Application Report
SPRA789 - September 2002

Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6X1X

Murat Karaorman
Vincent Wan
Texas Instruments, Santa Barbara

ABSTRACT

Following an overview of DMA abstractions for eXpressDSP-compliant algorithms, this application describes a C6x1x-optimized (C6211, C6711) ACPY2 library implementation and DMA Resource Manager, which can also be run on C64x devices. An algorithm and example application is provided (including source code) to demonstrate enhancements to DMA access.

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

Contents

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 client 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 any hardware peripherals, including the DMA. All system DMA resources must be controlled by the client application.

The *TMS320 DSP Algorithm Standard* (also known as XDAIS) specifies standard interfaces, IDMA2 and ACPY2, that allow the client application and algorithms to negotiate DMA resources, which in turn grants algorithms controlled access to DMA services. The client application uses the algorithm’s IDMA2 interface to query 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. Algorithms call the ACPY2 API functions implemented by the client application to schedule DMA transfers on the logical DMA channels.

This application note presents a C6x1x-specific (C6211, C6711) implementation of the ACPY2 APIs, a DMA Manager (DMAN) interface and implementation and an example algorithm and application (including source code) to demonstrate an end-to-end system with algorithms that use DMA.

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. Section 3 provides a function-by-function description of this implementation.

The fastcopytest example in section 4 presents a complete example to illustrate how to use an algorithm that implements the IDMA2 interface.

1.1 Overview of Standard DMA Interfaces

Figure 1 shows which modules are implemented by the client application and which are implemented by the algorithm. Arrows indicate which modules use other modules. The interfaces in the center are used to make implementations independent of one another.

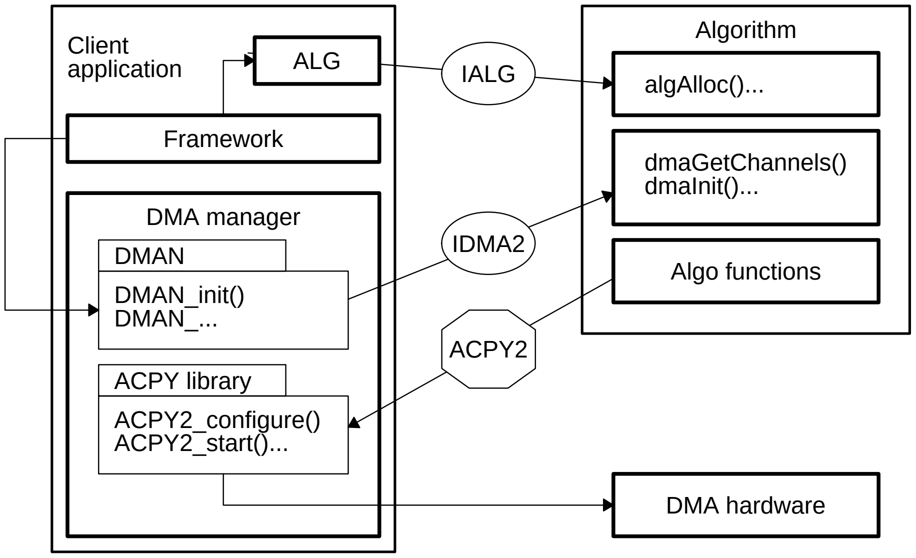

Figure 1. Client Application and Algorithm Interaction With DMA Resources

Algorithms must access DMA hardware via the “logical” DMA channel handles 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. In this application note we introduce a new optional module, DMAN, which provides a convenient wrapper layer around the IDMA2 interface for querying and granting algorithms logical DMA channels.

- **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 in order to use algorithms that use the DMA resource. The ACPY2 interface describes the comprehensive list of DMA operations algorithm can perform on the logical DMA channels acquired through the IDMA2 protocol. The ACPY2 functions allow:
  - Configuring channel DMA transfer parameters applied to each submitted DMA request
  - Submitting asynchronous DMA transfers requests.
  - Synchronizing with scheduled transfers (both blocking and non-blocking).

- **DMAN** - Client applications may provide and use a DMA manager to grant DMA resources to algorithms. The DMAN interface is neither required nor specified by the XDAIS rules and guidelines. However, it may be useful for a client application to modularize use of the IDMA2 and ACPY2 interfaces. The DMAN module provided with this application note may be used for this purpose.

The *Use of the DMA Resource* chapter in *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352) describes the use of the ACPY2 and IDMA2 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.

The following tables summarize the API functions and structures used by the IDMA2 and ACPY2 interfaces. DMAN interface and design details are presented in the next section.

Table 1. IDMA2 Functions

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 handle(s) to the algorithm at initialization.

Table 2. ACPY2 Functions

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

Table 3. IDMA2_ChannelRec Structure Fields

Structures	Description
handle	Handle to logical DMA channel
queueId	Selects the serialization queue

Table 4. IDMA2_Params Structure Fields

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
srcElementIndex	Gap + elemSize between consecutive elements in source data (in 8-bit bytes)
dstElementIndex	Gap + elemSize between consecutive elements in destination data (in 8-bit bytes)
srcFrameIndex	Gap between consecutive source data frames for 2D transfers (in 8-bit bytes)
dstFrameIndex	Gap between consecutive destination data frames for 2D transfers (in 8-bit bytes)

1.2 DMA Transfer Configuration Settings

Each DMA transfer is submitted on a logical channel by calling the `ACPY2_start` or ACPY2_startAligned function. Each 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 unit of DMA transfer is a *transfer block* composed of *frames* and *elements*. The physical layouts of the source or destination memory regions do not have to be single contiguous chunks. The *source* and *destination addresses* for the blocks and the *number of elements* in each frame are passed as function arguments to `ACPY2_start`. The remaining configuration parameters are the intrinsic properties of the logical channel and are set by the algorithm by calling the ACPY2 configuration functions. The previously configured properties of the logical channel at the time of transfer request determine the actual memory that gets copied from source to destination. A 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.

- Transfer type (`xType`): 1D-to-1D, 1D-to-2D, 2D-to-1D or 2D-to-2D
- Element size (`elemSize`): The number of 8-bit bytes per element ∈ {1, 2, 4}.
- Number of frames (`numFrames`): The number of frames in a block.
- Element index (`srcElementIndex` or `dstElementIndex`): 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.
- Frame index (`srcFrameIndex` or `dstFrameIndex`): size of the gap in 8-bit bytes between two consecutive frames within a block. Defined for 2D transfers only.

- Number of elements (argument to ACPY2_start or ACPY2_startAligned): the number of elements per frame.
- Source and destination addresses (argument to ACPY2_start or ACPY2_startAligned): as 8-bit byte-addresses.

Element and frame index parameters are shared by both source & destination if the hardware does not support setting these independently, as is the case for the C6x1x EDMA architecture. Configure functions should indicate error status when any configuration settings are not supported by the client implementation.

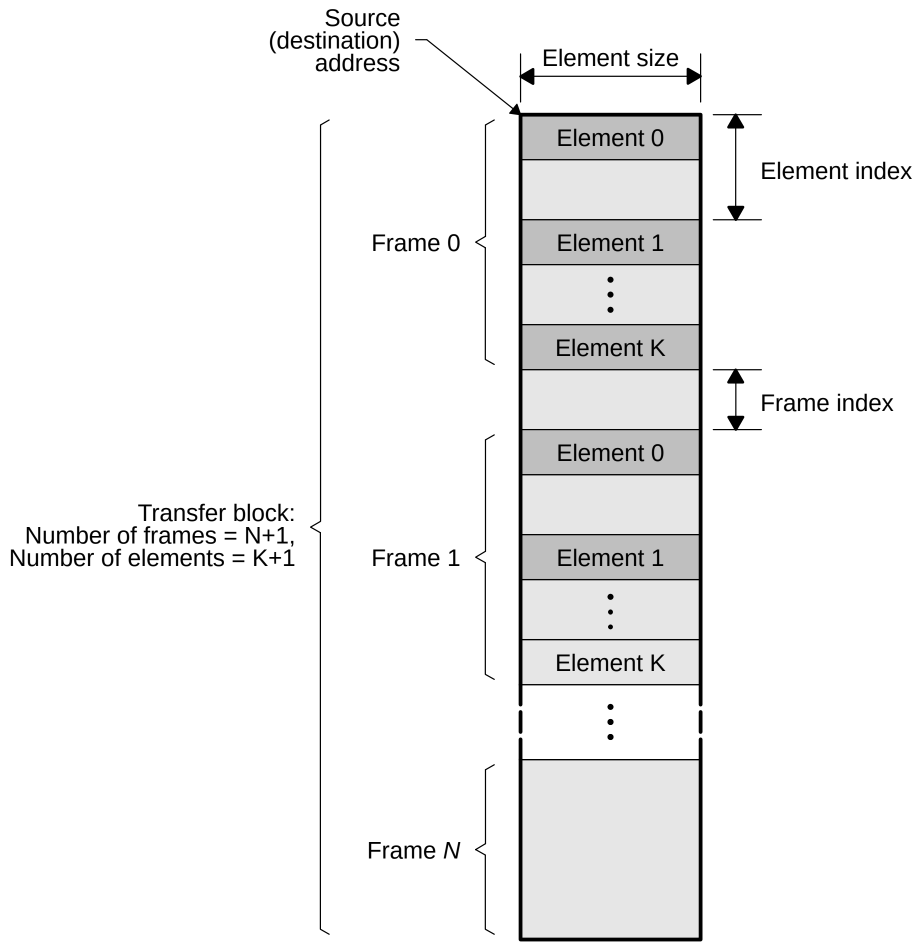

Figure 2. DMA Transfer Block

2 Generic DMA Manager Module: DMAN

An optional DMAN module is provided with this application note. This module can be customized as needed, and is a convenient way to create algorithm instances.

2.1 Using DMAN for Algorithm Integration

The steps for integrating an algorithm that uses DMA resources can be greatly simplified by making use of the DMAN module provided with this application note. The DMAN module acts as a DMA manager to grant DMA resources to algorithms. This module is neither required nor specified by the XDAIS rules and guidelines. However, it simplifies and modularizes use of the IDMA2 and ACPY2 interfaces.

Following steps provide a generic and convenient set of instructions to use the DMAN module to instantiate algorithm instances that requests DMA resources. The code examples are from the fastcopytest.c example provided with this application note.

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

```
#include <dman.h>
```

2. Implement or include an ACPY2 library in the application. The ACPY2 module must implement the functions specified by the XDAIS specification. An ACPY2 implementation for C6x1x is provided with this application note.

```
#include <acpy2_6x1x.h>
```

3. Initialize the algorithm by calling its algInit() function, and set values for fields in the algorithm-specific params structure, which is declared in i<mod>.h.

```
FCPY_Params fcpyParams;
FCPY_Handle alg;
...
FCPY_init();
fcpyParams = FCPY_PARAMS; /* use the default creation parameters */
```

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

```
if ((alg = FCPY_create(&fcpyParams)) == NULL) {
     SYS_abort(“Could not create algorithm instance”);
}
```

5. Initialize the ACPY2 library.

```
ACPY2_6X1X_init();
```

6. Initialize and set up the DMAN module.

```
DMAN_init();
DMAN_setup(INTERNALHEAP);
```

7. Use the DMAN module to grant the DMA resources requested by the algorithm.

```
if (DMAN_addAlg((IALG_Handle)alg, &FCPY_IDMA2) == FALSE) {
    SYS_abort(“Problem adding algorithm’s dma resources”);
}
```

2.2 DMAN Module APIs

These functions are intended to provide application frameworks a convenient and easy to use layer to integrate algorithms requesting DMA resources.

Table 5. DMAN Functions

Functions	Description
DMAN_addAlg ()	Grant logical channel resources to an algorithm instance.
DMAN_exit ()	Finalization method of the DMAN module.
DMAN_init ()	Initialize the DMAN module.
DMAN_removeAlg ()	Remove logical channel resources from an algorithm instance.
DMAN_setup	Setup the DMAN module. Specifies the heap identifier used for MEM module for allocating memory for channel handles.

2.3 Implementing the DMAN Interface

The following list describes each function that must be implemented.

2.3.1 DMAN_addAlg()

- Add an algorithm to the DMA Manager. The DMA Manager will grant DMA resources to the algorithm as a result. This function is called when initializing an algorithm instance.

```
/*
 *  ======== dman_addalg.c ========
 *  Grant logical channel resources to an algorithm instance
 */

static Bool allocChannels(IDMA2_ChannelRec dmaTab[], Int n);

/*
 *  ======== DMAN_addAlg ========
 *  Add an algorithm to the DMA Manager.  The DMA Manager will grant DMA
 *  resources to the algorithm as a result.  This function is called when
 *  initializing an algorithm instance.
 */
Bool DMAN_addAlg(IALG_Handle algHandle, IDMA2_Fxns *dmaFxns)
{
    Int numChan;
    IDMA2_ChannelRec dmaTab[_DMAN_MAXDMARECS];

    /* verify that alg and idma2 fxns are from same implementation */
    if ((algHandle != NULL) && (dmaFxns != NULL) &&
        (dmaFxns->implementationId == algHandle->fxns->implementationId)) {

        numChan = dmaFxns->dmaGetChannelCnt();

        /*
         *  Stack-based dmaTab records avoid the risk of fragmentation.
         *  A maximum number of records is set.  If, in the unlikely case,
         *  more records than the maximum are requested, return failure
         */
        if (numChan > _DMAN_MAXDMARECS) {
           return (FALSE);
        }

        numChan = dmaFxns->dmaGetChannels(algHandle, dmaTab);
        if (numChan <= 0) {
            return (FALSE);
        }

        if (allocChannels(dmaTab, numChan) == TRUE) {
            if (dmaFxns->dmaInit(algHandle, dmaTab) == IALG_EOK) {
                return (TRUE);  // DMA init success
            }
            /* If dmaInit fails for any reason, free channel resources */
            _DMAN_freeChannels(dmaTab, numChan);
        }
    }
    return (FALSE);
}

/*
 *  ======== allocChannels ========
 *  Allocate and initialize logical channels (IDMA2_Obj’s) requested in
 *  a dmaTab[].
 */
static Bool allocChannels(IDMA2_ChannelRec dmaTab[], Int numChan)
{
    Int i;
    Int chanObjSize = ACPY2_getChanObjSize();

    for (i = 0; i < numChan; i++) {
        /*
         * Word alignment is done to help simplify accesses of fields
```

```
             * in the IDMA2_Obj structure by assembly implementations of ACPY2
             */
            dmaTab[i].handle = (IDMA2_Handle)MEM_alloc(_DMAN_heapId, chanObjSize,
                                                  sizeof (Int));
            if (dmaTab[i].handle == MEM_ILLEGAL) {
                _DMAN_freeChannels(dmaTab, i);
                return (FALSE);
            }

            /* Initialize channel object */
            ACPY2_initChannel(dmaTab[i].handle, dmaTab[i].queueId);
        }

        return (TRUE);
    }

    /*
     *  ======== freeChannels ========
     *  Reclaim logical channel resources (IDMA2_Obj's).
     */
    Void _DMAN_freeChannels(IDMA2_ChannelRec dmaTab[], Int numChan)
    {
        Int i;
        Int chanObjSize = ACPY2_getChanObjSize();

        for (i = 0; i < numChan; i++) {
            if (dmaTab[i].handle != MEM_ILLEGAL) {
                MEM_free(_DMAN_heapId, dmaTab[i].handle, chanObjSize);
            }
        }

        return;
    }
```

2.3.2 *DMAN_init()*.

- Initialize the DMAN module.

```
Void DMAN_init(Void)
{
}
```

2.3.3 *DMAN_exit()*.

- Finalization method of the DMAN module

```
Void DMAN_exit(Void)
{
}
```

2.3.4 *DMAN_removeAlg()*.

This function removes logical channel resources from an algorithm instance. This function is called before deallocating an algorithm instance in dynamic systems.

```
/*
 *  ======== DMAN_removeAlg ========
 *  Remove logical channel resources from an algorithm instance.
 *  This function is called before deallocating an algorithm instance
 *  in dynamic systems.
 */
Bool DMAN_removeAlg(IALG_Handle algHandle, IDMA2_Fxns *dmaFxns)
{
    IDMA2_ChannelRec dmaTab[_DMAN_MAXDMARECS];
    Int numChan;
    Int actualNumChan;

    /* verify that alg and idma2 fxns are from same implementation */
    if ((algHandle != NULL) && (dmaFxns != NULL) &&
        (dmaFxns->implementationId == algHandle->fxns->implementationId)) {

        numChan = dmaFxns->dmaGetChannelCnt();

        /*
         *  Stack-based dmaTab records reduces fragmentation.
         *  A maximum number of records is set.  If, in the unlikely case,
         *  more records than the maximum are requested, return failure
         */
        if (numChan > _DMAN_MAXDMARECS) {
            return (FALSE);
        }

        actualNumChan = dmaFxns->dmaGetChannels(algHandle, dmaTab);
        if (actualNumChan <= 0) {
            return (FALSE);
        }

        _DMAN_freeChannels(dmaTab, actualNumChan);

        return (TRUE);
    }
    return (FALSE);
}
```

2.3.5 DMAN_setup()

Function to setup the DMAN module. Specifies the heap ID for dynamic allocation. This function is called when initializing the DMAN module. In a dynamic system, it can also be called to change the heap used by DMAN **after** all algorithms have first been removed from DMAN using DMAN_removeAlg.

```
Void DMAN_setup(Int heapId)
```

```
{
    /* Assumed heapId to be a valid segment address */
    _DMAN_heapId = heapId;
}
```

3 ACPY2 Interface Reference Example

This section describes the C6x1x-specific (C6211, C6711) ACPY2 implementation provided with this application note. A hand-optimized assembly version of the same library is provided in the TMS320 DSP Algorithm Standard Developer's Kit 2.5. After the overview, details and code are shown for each function.

3.1 Overview of the QDMA Mechanism on C6211/ C6711/ C64x

The enhanced direct memory access (EDMA) controller handles all data transfers between the level-two (L2) cache/memory controller and device peripherals on the C621x/C671x/C64x. The EDMA controller in the C621x/C671x/C64x is different from the DMA controller in C620x/C670x devices. Enhancements include providing 64 channels (C64x) or 16 channels (C621x/C671x) with programmable priority and the ability to link and chain data transfers.

The EDMA device on the C6000 supports the quick DMA (QDMA) mechanism, which is one of the most efficient ways of moving data. The EDMA can perform fast and efficient transfers by accepting a QDMA request from the CPU. A QDMA transfer is best suited to applications that require quick data transfers, such as data requests in a tight loop algorithm.

Figure 3 shows how the QDMA registers can be used for data transfers.

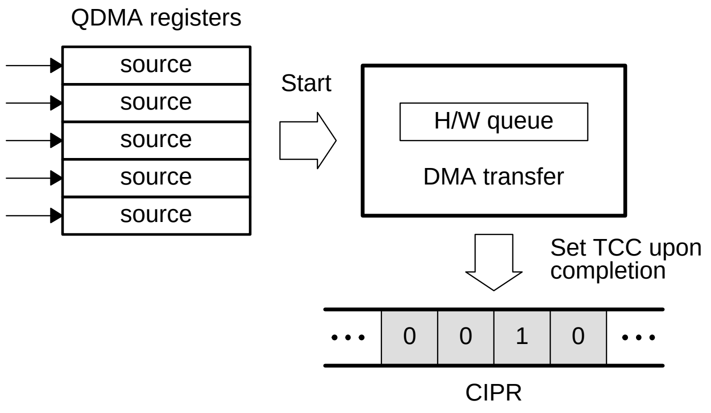

Figure 3. QDMA and CIPR Registers Used in Data Transfers

The QDMA consists of two sets of five memory mapped, write-only registers: the "QDMA registers" and the "QDMA pseudo registers." Writes to the QDMA registers configure, but do not submit, the next DMA data transfer request. Writing to any one of the five pseudo registers submits a transfer request. Hence, as shown in Figure 3, one would generally write to four QDMA registers (source, destination, option, index) and to one pseudo register (count) to configure and submit a transfer request.

Two transfer request queues are available on the C6211/ C6711 for the QDMA mechanism: a high-priority queue and a low-priority queue.

On the C6211/ C6711, at most 3 simultaneous requests can be submitted on each queue through the QDMA mechanism. Each request must assigned one of 16 transfer complete codes (TCC) by specifying it in one of the five QDMA registers. When the transfer is completed, a bit in the 16-bit channel interrupt pending register (CIPR) corresponding to the TCC is set. Therefore, this register can be monitored to verify the status of each transfer request.

On the C64x devices, there is more flexibility. The maximum number of simultaneous requests in each transfer request queues is programmable and a total of 64 TCCs are available. Note that the ACPY2 implementation provided with this application note has been optimized for the C6211/C6711, so it only supports the lower 16 TCCs and two of the hardware queues (high-priority and medium-priority) on the C64x when used on the latter.

3.2 High-Performance ACPY2 Library Implementation

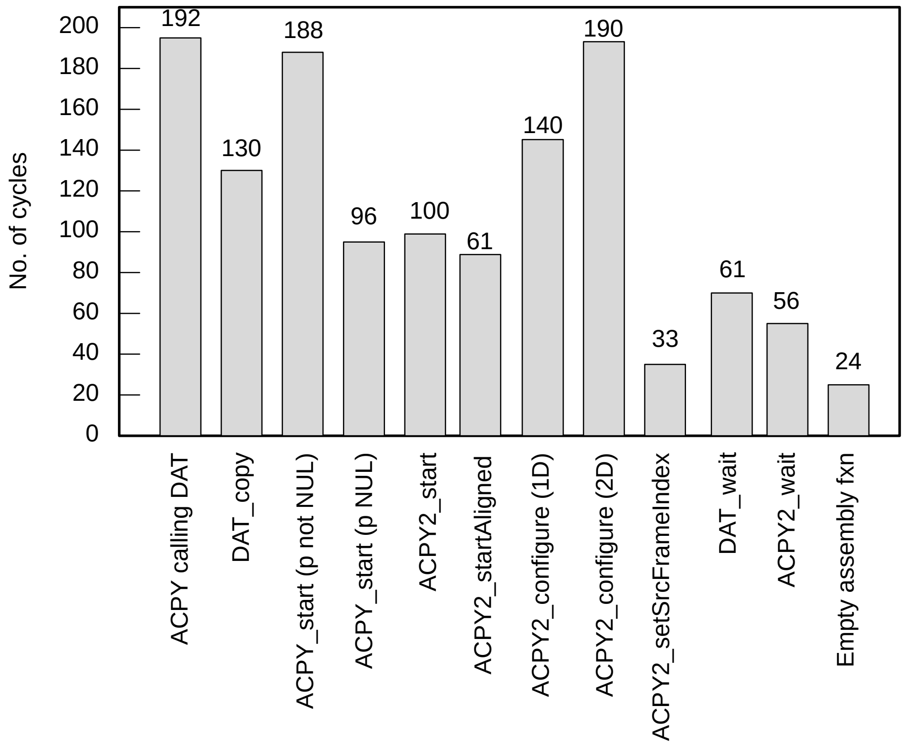

Figure 4. Performance Comparison

The best-case performance numbers for the high-performance ACPY2 library implementation supplied with this application report are shown in Figure 4. The numbers were obtained using the profiler clock in Code Composer Studio 2.1. When the new DMA guidelines from TMS320 DSP Algorithm Standard are followed, this implementation is capable of providing better performance than traditional ACPY implementations that use the DAT module, as can be seen in the first column in Figure 4.

This example implementation for the C6211/ C6711/ C64x devices is based on the following logical IDMA2 channel structure:

```
/*
 * IDMA2_Obj is the structure representation of a logical DMA channel.
 * It contains the state information of the channel
 */
typedef struct IDMA2_Obj {
    IDMA2_Params params ; // Used to hold the current channel config params
    Int lastTCC         ; // The last TCC used by this channel in decimal
    EDMA_Config config  ; // Config structure used to store QDMA parameters
                          // before writing them to registers
} IDMA2_Obj;
```

This structure contains three fields:

- params holds the current set of IDMA2 parameters used by this channel.
- lastTCC holds the TCC used by this channel to submit its last transfer.
- Config is an EDMA_Config structure used to store intermediate values to be directly stored in QDMA registers when submitting a transfer.

To optimize the process of starting transfers in `ACPY2_start()`, it is best to fill the EDMA_config structure as much as possible when configuring channels. This minimizes overhead when submitting transfer requests.

3.2.1 ACPY2_init()

ACPY2_init() initializes the ACPY2 module. It is called by the client application to allocate TCCs for use in ACPY2 and to initialize the corresponding bits in the CIPR. This function uses dummy transfers to set the TCC bits allocated for ACPY2. This causes bits in CIPR register to be set. Later, when `ACPY2_start()` is called, the transfer can take ownership of one of the TCC bits set, and reset it. `ACPY2_wait()` can then wait on the TCC bit to get set again to ensure transfer completion.

```
/*
 *  ======== acpy2_init.c ========
 */

#pragma CODE_SECTION(ACPY2_init, ”.text:ACPY2_init”)

#pragma DATA_SECTION(_ACPY2_TCCTable, ”.bss:_ACPY2_TCCTable”)

#include <std.h>
#include <sys.h>

#include <csl_edma.h>

#include <_acpy2.h>
#include <acpy2_6x1x.h>
#include <idma2_priv.h>

//Table recording which handle last used a particular TCC
IDMA2_Handle _ACPY2_TCCTable[_ACPY2_TCCTABLESIZE] =
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

Int _ACPY2_TCCmask = 0;  //Bits in CIPR reserved for ACPY2’s use
Char _ACPY2_TCCsAllocated[_ACPY2_TCCTABLESIZE] = {
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
```

```
/*
 * ======== ACPY2_init ========
 * Initialize the ACPY2 module
 *
 * The strategy consists of setting the TCC bits allocated for ACPY2
 *    using dummy transfers.  In this manner, when ACPY2_start is called,
 *    the transfer can take ownernership of one of the TCC bits set, and
 *    reset it.  ACPY2_wait can then wait on the TCC bit to get set again
 *    to ensure transfer completion.
 */
Void ACPY2_init(Void)
{
    Int count = 0;

    //Used for dummy transfers to set CIPR register used in QDMA transfers.
    Uint32 dummySrc;
    Uint32 dummyDst;
    EDMA_Config dummyCfg;

    //if ACPY2_init has never been called before
    if (_ACPY2_TCCmask == 0) {

        //Initialize dummyCfg structure
        dummyCfg.opt = EDMA_FMKS(OPT,PRI,HIGH) |
            EDMA_FMKS(OPT,ESIZE,8BIT) |
            EDMA_FMKS(OPT,2DS,NO) |
            EDMA_FMKS(OPT,SUM,INC) |
            EDMA_FMKS(OPT,2DD,NO) |
            EDMA_FMKS(OPT,DUM,INC) |
            EDMA_FMKS(OPT,TCINT,YES) |
            EDMA_FMKS(OPT,TCC,OF(0)) |
            EDMA_FMKS(OPT,LINK,NO) |
            EDMA_FMKS(OPT,FS,YES);

        dummyCfg.src = (Uint32)&dummySrc;
        dummyCfg.cnt = 0x4;  //Transfer 4 bytes to fill dummyDst
        dummyCfg.dst = (Uint32)&dummyDst;
        dummyCfg.idx = 0;

        //Allocate the TCCs for ACPY2.
        for (count = 0; count < ACPY2_6X1X.numTCC; count++) {
            _ACPY2_TCCsAllocated[count] = EDMA_intAlloc(-1);
            if (_ACPY2_TCCsAllocated[count] == -1) {
                SYS_abort(”Not enough TCCs available”);
            }
            else if (_ACPY2_TCCsAllocated[count] >= _ACPY2_TCCTABLESIZE) {
               /*
                * NOTE: On C64x, if EDMA_intAlloc returns a TCC value that is
                * not supported on the C6211, this implementation will fail
                * since it is built to work with C6211.  It simply maintains
                * compatibility with C64x devices.
                */
```

```
                    SYS_abort(”TCC returned exceeds the maximum TCC supported” \
                              ”by this implementation.”);
                }

                _ACPY2_TCCmask |= (1 << _ACPY2_TCCsAllocated[count]);

                /*
                 * Do a dummy transfer so that bits in CIPR register are set,
                 * meaning TCCs are ready for another transfer
                 */
                EDMA_FSETA(&(dummyCfg.opt), OPT, TCC, _ACPY2_TCCsAllo-
cated[count]);
                EDMA_qdmaConfig(&dummyCfg);
            }
        }
    }
```

3.2.2 ACPY2_exit()

ACPY2_exit() is called by the client application. It frees TCCs reserved for use by ACPY2.

```
/*
 *  ======== acpy2_exit.c ========
 */

Void ACPY2_exit(Void)
{
    int count;

    //Free TCCs allocated by ACPY2
    for (count = 0; count < ACPY2_6X1X.numTCC; count++) {
        EDMA_intFree(_ACPY2_TCCsAllocated[count]);
    }
}
```

3.2.3 ACPY2_initChannel()

ACPY2_initChannel() is called by the client application. It initializes a channel’s IDMA2_Obj structure. It also accepts the queueId of the logical channel and maps it to a corresponding DMA priority level, attempting to utilize all hardware queues to maximize DMA bandwidth (For more details on EDMA hardware queues usage, see *TMS320C621x/C671x EDMA Queue Management Guidelines*, SPRA720). All transfers submitted on the same priority level are serialized in hardware on the EDMA device.

```
/*
 *  ======== ACPY2_initChannel ========
 *  Initialize the IDMA2 channel object passed in.  Set the priority level
 *  based on the queue id.
 *
 */
Void ACPY2_initChannel(IDMA2_Handle handle, Int queueId)
{
    /*
     * NOTE: qid is an arbitrary number that has to be mapped to physical
     * hardware queues on the C6x EDMA device.  The scheme chosen here
     * is to map even queue ids to low priority h/w queue and odd queue
     * ids to high priority h/w queue.    This scheme can be customized to
     * suit a specific application.
     */
    if ((queueId % _ACPY2_NUM_HWQUEUES) == 0) {
        //Set to low priority
        (handle->config).opt = EDMA_FMK(OPT,PRI,EDMA_OPT_PRI_LOW);
    }
    else {
        //Set to high priority
        (handle->config).opt = EDMA_FMK(OPT,PRI,EDMA_OPT_PRI_HIGH);
    }
}
```

3.2.4 *ACPY2_getChanObjSize()*

ACPY2_getChanObjSize() is called by the client application. It returns the size of the channel's IDMA2_Obj structure. It is typically used when the client application is allocating memory for the logical channels.

```
Uns ACPY2_getChanObjSize(Void)
{
    return (sizeof (IDMA2_Obj));
}
```

3.2.5 *ACPY2_configure()*

ACPY2_configure() is called by the algorithm to configure data transfer parameters for the channel. Using the IDMA2_Params argument, it sets up the following values and stores them within the channel object:

- Value to be written to the option register based on the transfer type and element size of the logical channel.
- Value to be written to the index register based on the source/destination element index and frame index parameters.
- Value to be written to the count pseudo-register based on the numFrames parameter.

```
/*
 * ======== ACPY2_configure ========
 * Configure a logical channel
 */
Void ACPY2_configure(IDMA2_Handle handle, IDMA2_Params *params)
{
    //Save priority level set
    Uns priField = EDMA_FGETA(&((handle->config).opt),OPT,PRI);

    handle->params = *params;

    /*
     *  Default value for option register.  The assumption here for c64x
     *  devices is that the extra fields have default values of 0.
     */
    handle->config.opt =
        EDMA_FMKS(OPT,PRI,OF(priField)) |
        EDMA_FMKS(OPT,ESIZE,OF(0)) |
        EDMA_FMKS(OPT,TCINT,YES) |
        EDMA_FMKS(OPT,TCC,OF(0)) |
        EDMA_FMKS(OPT,LINK,NO);

    //Frame and element count set to 0 for 1D transfers
    handle->config.cnt = 0;

    if (params->xType != IDMA2_1D1D) {  //For 2D transfers

        /*
         *  params->stride and params->numFrames are only initialized when it
         *  is not a 1D transfer
         */
        handle->config.cnt = EDMA_FMK(CNT,FRMCNT,((params->numFrames) - 1));

        //Check if element indexing is used
        if ((params->srcElementIndex == 0) &&
            (params->dstElementIndex == 0)) {
            switch (params->xType) {
               case IDMA2_2D2D:
                    handle->config.opt |=
                        EDMA_FMKS(OPT,FS,YES)    |
                        EDMA_FMKS(OPT,SUM,INC) |
                        EDMA_FMKS(OPT,DUM,INC) |
                        EDMA_FMKS(OPT,2DS,YES) |
                        EDMA_FMKS(OPT,2DD,YES);
                    //Check for invalid src and dst frame index combinations
                    if (params->srcFrameIndex != params->dstFrameIndex) {
                       SYS_abort(”Frame indices must be same in 2D2D trans
                                  fers”);
                    }
                    else {
                        handle->config.idx = EDMA_FMK(IDX,FRMIDX,
                            (params->dstFrameIndex));
```

```
                }
                break;
            case IDMA2_1D2D:
                handle->config.opt |=
                    EDMA_FMKS(OPT,FS,YES)   |
                    EDMA_FMKS(OPT,SUM,INC) |
                    EDMA_FMKS(OPT,DUM,INC) |
                    EDMA_FMKS(OPT,2DS,NO) |
                    EDMA_FMKS(OPT,2DD,YES);
                handle->config.idx = EDMA_FMK(IDX,FRMIDX,
                    (params->dstFrameIndex));
                break;
            case IDMA2_2D1D:
                handle->config.opt |=
                    EDMA_FMKS(OPT,FS,YES)   |
                    EDMA_FMKS(OPT,SUM,INC) |
                    EDMA_FMKS(OPT,DUM,INC) |
                    EDMA_FMKS(OPT,2DS,YES) |
                    EDMA_FMKS(OPT,2DD,NO);
                handle->config.idx = EDMA_FMK(IDX,FRMIDX,
                    (params->srcFrameIndex));
                break;
            default:
                //Should never end up here!
                SYS_abort(”Invalid transfer type”);
                break;
        }
    }
    else {
        SYS_abort(”Element index has to be zero in 2D transfers on
                  C6x1x”);
    }

}
else {   //For 1D transfers
    //Check if element indexing is used
    if ((params->srcElementIndex == 0) &&
        (params->dstElementIndex == 0)) {
        //Both src and dst have element indexing mode disabled
        handle->config.idx = 0;
        handle->config.opt |=
             EDMA_FMKS(OPT,FS,YES)   |
             EDMA_FMKS(OPT,SUM,INC) |
             EDMA_FMKS(OPT,DUM,INC) |
             EDMA_FMKS(OPT,2DS,NO) |
             EDMA_FMKS(OPT,2DD,NO);
    }
    else if (params->srcElementIndex == 0) {
        //Enable element indexing mode for dst
        handle->config.idx = EDMA_FMK(IDX,ELEIDX,
            (params->dstElementIndex));
        handle->config.opt |=
            EDMA_FMKS(OPT,FS,NO)   |
                EDMA_FMKS(OPT,SUM,INC) |
                EDMA_FMKS(OPT,DUM,IDX) |
                EDMA_FMKS(OPT,2DS,NO) |
                EDMA_FMKS(OPT,2DD,NO);
        }
        else if (params->dstElementIndex == 0) {
            //Enable element indexing mode for src
            handle->config.idx = EDMA_FMK(IDX,ELEIDX,
                (params->srcElementIndex));
            handle->config.opt |=
                EDMA_FMKS(OPT,FS,NO)   |
                EDMA_FMKS(OPT,SUM,IDX) |
                EDMA_FMKS(OPT,DUM,INC) |
                EDMA_FMKS(OPT,2DS,NO) |
                EDMA_FMKS(OPT,2DD,NO);
      }
      else if (params->dstElementIndex == params->srcElementIndex) {
            //Enable element indexing mode for src and dst
            handle->config.idx = EDMA_FMK(IDX,ELEIDX,
                (params->srcElementIndex));
            handle->config.opt |=
                EDMA_FMKS(OPT,FS,NO)   |
                EDMA_FMKS(OPT,SUM,IDX) |
                EDMA_FMKS(OPT,DUM,IDX) |
                EDMA_FMKS(OPT,2DS,NO) |
                EDMA_FMKS(OPT,2DD,NO);
        }
        else {
            SYS_abort(”Invalid combination of src and dst element indices”);
        }
    }

    //Set the element size field
    switch (params->elemSize) {
        case IDMA2_ELEM8:
            (handle->config).opt |= EDMA_FMKS(OPT,ESIZE,8BIT);
            break;
        case IDMA2_ELEM16:
            (handle->config).opt |= EDMA_FMKS(OPT,ESIZE,16BIT);
            break;
        default:
            break;
    }
}
```

ACPY2_setNumFrames(), ACPY2_setSrcFrameIndex() and ACPY2_setDstFrameIndex are faster alternatives for reconfiguring channels.

3.2.6 ACPY2_setNumFrames()

ACPY2_setNumFrames() allows an algorithm to rapidly change the numFrames parameter of an IDMA2 channel. Using the value argument, it sets the numFrames parameter. It also converts the value to a corresponding bit field to be written to the count pseudo-register and stored in the channel object.

```
/*
 *  ======== ACPY2_setNumFrames ========
 *  Rapidly configure the numFrames parameter of an IDMA2 channel
 */

Void ACPY2_setNumFrames(IDMA2_Handle handle, Uns numFrames)
{
    handle->params.numFrames = numFrames;
    handle->config.cnt = EDMA_FMK(CNT,FRMCNT,
        ((handle->params.numFrames) - 1));
}
```

3.2.7 ACPY2_setSrcFrameIndex()

ACPY2_setSrcFrameIndex() allows an algorithm to rapidly change the source frame index parameter of an IDMA2 channel. It also converts the frameIndex value to a corresponding bit field to be written to the count pseudo-register and stores this bit field in the channel object. Note that on the C6x1x EDMA device, ACPY2_setSrcFrameIndex() has the same behavior as ACPY2_setDstFrameIndex() in 2D to 2D transfers since the hardware only supports one frame index value.

```
/*
 *  ======== ACPY2_setSrcFrameIndex ========
 *  Rapidly configure the source Frame index parameter of an IDMA2 channel
 *  Note that both source and destination indexes are set simultaneously
 *  on the C6x1x with this API.
 */
Void ACPY2_setSrcFrameIndex(IDMA2_Handle handle, Int frameIndex)
{
    handle->params.srcFrameIndex = frameIndex;

    /*
     * For 2D to 2D transfers, src and dst indices must be set to same
     * value.
     */
    if (handle->params.xType == IDMA2_2D2D) {
        handle->params.dstFrameIndex = frameIndex;
    }

    /*
     * The idx register value is recomputed
     */
    handle->config.idx &= 0x0000FFFF;  //clear the frame index field
    handle->config.idx |=
        EDMA_FMK(IDX,FRMIDX,(handle->params.srcFrameIndex));
}
```

3.2.8 ACPY2_setDstFrameIndex()

ACPY2_setDstFrameIndex() allows an algorithm to rapidly change the destination frame index parameter of an IDMA2 channel. It also converts the frameIndex value to a corresponding bit field to be written to the count pseudo-register and stores this bit field in the channel object. Note that on the C6x1x EDMA device, ACPY2_setDstFrameIndex() has the same behavior as ACPY2_setSrcFrameIndex() in 2D to 2D transfers since the hardware only supports one frame index value.

```
/*
 *  ======== ACPY2_setDstFrameIndex ========
 *  Rapidly configure the destination frame index parameter of an
 *  IDMA2 channel.  Note that both source and destination indexes
 *  are set simultaneously on the C6x1x with this API.
 */
Void ACPY2_setDstFrameIndex(IDMA2_Handle handle, Int frameIndex)
{
    handle->params.dstFrameIndex = frameIndex;

    /*
     * For 2D to 2D transfers, src and dst indices must be set to same
     * value.
     */
    if (handle->params.xType == IDMA2_2D2D) {
        handle->params.srcFrameIndex = frameIndex;
    }

    /*
     * The idx register value is recomputed
     */
    handle->config.idx &= 0x0000FFFF;  //clear the frame index field
    handle->config.idx |=
        EDMA_FMK(IDX,FRMIDX,(handle->params.dstFrameIndex));
}
```

3.2.9 ACPY2_start() and ACPY2_startAligned

The ACPY2 library implements two interface functions to submit DMA transfer requests: `ACPY2_start()` and `ACPY2_startAligned()`. The only operational difference between `ACPY2_startAligned()` and `ACPY2_start()` is the additional requirement by `ACPY2_startAligned()` for its source and destination addresses and indexes to be properly aligned with respect to the configured element size.

The supplied `ACPY2_startAligned()` implementation described in this section, whose operation is outlined in Figure 5, offers very high performance when the source and destination addresses and indexes are properly aligned.

The `ACPY2_start()` implementation makes no assumptions on the alignment of the source and destination addresses. It accepts addresses at any alignment and adjusts the transfer parameters (including element size, number of elements, transfer type) to transparently perform the desired transfer using the given alignment. It is intended to simplify algorithm development in the initial states. `ACPY2_start()` thus strives to maintain simplicity while maintaining reasonable levels of performance. The `ACPY2_startAligned()` API, on the other hand, makes no runtime checks on the alignment and performs the transfer using the configured transfer settings of the channel. Passing source or destination addresses (or indexes) with incorrect alignment with respect to the configured element size of the DMA handle will result in unspecified behavior.

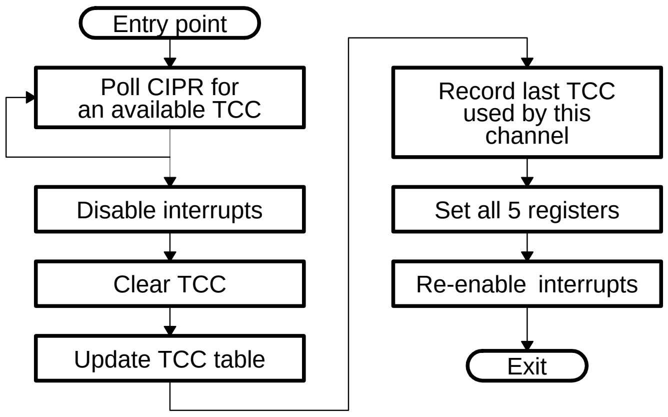

Figure 5. Flowchart for ACPY2_startAligned

As shown in Figure 5, `ACPY2_startAligned()` starts by polling the CIPR register for an available TCC to use with ACPY2. When it finds one, it reserves it by clearing the corresponding bit in the CIPR.

At this point, `ACPY2_startAligned()` updates the TCC table. As shown in Figure 6, the TCC table is a sixteen-entry array of IDMA2_Handle's. It is initialized to contain zeros (0) in acpy2_init.c and has a one-to-one correspondence with the 16 TCCs available on the C6211/ C6711. `ACPY2_startAligned()` records the logical channel handle for the transfer in the corresponding entry of the TCC table. Furthermore, it records the TCC it reserved as the last TCC used by this channel in its IDMA2_Obj structure. Both of these records are necessary to ensure proper execution of subsequent `ACPY2_wait()` and `ACPY2_complete()` calls on this channel.

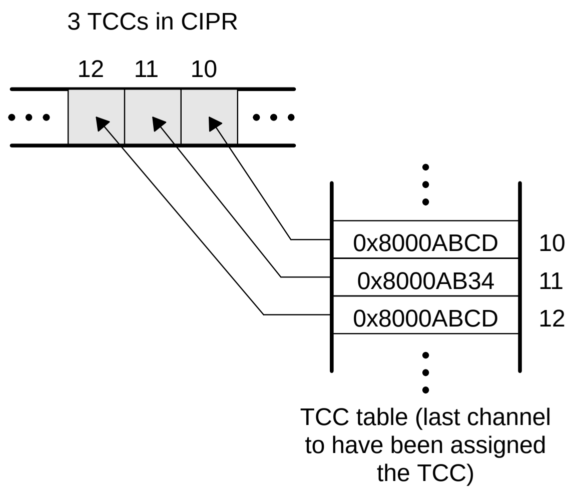

Figure 6. TCC Table and TCCs

Finally, the transfer is submitted by writing the appropriate values to all five QDMA registers, with the help of the arguments of the function and the config field in the channel's IDMA2_Obj structure.

The implementation of `ACPY2_start()` is sketched in the flowchart in Figure 7. ACPY2_start optimizes the element size of the transfers based on the alignment of the source address, the destination address, the element count, and the stride of the transfer. For example, if the channel has previously been configured as a 1D to 1D 8-bit channel, but 80 bytes (a multiple of 4) is to be transferred from a 32-bit aligned source address to a 32-bit aligned destination address, then it is possible to schedule the transfer as a 32-bit element size transfer instead of an 8-bit transfer to speed it up. Subsequent steps of ACPY2_start uses the same logic used in ACPY2_startAligned as explained above for polling and managing the TCCs and configuring the DMA hardware.

Note that this optimization is skipped when element indexing is used, or in other words, when there is a non-zero gap between consecutive elements in a transfer. Instead, the transfer is automatically scheduled as an 8-bit transfer. This is a conservative approach to ensure correctness in non-aligned transfers, while keeping the overhead of transfer submission to a minimum.

As a rule of thumb, algorithm developers are encouraged to use aligned buffers and indexes at all times in DMA transfers to minimize transfer submission overhead with the use of ACPY2_startaligned.

A highly optimized assembly version of both functions are provided with this application report. Since we do not assume knowledge of C6000 assembly language, we provide the functionally equivalent C- model implementation of both functions in this section.

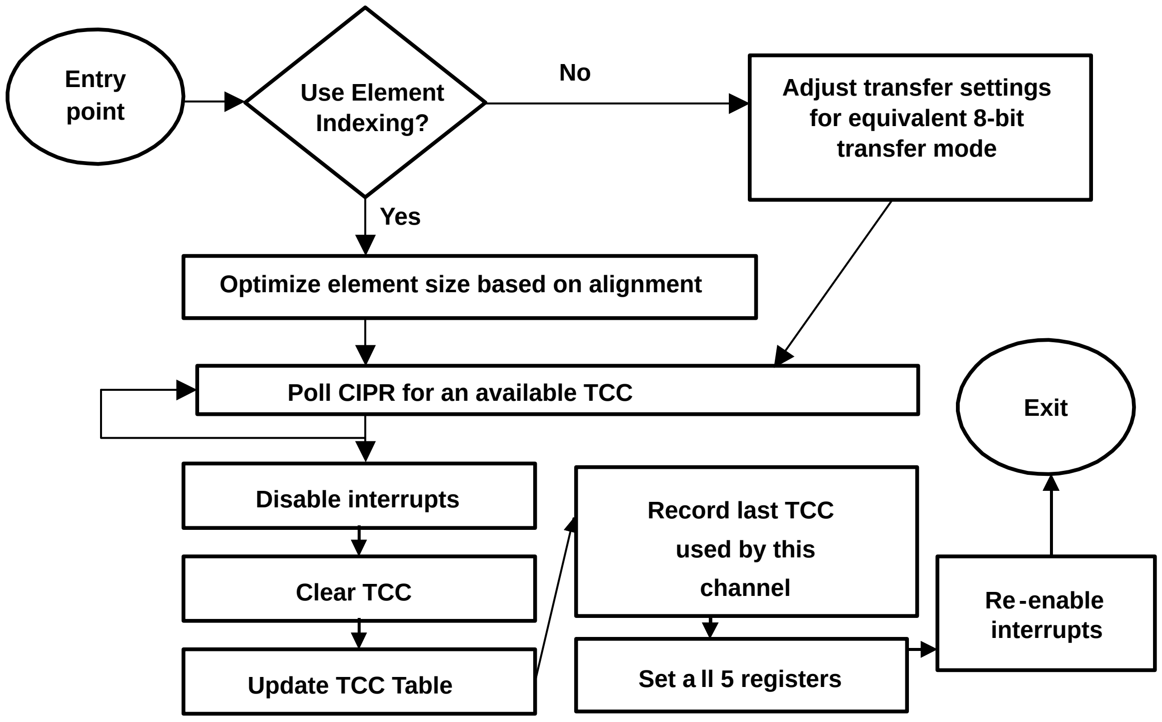
Entry point
Use Element Indexing?
No
Adjust transfer settings for equivalent 8-bit transfer mode
Yes
Optimize element size based on alignment
Poll CIPR for an available TCC
Exit
Disable interrupts
Record last TCC used by this channel
Clear TCC
Re-enable interrupts
Update TCC Table
Set all 5 registers

Figure 7. ACPY2_start Flowchart

```
/*
 *  ======== acpy2_startAligned.c ========
 */
Void ACPY2_startAligned(IDMA2_Handle handle, Void * src, Void *dst, Uns cnt)
{
    Int csr;
    Int temp;     //variable used to check available TCCs
    Int usedTCC;  //actual TCC chosen for present transfer
    Uns * base = (Uns *)_EDMA_QOPT_ADDR;  //base address of QDMA regs

    //Disable interrupts when modifying QDMA registers
    csr = HWI_disable();

    //Determines if there are TCCs available for ACPY2's use
    temp = EDMA_RGET(CIPR) & _ACPY2_TCCmask;
    base[_EDMA_QSRC_OFFSET]  = (Uns)src;
    base[_EDMA_QDST_OFFSET]  = (Uns)dst;

    while(temp == 0)
    {
        HWI_restore(csr);
        csr = HWI_disable();
        temp = EDMA_RGET(CIPR) & _ACPY2_TCCmask;
        base[_EDMA_QSRC_OFFSET]  = (Uns)src;
        base[_EDMA_QDST_OFFSET]  = (Uns)dst;
    }

    //Calculate the TCC number used
    usedTCC = 31 - _lmbd(1, temp);
    //Clear TCC bit in CIPR to record next transfer completion event
    EDMA_RSET(CIPR,1 << usedTCC);

    //Record in TCC table as the last handle to have used this TCC
    _ACPY2_TCCTable[usedTCC] = handle;

    //Record this TCC as the last TCC used by this handle
    handle->lastTCC = usedTCC;

    //Write to QDMA registers.
    base[_EDMA_QIDX_OFFSET] = (handle->config).idx;
    base[_EDMA_QOPT_OFFSET] = (handle->config).opt | (usedTCC << 16);
    if(handle->params.xType != IDMA2_1D1D)
    {
      base[_EDMA_QSCNT_OFFSET] = (handle->config).cnt + cnt;
    }
    else
    {
        base[_EDMA_QSCNT_OFFSET]  = (Uns)cnt;
    }

    HWI_restore(csr); //reenable interrupts

}

static inline Void elemIdxAdjust(IDMA2_Handle handle, Uns * opt, Uns * idx, Uns *
cnt);

Void ACPY2_start(IDMA2_Handle handle, Void * src, Void * dst, Uns cnt)
{
    Int csr;
    Int temp;                   //variable used to check available TCCs
    Int usedTCC;                //actual TCC chosen for present transfer
    Int alignment;
    Uns opt = handle->config.opt;
    Uns idx = handle->config.idx;
    Uns * base = (Uns *)_EDMA_QOPT_ADDR;  //base address of QDMA regs

    //Check to see if element indexing is used
    if ((handle->params.srcElementIndex != 0) ||
        (handle->params.dstElementIndex != 0)) {
        elemIdxAdjust(handle, &opt, &idx, &cnt);
    }
    else {
        //Perform alignment checking to optimize transfer
        alignment = 0;
        if (handle->params.elemSize == IDMA2_ELEM32) {
            if (handle->params.xType == IDMA2_1D1D) {
                alignment = ((Uns)src | (Uns)dst);
            }
            else {
                alignment = ((handle->params.srcElementIndex) |
                    (handle->params.dstElementIndex) |
                    (Uns)src | (Uns)dst);
            }
            if ((alignment & 0x1) == 0) {  //at least 16-bit aligned
                if ((alignment & 0x2) != 0) {  //16bit xfer
                    cnt <<= 1;
                    opt |= 0x08000000;
                }
            }
            else {  //not aligned => 8 bit transfer needed
```

```
                cnt <<= 2;
                opt |= 0x10000000;
            }
        }
        else if (handle ->params.elemSize == IDMA2_ELEM16) {
            if (handle->params.xType == IDMA2_1D1D) {
                alignment = ((Uns)src | (Uns)dst | (cnt << 1));
            }
            else {
                alignment = ((handle->params.srcElementIndex) |
                    (handle->params.dstElementIndex) | (Uns)src
                    | (Uns)dst | (cnt << 1));
            }
            if ((alignment & 0x1) == 0) { //at least 16-bit aligned
                if ((alignment & 0x2) == 0) {  //32-bit xfers
                    cnt >>= 1;
                    opt &= 0xE7FFFFFF;  //clear element size field
                }
            }
            else {  //not aligned => 8 bit transfer needed
                cnt <<= 1;
                opt &= 0xE7FFFFFF;  //clear element size field
                opt |= 0x10000000;
            }
        }
        else {
            if ((handle->params.xType) != IDMA2_1D1D) {
                alignment = ((handle->params.srcElementIndex) |
                    (handle->params.dstElementIndex) | (Uns)src | (Uns)dst
                    | cnt);

            }
            else {
                alignment = ((Uns)src | (Uns) dst | (Uns)cnt);
            }
            if ((alignment & 0x1) == 0) { //at least 16-bit aligned
                cnt >>= 1;
                opt &= 0xE7FFFFFF;  //clear element size field
                opt |= 0x08000000;
                if ((alignment & 0x2) == 0) {  //32 bit xfer
                    opt &= 0xE7FFFFFF;  //clear element size field
                    cnt >>= 1;
                }
            }
        }
    }

    //Disable interrupts when modifying QDMA registers
    csr = HWI_disable();

    //Determines if there are TCCs available for ACPY2’s use
    temp = EDMA_RGET(CIPR) & _ACPY2_TCCmask;
    base[_EDMA_QSRC_OFFSET]  = (Uns)src;
    base[_EDMA_QDST_OFFSET]  = (Uns)dst;

    while(temp == 0)
    {
        HWI_restore(csr);
        csr = HWI_disable();
        temp = EDMA_RGET(CIPR) & _ACPY2_TCCmask;
        base[_EDMA_QSRC_OFFSET]  = (Uns)src;
        base[_EDMA_QDST_OFFSET]  = (Uns)dst;
    }
    //Calculate the TCC number used
    usedTCC = 31 - _lmbd(1, temp);

    //Clear TCC bit in CIPR to record next transfer completion event
    EDMA_RSET(CIPR,1 << usedTCC);

    //Record in TCC table as the last handle to have used this TCC
    _ACPY2_TCCTable[usedTCC] = handle;

    //Record this TCC as the last TCC used by this handle
    handle->lastTCC = usedTCC;

    //Write to QDMA registers.
    base[_EDMA_QIDX_OFFSET] = idx;
    base[_EDMA_QOPT_OFFSET] = opt | (usedTCC << 16);
    if(handle->params.xType != IDMA2_1D1D)
    {
        base[_EDMA_QSCNT_OFFSET] = (handle->config).cnt + (Uns)cnt;
    }
    else
    {
        base[_EDMA_QSCNT_OFFSET]  = (Uns)cnt;
    }

    HWI_restore(csr); //reenable interrupts

}

static inline Void elemIdxAdjust(IDMA2_Handle handle, Uns * opt, Uns * idx,
    Uns * cnt)
{
    Uns elementSize;

    /*
     * When element index is used, go with the conservative approach
     * to schedule everything as 8-bit transfers to ensure there is no
     * alignment issues
     */
    *opt &= 0xE7FFFFFF;  //clear element size field
    *opt |= 0x10000000;  // Mask for 8-bit element size

    //elementSize = {1 (for 8bit), 2 (for 16bit), 4 for (32bit)}
    elementSize = (handle->params.elemSize) * 2;
    if (elementSize == 0) {
        elementSize++;
    }

    //new FIdx = EIdx - esize
    *idx = (handle->config.idx - elementSize) << 16;

    //new numFrames = cnt - 1
    *cnt -= 1;
    *cnt = *cnt << 16;

    //new # element/frame = element size
    *cnt = *cnt + elementSize;

    *opt &= 0xFC9FFFFF;  //clear SUM and DUM addressing fields

    if ((handle->params.srcElementIndex != 0) &&
        (handle->params.dstElementIndex != 0)) {
        //set as 2D2D transfer if both element indices are set
        *opt |= 0x05A00000;
    }
    else if (handle->params.srcElementIndex == 0) {
        //set as 1D2D transfer if only dstEIdx is used
        *opt |= 0x01A00000;
    }
    else if (handle->params.dstElementIndex == 0) {
        //set as 2D1D transfer if only srcEIdx is used
        *opt |= 0x05200000;
    }
}
```

3.2.10 ACPY2_wait()

`ACPY2_wait()` uses the entry in the TCC table corresponding to the last TCC used by a specific logical channel x to find out if all transfers issued on the channel has completed. If the lastTCC entry does not correspond to channel x's handle, another channel y has used channel x's last TCC. From this information, it can be inferred that all transfers on channel x have completed. Otherwise, x is the last channel to use the last TCC and the CIPR should be polled until the TCC becomes available.

```
/*
 * ======== ACPY2_wait ========
 * wait for the data transfers to complete
 */
Void ACPY2_wait(IDMA2_Handle handle)
{
    Uint32 mask = 1 << (handle->lastTCC);

    /*
     * If the entry in the table corresponding to the last TCC
     * used by this handle matches the handle itself, then we need
     * to wait for the TCC bit to get set to ensure transfer completion.
     * Otherwise, the TCC has already been assigned to a subsequent
     * transfer, meaning the last transfer on this handle has already
     * completed, and there is no need to wait.
     */
    if (_ACPY2_TCCTable[handle->lastTCC] == handle) {
        //This loop will not get optimized out because EDMA_RGET(CIPR)
        //macro maps into (volatile)(address_of_CIPR)
        while ((EDMA_RGET(CIPR) & mask) != mask) {
            ;
        }
    }
}
```

3.2.11 ACPY2_complete()

Similar to `ACPY2_wait()`, `ACPY2_complete()` checks the entry in the TCC table corresponding to the last TCC used by a specific logical channel and the CIPR bit of the TCC to determine transfer completion status. It does not wait for transfers to complete. Instead, it returns 1 if transfers are complete and 0 if transfers are not complete.

```
/*
 * ======== ACPY2_complete ========
 * Check to see if the data transfers have completed
 */


Int ACPY2_complete(IDMA2_Handle handle)
{
    Uint32 mask = 1 << (handle->lastTCC);

    /*
     * If the entry in the table corresponding to the last TCC
     *     used by this handle matches the handle itself, then we need
     *     to check the TCC bit to verify transfer completion.
     *     Otherwise, the TCC has already been assigned to a subsequent
     *     transfer, meaning the last transfer on this handle has already
     *     completed.
     */
    if ((_ACPY2_TCCTable[handle->lastTCC] == handle) &&
        !(EDMA_RGET(CIPR) & mask) ) {
        return (0);  //Not complete
    }
    else {
        return (1);  //complete
    }
}
```

4 The fastcopytest Example

To show how the ACPY2 APIs 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.

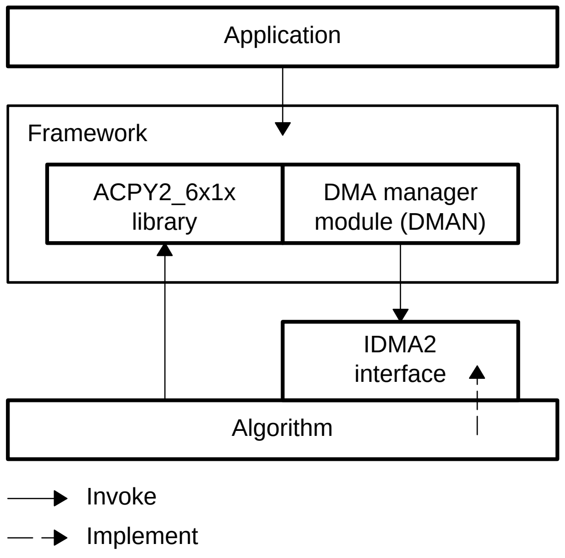

Figure 8. Dependencies in the fastcopytest Example

As Figure 8 shows, to simplify interaction between the IDMA2 interface and the ACPY2 library, a DMA manager module (DMAN) 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 IDMA2 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 ACPY2 APIs. 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 *TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609), for more details about the DMA-related cache coherence issues that are addressed in this example code.

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.

5 Conclusion

Using the new DMA guidelines in TMS320 DSP Algorithm Standard Developer’s Kit 2.5, you can easily implement a DMA manager for eXpressDSP-compliant algorithms. The example code provided with this application note should provide a good starting point for building more sophisticated DMA managers and understanding the roles of the IDMA2 and ACPY2 interfaces in a system containing DMA-based algorithms.

6 Installation of Example Code

To install the example code attached to this application note, simply unzip the file into %TI_DIR%\myprojects directory (where %TI_DIR% is the installation directory of Code Composer Studio, e.g. c:\ti). It will create a directory spra789 and a series of subdirectories that contains the following modules:

- ACPY2_6X1X - The ACPY2 implementation. It is optimized for C6211 and C6711 EDMA devices and is compatible with C6414, C6415 and C6416 devices. However, it does not make use of the C64x EDMA extensions such as larger number of hardware queues and transfer completion.
- DMAN - The DMA management module used in conjunction with the ACPY2 library to provide DMA service to XDAIS algorithms implementing the IDMA2 interface.
- FCPY_TI - An algorithm that shows an example of an IDMA2 interface and how DMA transfers can be used inside an eXpressDSP-compliant algorithm.
- FASTCOPYTEST - An example application that instantiate the FCPY_TI algorithm and uses it to copy data.

Each module can be individually built using the .pjt file available under the directory \spra789\src\(mod_name)\, where (mod_name) corresponds to the module name.

7 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 ACPY2 implementation to illustrate how IDMA2 functions are implemented and how ACPY2 functions are used to perform DMA transfers.

Example A-1. Code Example for the fastcopytest

```
/*
 *  ======== 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(input,“.image:ext_sect1”);
#pragma DATA_SECTION(output,”.image:ext_sect2”);
#pragma DATA_ALIGN(input,128);   // aligned on cache boundary
#pragma DATA_ALIGN(output,128);  // aligned on cache boundary
#include <std.h>
#include <sys.h>
#include <log.h>
#include <csl_cache.h>
#include <alg.h>
#include <ialg.h>
#include <ifcpy.h>
#include <dman.h>
#include <acpy2_6x1x.h>
#include <fcpy.h>
#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 */
//2D 64x64 Input and output data buffers
int input[INPUTSIZE];
int output[OUTPUTSIZE];
extern far Int INTERNALHEAP;
extern far Int EXTERNALHEAP;
extern LOG_Obj LOG_myLog;
extern far IFCPY_Fxns FCPY_IFCPY;        /* FCPY algorithm’s v-table */
extern far IDMA2_Fxns FCPY_IDMA2;        /* FCPY algorithm’s IDMA2 v-table */

Int main(Void)
{
    Int i;
    FCPY_Params fcpyParams;
    FCPY_Handle alg;
    Bool errorFlag = FALSE;
    IFCPY_Fxns * fxns = (IFCPY_Fxns *)&FCPY_IFCPY;
    FCPY_init();   //Initialize the framework
```

```
// Set up 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 ((alg = FCPY_create(fxns, &fcpyParams)) == NULL) {
  SYS_abort(“Could not create algorithm instance”);
}

//
// Initialize DMA manager and ACPY2 library for XDAIS algorithms
// and grant DMA resources
//
ACPY2_6X1X_init();
DMAN_init();
DMAN_setup(INTERNALHEAP);
if (DMAN_addAlg((IALG_Handle)alg, &FCPY_IDMA2) == FALSE) {
  SYS_abort(“Problem adding algorithm’s dma resources”);
}
CACHE_clean(CACHE_L2ALL, NULL, NULL);

// Initialize data arrays
for (i = 0; i < INPUTSIZE; i++)
{
    input[i] = i;
    output[i] = 0xDEADBEEF;
}
CACHE_clean(CACHE_L2ALL, NULL, NULL);

//
// Copy input to the output one quadrant at a time
//
// Quadrant 2
FCPY_apply((FCPY_Handle)alg, input, output);
// Quadrant 1
FCPY_apply((FCPY_Handle)alg, input + (SSTRIDE/4), output +
(DSTRIDE/4));
// Quadrant 3
FCPY_apply((FCPY_Handle)alg, input + (INPUTSIZE/2),
            output + (OUTPUTSIZE/2));
// Quadrant 4
FCPY_apply((FCPY_Handle)alg, input + (INPUTSIZE/2) + (SSTRIDE/4),
            output + (OUTPUTSIZE/2) + (DSTRIDE/4));

// Verify output
for (i = 0; i < OUTPUTSIZE; i++)
{
  if (output[i] != i) {
```

```
                LOG_printf(&LOG_myLog, “ %d th element in output should not be
                %d.\n”
                      , i, output[i]);
                errorFlag = TRUE;
            }
        }

        if (errorFlag == FALSE) {
            LOG_printf(&LOG_myLog, “pass \n”);
        }

        //
        // Withdraw DMA resources from algorithm and deinitialize the DMA
        // manager and ACPY2 library
        //
        if (DMAN_removeAlg((IALG_Handle)alg, &FCPY_IDMA2) == FALSE) {
            SYS_abort(”Problem removing algorithm’s dma resources”);
        }
        // delete the algorithm instance
        ALG_delete((IALG_Handle)alg);
        // module finalization
        DMAN_exit();
        ACPY2_6X1X_exit();
        FCPY_exit();    //Deinitialize the framework

        return (0);
    }
```