NUM_LOGICAL_CH);
}}
```

- **dmaGetChannelCnt() function**. 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() function**. 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, IDMA2_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);
}
```

3.4 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 graphically in Figure 1 and corresponds to one of the fields (shown in parenthesis here) of the `IDMA2_Params` structure in Table 4.

Transfer Type (xType):	1D-to-1D, 1D-to-2D, 2D-to-1D or 2D-to-2D
Element Size (elemSize):	number of 8-bit bytes per element $\in$ {1, 2, 4},
Number of Frames (numFrames):	number of frames in a block, 1 $\leq$ number $\leq$ 65535
Element Indexes (srcElementIndex or dstElementIndex):	size of the gap between two consecutive elements within a frame *plus* the element size in 8-bit bytes. When element index is set to zero (0) element indexing is not used
Frame Indexes (srcFrameIndex or dstFrameIndex):	the size of the gap in 8-bit bytes between two consecutive frames within a block. Defined for 2D transfers only.

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 ACPY2 functions to configure the transfer parameters of the logical channel for the type of DMA transfer:

- Configure-all function: `ACPY2_configure`. It takes an `IDMA2_Params` argument, and replaces the entire channel settings with the new configuration.
- Fast configuration functions: `ACPY2_setNumFrames, ACPY2_setSrcFrameIndex, ACPY2_setDstFrameIndex`. Each function selectively updates the number of frames, source frame index and destination frame index parameters of the current configuration, respectively.

3.4.1 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 APIs, the change to the transfer request submission APIs, ACPY2_start and ACPY2_startAligned, and the new DMA Guideline 2, all result from the performance requirements that were not adequately addressed by the deprecated ACPY specifications.

To illustrate the cost of configuration operations refer to Figure 6 that shows performance figures for an implementation of the ACPY/ACPY2 library on the EDMA device of C6x1x processors. These numbers were gathered on the C6711 DSK using Code Composer Studio 2.1's profiler clock, and by setting breakpoints before and after the function calls to observe the change in the clock's cycle count. The three most important numbers at this point of the discussion are the ones for ACPY_start() when passed a non-null IDMA_Params structure, ACPY2_start(), ACPY2_startAligned, and ACPY2_configure(). Also note that "DAT" calls refer to the DMA support module of the chip support library (CSL) and their peformance figures are included for reference.

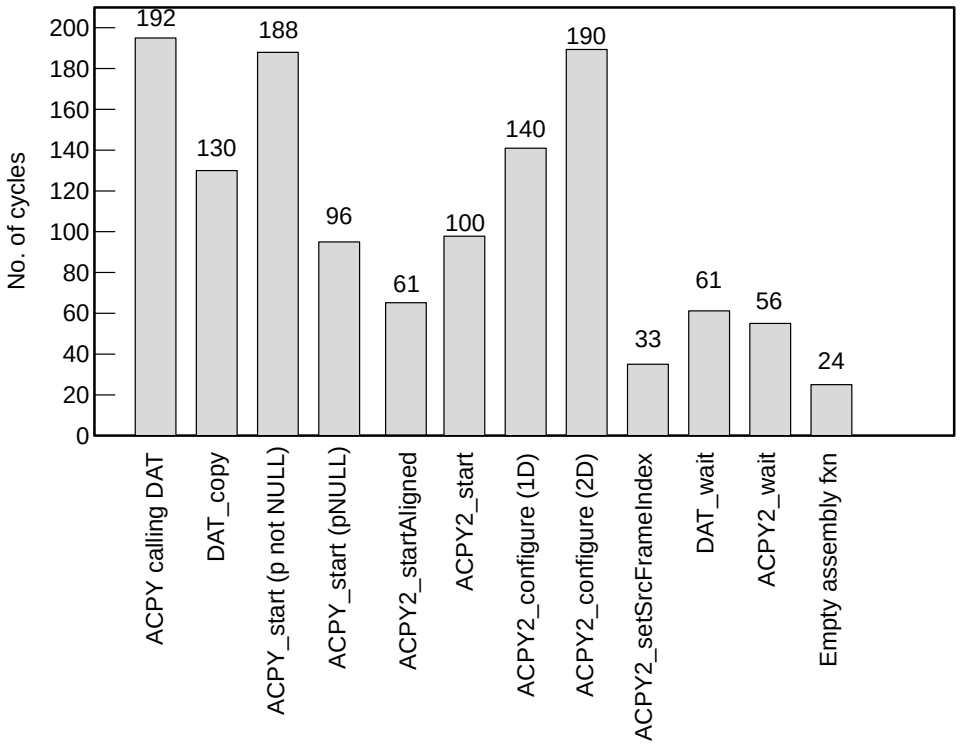


Figure 6. Performance Figures for ACPY/ACPY2 Implementation on C6711 DSK

In the old ACPY specifications, the high number of cycles for ACPY_`start()` with a non-null IDMA_Params shows that 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 avoid calling ACPY2_configure. Algorithms should use the fast configuration APIs where possible.

DMA Guideline 2 is useful to follow when different types of DMA transfers are needed in a critical loop of an algorithm. By defining different IDMA2 logical channels for each transfer type, ACPY2_`configure()` can be called on each channel at the beginning of the algorithm code. Then, transfer requests can be rapidly submitted on these preconfigured channels in the critical loop using ACPY2_`start()` calls, eliminating reconfiguration overhead.

Using ACPY2_`startAligned()` cuts the DMA transfer request submission overhead from 188 cycles to 61 cycles, approximately a 67% reduction in cycles. This improvement is particularly significant when requests are submitted within a loop. By following the DMA Guideline 2 the algorithm code can be structured as follows to minimize channel configuration overhead:

```
Void MYALG_TI_process(...) {
    /* Configure the logical channels */
```

```
    ACPY2_configure(dma1D1D8bit_handle,  &params1);
    ACPY2_configure(dma2D2D32bit_handle, &params2);
    ACPY2_configure(dma1D1D16bit_handle, &params2);
    ...
    /* Critical loop in the algorithm */
    for (...) {
        ...
        ACPY2_startAligned(dma1D1D8bit_handle, ... ); // channel already con
                                                         figured
        ...
   }
    ...
}
```

3.5 Scheduling Asynchronous DMA Transfers on Logical Channels

Algorithms call the ACPY2_start() or ACPY2_startAligned() function to submit a request for an asynchronous transfer of a memory block copy from the specified source to the destination memory block. The only operational difference between ACPY2_startAligned() and ACPY2_start() is the additional requirement by ACPY2_startAligned() for its source and destination addresses to be properly aligned with respect to the configured element size.

Both functions return 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 ACPY2_start() or ACPY2_startAligned() call has been issued.

3.5.1 Using ACPY2_start

ACPY2_start() makes no assumptions on the alignment of the source and destination addresses and indexes. It accepts addresses and indexes at any alignment and when allowed by the architecture, such as the C6000, and adjusts the transfer parameters (including element size, number of elements, transfer type) to transparently perform the desired transfer using the given alignment, if necessary. This API is provided in the specifications with the intention to simplify algorithm development in the initial stages. ACPY2_start() thus strives to maintain simplicity while maintaining reasonable levels of performance.

3.5.2 Using ACPY2_startAligned

The ACPY2_startAligned() API, on the other hand, expects the source and destination addresses and indexes to be properly aligned with respect to the configured element size. When using 32-bit transfer mode, these addresses must be 32-bit aligned. For 16-bit transfers, 16-bit alignment is required. Passing source or destination addresses/indexes with incorrect alignment with respect to the configured element size of the DMA handle will result in unspecified behavior. In this respect the sole aim of ACPY2_startAligned() is to guarantee performance by eliminating run-time checks by a prenegotiated contract with the algorithm developer.

3.5.3 Algorithm Design Considerations

There are cases which are well served by ACPY2_startAligned and there are cases which are well served by ACPY2_start. Good examples of cases that are served well by ACPY2_startAligned are constant buffer rate DMA scenarios with input buffering of bitstreams for processing and creating output frames in external memory. Such transfers allow for full utilization of DMA bandwidth as everything is known about the transfer upfront.

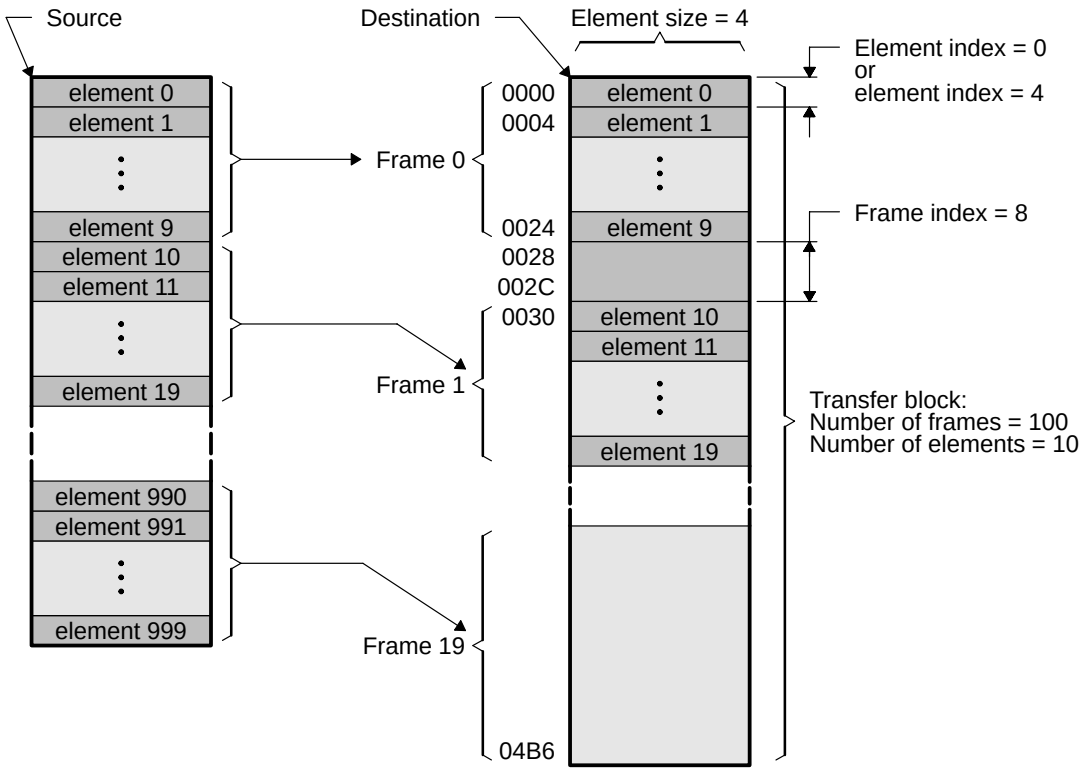


Figure 7. 1D-to-2D DMA Transfer Example

Figure 7 illustrates the source and destination memory layout of a typical 1D-to-2D DMA transfer example. The following code fragment is also used to perform the depicted transfer.

```
MYALG_TI_Obj *myAlg = (Void *)handle;  /* myAlg points to algorithm instance
                                          object */
IDMA2_Params params;
/* Configure Transfer Parameters */
params.xType = IDMA2_1D2D;
params.elemSize = IDMA2_ELEM32;
params.numFrames = 100;
params.srcFrameIndex = 8;       /* Not used in 1D2D transfer */
params.dstFrameIndex = 8;
params.srcElementIndex = params.dstElementIndex = 0;

/* Configure logical dma channel */
ACPY2_configure(myAlg->dmaHandle1, &params);
```

```
/*
 * Schedule DMA transfer from input buffer, in, to instance working buffer,
 * workBuf1. Set number of elements in each destination frame to 10.
 */
ACPY2_start(myAlg->dmaHandle1, (IDMA2_AdrPtr)in, (IDMA2_AdrPtr)
(myAlg->workBuf1), 10);
```

3.6 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 ACPY2 module.

When an algorithm needs to check the completion status of submitted transfers, it can call a blocking wait function, `ACPY2_wait()`, or a non-blocking completion status check function, `ACPY2_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.

ACPY2 specifications ensure that all transfers issued on the same channel start and complete in the same real-time order as they were issued. Additionally, separate channels sharing the same queueId also complete sequentially. 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.

Consider the following pseudocode, which implements a double buffering scheme for an in-place algorithm process:

```
ACPY2_start(h0, src++, buf[0]);           // copy in a buffer
foreach (pair of src and dst buffers) {
    ACPY2_start(h1, src++, buf[1]);      // copy in a buffer
    ACPY2_wait(h0);                      // wait for buffer to be ready
    Process(buf[0]);                     // do the work
    ACPY2_start(h0, buf[0], dst++);     // copy result out.
        // Need ACPY2_wait without serialization guarantee
    ACPY2_start(h0, src++, buf[0]);     // copy in a buffer
    ACPY2_wait(h1);                      // wait for buffer to be ready
    Process(buf[1]);                    // do the work
    ACPY2_start(h1, buf[1], dst++);     // copy result out.
        // Need ACPY2_wait without serialization guarantee
}
ACPY2_wait(h0);
ACPY2_wait(h1);
```

This code was not guaranteed to run correctly under the previous XDAIS specifications because there was no guarantee that the `ACPY2_start()` call on line 6 would complete before the call on line 8. Hence, depending on the implementation, the second transfer request might have been serviced before the first one, causing buf[0] and buf[1] to be overwritten with new data from the source buffer before the processing results were written to the destination buffers.

One solution using the deprecated ACPY specifications is to insert an ACPY_wait call on lines 7 and 12 to wait for handle 0 and handle1 to complete their previous transfers before submitting the next transfer. However, this means the algorithm has to wait for results to be copied out before it can process new data that has been brought in through the double buffering scheme, thereby missing the goal of double buffering.

Therefore, it is necessary for the XDAIS specification to guarantee that transfers submitted on the same logical channel are handled serially. However, even this FIFO ordering guarantee is insufficient in some cases, as in the following code:

```
ACPY2_start(h0, src++, buf[0]);           // copy in a buffer
foreach (pair of src and dst buffers) {
    ACPY2_start(h1, src++, buf[1]);       // copy in a buffer
    ACPY2_wait(h0);                      // wait for buffer to be ready
    Process(buf[0]);                     // do the work
    ACPY2_start(h2, buf[0], dst++);      // copy result out.
        // Need ACPY2_wait without serialization guarantee
    ACPY2_start(h0, src++, buf[0]);      // copy in a buffer
    ACPY2_wait(h1);                     // wait for buffer to be ready
    Process(buf[1]);                    // do the work
    ACPY2_start(h3, buf[1], dst++);     // copy result out.
        // Need ACPY2_wait without serialization guarantee
}
ACPY2_wait(h0);
ACPY2_wait(h1);
```

Here, two new logical channels (h2 and h3) are defined to copy results out to the destination buffer. This kind of situation is more likely when following the new DMA Guideline 2 to define multiple channels for different transfer types. Therefore, a mechanism is needed to allow the algorithm to specify that transfers on h0 and h2 (or h1 and h3) are serviced serially.

To this effect, the new field queueId has been added to the IDMA2_ChannelRec structure:

```
typedef struct IDMA2_ChannelRec {
    IDMA2_Handle  handle;   // Handle to logical DMA channel
    Int           queueId;  // Selects the serialization queue
} IDMA2_ChannelRec;
```

IDMA2 channels sharing the same queueId should have their transfers serialized by the ACPY2 library. This queueId is passed by the client application to the ACPY2 library using the new API ACPY2_initChannel.

3.7 C6000 Specific Issues for Algorithm Producers

3.7.1 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 2.3.2, the figures show problems that can occur if coherency between the cache and the external memory are not observed. Please refer to *TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609), for more details. In this section we summarize the relavant issues that might affect algorithm developers using DMA.

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

DMA Rule 6

C6000 algorithms using IDMA2 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 will end up in the L2 cache, therefore no external coherency problems would 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 4.

It is important that these buffers are allocated on a cache line boundary and be a multiple of caches lines in size. As shown in Figure 8, 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.

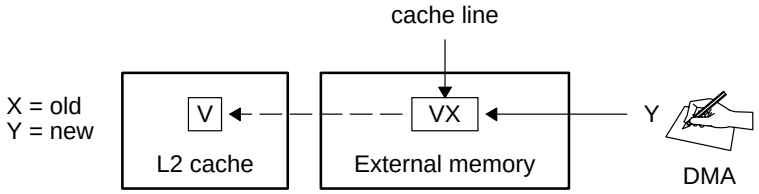


Figure 8. Cache Line Boundaries and the L2 Cache

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 of 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 the minimize an algorithm’s stack size requirements.

3.8 C5000 Specific Issues for Algorithm Producers

3.8.1 Source and Destination Types Require 32-Bit Extended Byte Addressing

The deprecated ACPY_start specification utilized a data-address type, `Void *`, for the source and destination address arguments. This imposed a restriction on the C5000 applications to pass only 23-bit word addresses, since C5000 standard C data pointers are 23-bit word addresses. While this is a general limitation when using data pointers with the standard C programming environment on C5000, it is especially limiting when an algorithm needs to request DMA transfers for data in external memory in the extended address space. Full 32-bit addressing capability is needed for designing and implementing algorithm interfaces which may be designed to use application buffers passed using 32-bit addresses

With the ACPY2 specification, a new type, `IDMA2_AddrPtr`, with a 32-bit representation on the C5000 targets has been introduced to address the ACPY limitations:

```
typedef Void (*IDMA2_AdrPtr)();
```

The new type is used to declare the source and destination addresses in ACPY2_start API. The function prototype for the new ACPY2_start API is:

```
Void ACPY2_start(IDMA2_Handle, handle, IDMA2_AdrPtr src, IDMA2_AdrPtr dst,
Uns cnt);
```

3.8.2 Source and Destination Addresses Use Byte Addressing

The deprecated declaration of *`Void *` data type* for source and destination addresses in ACPY_start API restricts the C5000 applications to pass only 23-bit word addresses. This effectively disallows passing any odd byte-addresses for source or destination of DMA transfers, even though most existing DMA hardware devices use byte addresses for source and destination registers.

With the API change for ACPY2_start, on C5000 targets, algorithms will need to explicitly pass byte addresses doing the proper typecasting and a 1-bit left shifting of any data pointer. A macro to simplify the conversion is also found in the IDMA2 specification.

```
#define IDMA2_ADRPTR(addr) ((IDMA2_AdrPtr)((LgUns)addr<<1))
```

The following code snippet illustrates how the macto can be utilized by an algorithm developer:

```
ACPY2_start(obj->handle1,IDMA2_ADRPTR((Int *)in+offset),
                         IDMA2_ADRPTR((Int *)out), numElems);
```

3.8.3 C55x Issue: Supporting Packed/Burst Mode DMA Transfers

On the C55/OMAP 1510 platform algorithms and applications need to perform DMA transfers in burst enabled/packed transfer modes as much as possible since this leads to speedup factors of 3 to 4 for transfers from Internal to external memory and factors of 2 to 3 speedup for external to internal transfers.

Packing:

When enabled by the software, packing packs several consecutive element transfers into wider accesses. For example, if element size is 16 bits, the 32-bit wide SARAM port can pack two accesses so that 4 bytes at a time are written into the channel FIFO. This can reduce overhead and improve channel throughput. Packing options are determined by port access capabilities and element size.

For the hardware to do successful packing:

- For source/destination packing, start/destination address has to be aligned on the data type boundary (for 16/32-bit transfer, the address should be aligned on a 16/32-bit boundary).
- If frame index or element index is used for single- or double-indexed addressing, the resulting address should be aligned on the data type boundary.

Bursting:

Bursting is an extension of the packing concept. A burst is defined as 16 bytes. For successful bursting to occur, again, the source/destination addresses need to be aligned on the burst8 (16-bit) or burst8 (32-bit) boundary for 16-bit or 32-bit data transfer, respectively.

DMA Guideline 4 transparently assists *ACPY2 library implementations* on the OMAP C55x platforms to support transfers in burst enabled/packed mode.

DMA2 Guideline 4

To facilitate high performance, all C55x algorithms should request DMA transfers with source and destinations aligned on the data type boundary. Also, if they use frame index or element index, the resulting address must be ensured that it is aligned on the data type boundary.

3.8.4 C55x Issue: Addressing Automatic Endianism Conversion

3.8.4.1 Summary of hardware endianism treatment by of (OMAP1510) MGS3 DMA

The GDMA ports (SARAM, DARAM, EMIF, RHEA, and API) can have different endianisms. The GDMA must adapt the endianism of the data read in the source port in order to match its internal FIFO endianism. Before writing out data, the GDMA must adapt the endianism of the data in its internal FIFO to match the destination port endianism.

MGS3 DMA supports *on the fly Endianism Conversion*, during DMA transfer between DSP internal memory (SARAM/DARAM) and EMIF, which is controlled by software on DMA channel basis. The DMA endianism conversion insures the scalar preservation, based on the data type. Therefore, the parameters of DMA transfer, for example: start address, element index, and frame index must be chosen according to this constraint.

3.8.4.2 New Endianism Support Requirements and Rules

This automatic endianism conversion takes place during *any* 32-bit access of the external memory through its EMIF port. This affects DMA or CPU read/writes alike. The only mechanism the h/w offers for disabling the auto-conversion, (only for the DMA operations) is to initiate the transfer in non-packed/non-burst mode. Operating in non-packed/non-burst mode is otherwise not desirable since the performance penalty is huge, so the original requirement was stated to use pack/burst modes as default channel configuration but have the ability to configure a channel to disable pack/burst.

Further complications to the problem arise when data in external memory is not 32-bit aligned or transfer size is not in multiples of 4 (32-bits): the h/w does **not** perform endianism conversion in these cases. Additionally, this behavior is not restricted to just DMA operations but normal 32-bit CPU read/writes are also subject to the same laws. New DMA Rule 10 ensures the automatic endianism conversion by the DMA controller, while moving data between internal and external memory. If this rule is not enforced in the algorithms, wrong data transfer (endianism issues) would occur and pack/burst mode will be disabled by the hardware, if the software enables them.

For example, for 16/32-bit transfer, the buffers must be aligned on the 16/32-bit boundary and sizes in multiples of 2/4 (bytes).

DMA Rule 10

C55x Algorithms must request all data buffers in external memory with 32-bit alignment and sizes in multiples of 4 (bytes).

DMA Rule 11

C55x Algorithms must use the same data-type and access mode when reading, writing or transferring data stored in external memory or in application passed data buffers.

Note that, specific ACPY2 implementations can additionally provide the framework the ability to disable and enable packed/burst mode on a per channel basis.

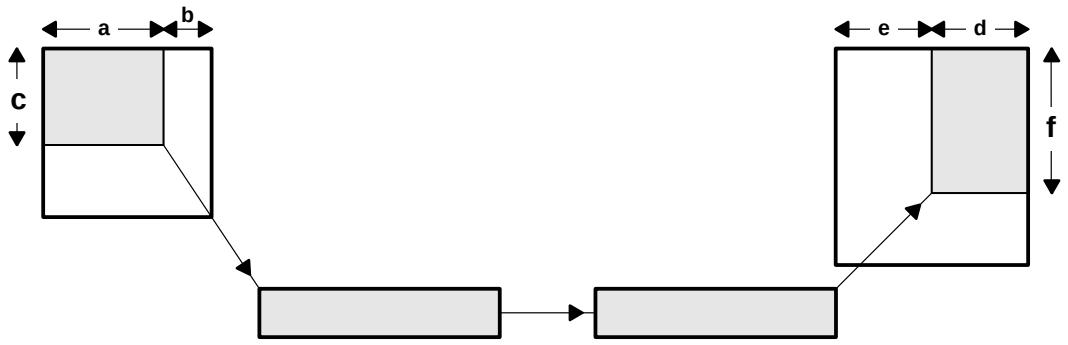
3.8.5 Using ACPY2_start vs ACPY2_startAligned

Although the specification of ACPY2_start as discussed in Section 3.5.1 makes no assumptions on the alignment of the source and destination addresses and accepts addresses at any alignment, the device specific DMA rules and guidelines discussed in this section impose stricter alignment requirements for C55x algorithms when submitting DMA transfer requests. Therefore, even when calling ACPY2_start algorithms must ensure the proper alignment requirements. Hence, C55x algorithms are encouraged to use the ACPY2_startAligned () API exclusively.

4 The Fast Copy (FCPY) Algorithm Example

In this section we present a toy algorithm, FCPY_TI, used to illustrate how to implement the IDMA2 interface and use DMA via ACPY2 calls.The application note, *Design and Implementation of an eXpressDSP-Compliant DMA Managaer for C6x1x* (SPRA789), includes a C6x1x based example application that uses the algorithm developed here. Full source code for the algorithm and the application is provided as an attachment to that application note.

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`, and `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 destination using the parameters `dstLineLen`, `dstNumLines`, and `dstStride`.



a = srcLineLen **d = dstLineLen**
b = srcStride **e = dstStride**
c = srcNumLines **f = dstNumLines**

Figure 9. 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.

4.1 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 function (FCPY_TI_control) that has 2 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.

4.1.1 Instance Heap Memory Requirements

This algorithm requests three buffers (Table 6).

Table 6. Instance Heap Memory Requirements

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

Alignment of these buffers are set at 128 to align at cache boundaries, so that cache coherence issues do not arise.

4.1.2 Use of IDMA2 and ACPY2 Interfaces

The IDMA2 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`, ACPY2 runtime APIs are used to show their invocation procedures.

5 Conclusion

Collectively, IDMA2 and ACPY2 describe a flexible and efficient model that greatly simplifies management of system DMA resources and services by the client application and a simple and powerful mechanism for the algorithm to configure and access DMA services.

This application note presented an overview of the fundamental DMA abstractions specified and supported by the TMS320 DSP Algorithm Standard. We also highlighted the DMA-related enhancements to the algorithm standard in version 2.5 of the TMS320 DSP Algorithm Standard Developer's Kit.

6 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 DSP Two-Level Internal Memory Reference Guide* (SPRU609)
6. *Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6x1x* (SPRA789)

Appendix A Code for FCPY_TI Algorithm

The FCPY_TI algorithm follows the guidelines for achieving high performance.

A.1 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 <ialg.h>
#include <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_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_Cmd ========
 *  This structure defines the control commands for the FCPY module.
 */
typedef enum IFCPY_Cmd {
    IFCPY_GETSTATUS,
    IFCPY_SETSTATUS
} IFCPY_Cmd;

/*
 *  ======== 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_ */
```

A.2 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 <ialg.h>
#include <xdas.h>
#include <ifcpy.h>
#include <idma2.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 */
    IDMA2_Handle dmaHandle1D1D8B;  /* DMA logical channel for 1D to 1D xfers */
    IDMA2_Handle dmaHandle1D2D8B;  /* DMA logical channel for 1D to 2D xfers */
    IDMA2_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);


/* IDMA2 fxn declarations */
extern Void FCPY_TI_dmaChangeChannels(IALG_Handle handle,
                         IDMA2_ChannelRec dmaTab[]);

extern Int FCPY_TI_dmaGetChannelCnt(Void);

extern Int FCPY_TI_dmaGetChannels(IALG_Handle handle,
                         IDMA2_ChannelRec dmaTab[]);

extern Int FCPY_TI_dmaInit(IALG_Handle handle, IDMA2_ChannelRec dmaTab[]);

#ifdef __cplusplus
}
#endif /*__cplusplus*/

#endif  /* FCPY_TI_PRIV_ */
```

A.3 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")

#include <std.h>

#include <fcpy_ti_priv.h>
#include <ifcpy.h>
#include <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 (Int);

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

    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;
}
```

A.4 fcpy_ti_idma2.c

```
/*
 *  ======== fcpy_ti_idma2.c ========
 *  FCPY Module - TI implementation of a FCPY algorithm
 *
 *  This file contains an implementation of the IDMA2 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 <ialg.h>
#include <idma2.h>

#define CHANNEL0 0
#define CHANNEL1 1
#define CHANNEL2 2

#define NUM_LOGICAL_CH 3

/*
 *  ======== FCPY_TI_dmaChangeChannels ========
 *  Update instance object with new logical channel.
 */
Void FCPY_TI_dmaChangeChannels(IALG_Handle handle, IDMA2_ChannelRec dmaTab[])
{
    FCPY_TI_Obj *fcpy = (Void *)handle;

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

}

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

/*
 *  ======== FCPY_TI_dmaGetChannels ========
 *  Declare DMA resource requirement/holdings.
 */
Int FCPY_TI_dmaGetChannels(IALG_Handle handle, IDMA2_ChannelRec dmaTab[])
{
    FCPY_TI_Obj *fcpy = (Void *)handle;

    /* Initial values on logical channels */
```

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

    /* Want all transfers to be serialized (to simplify debugging) */
    dmaTab[CHANNEL0].queueId = 0;
    dmaTab[CHANNEL1].queueId = 0;
    dmaTab[CHANNEL2].queueId = 0;

    return (NUM_LOGICAL_CH);
}

/*
 *  ======== FCPY_TI_dmaInit========
 *  Initialize instance object with granted logical channel.
 */
Int FCPY_TI_dmaInit(IALG_Handle handle, IDMA2_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);
}
```

A.5 fcpy_ti_idmavt.c

```
/*
 *  ======== fcpy_ti_idmavt.c ========
 *  This file contains the function table definitions for the
 *  IDMA2 interface implemented by the FCPY_TI module.
 */
#include <std.h>

#include <idma2.h>
#include <fcpy_ti.h>
#include <fcpy_ti_priv.h>

/*
 *  ======== FCPY_TI_IDMA2 ========
 *  This structure defines TI's implementation of the IDMA2 interface
 *  for the FCPY_TI module.
 */
IDMA2_Fxns FCPY_TI_IDMA2 = {      /* 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 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 <xdas.h>
#include <idma2.h>
#include <acpy2.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;
    IDMA2_Params params;

    /* Configure the logical channel */
    params.xType = IDMA2_1D1D;
    params.elemSize = IDMA2_ELEM8;
    params.numFrames = 0;    // Not used in 1D1D transfer
    params.srcFrameIndex = 0;       // Not used in 1D1D transfer
    params.dstFrameIndex = 0;       // Not used in 1D1D transfer
    params.srcElementIndex = 0;
    params.dstElementIndex = 0;

    /* Configure logical dma channel */
    ACPY2_configure(fcpy->dmaHandle1D1D8B, &params);

    /* Configure the logical channel */
    params.xType = IDMA2_2D1D;
    params.elemSize = IDMA2_ELEM8;
    params.numFrames = fcpy->srcNumLines;
    params.srcFrameIndex = fcpy->srcStride;
    params.dstFrameIndex = 0;

    /* Configure logical dma channel */
    ACPY2_configure(fcpy->dmaHandle2D1D8B, &params);

    /* Configure the logical channel */
    params.xType = IDMA2_1D2D;
    params.elemSize = IDMA2_ELEM8;
    params.numFrames = 0;
    params.srcFrameIndex = 0;
    params.dstFrameIndex = 0;

    /* Configure logical dma channel */
    ACPY2_configure(fcpy->dmaHandle1D2D8B, &params);

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

    /* NOTE: Extra data processing could be done here */

    /* Check that dma transfer has completed before finishing ”processing” */
    while (!ACPY2_complete(fcpy->dmaHandle2D1D8B)) {
      ;
    };

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

```
                    (Void *)(fcpy->workBuf2),
                    (Uns)((fcpy->srcLineLen) * (fcpy->srcNumLines)));

    /* wait for transfer to finish  */
    ACPY2_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 */
    ACPY2_start(fcpy->dmaHandle1D2D8B, (Void *)(fcpy->workBuf2),
                    (Void *)out, (Uns)(fcpy->dstLineLen));

    /* wait for all transfers to complete before returning to the client */
    ACPY2_wait(fcpy->dmaHandle1D2D8B);
}

/*
 *  ======== 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) {
        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);
}
```

A.6 fcpy_ti_ialgvt.c

```
/*
 *  ======== fcpy_ti_ialgvt.c ========
 *  This file contains the function table definitions for the
 *  IALG and IFCPY interfaces implemented by the FCPY_TI module.
 */

#include <std.h>

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

#define IALGFXNS \
    &FCPY_TI_IALG,     /* module ID */                          \
    NULL,             /* activate (NULL => no need to initialize buffers */ \
```

```
    FCPY_TI_alloc,     /* alloc */                              \
    NULL,              /* control (NULL => no control ops) */  \
    NULL,              /* deactivate (NULL => no need to save data */ \
    FCPY_TI_free,      /* free */                               \
    FCPY_TI_initObj,   /* init */                               \
    FCPY_TI_moved,     /* moved */                              \
    NULL               /* numAlloc (NULL => IALG_MAXMEMRECS) */

/*
 *  ======== FCPY_TI_IFCPY ========
 *  This structure defines TI's implementation of the IFCPY interface
 *  for the FCPY_TI module.
 */
IFCPY_Fxns FCPY_TI_IFCPY = {       /* module_vendor_interface */
    IALGFXNS,                      /* IALG functions */
    FCPY_TI_control,               /* Control function */
    FCPY_TI_doCopy                 /* The fcpy fxn */
};

/* Overlay v-tables to save data space */
asm("_FCPY_TI_IALG  .set _FCPY_TI_IFCPY");
```