Application Report
SPRA842 – February 2003

An RF5 Adaptation for IDMA2-Based Algorithms: A JPEG Example

Vincent Wan, Alan Campbell *Texas Instruments*

ABSTRACT

Reference Framework 5 (RF5) for eXpressDSP Software is provided as starterware for developing applications that use DSP/BIOS and the TMS320 DSP Algorithm Standard (also known as XDAIS). The XDAIS standard has introduced DMA specifications on how algorithms should use the DMA resources, and it is possible to incorporate an algorithm adhering to these specifications into RF5.

The purpose of this application note is to guide the user through an adaptation of RF5 that incorporates a JPEG encode/decode channel into RF5. The goal is to show how to add XDAIS algorithms that implement the IDMA2 interface into RF5, with the help of DMA management modules for XDAIS algorithms. The IDMA2 interface allows the algorithm to request and receive logical DMA resources.

This application note also lets the user gain insight into the performance improvements possible by converting CSL DAT-based algorithms to IDMA2-based algorithms.

The intended audience is primarily algorithm consumers who desire to integrate XDAIS algorithms that use DMA internally into RF5. However, this application note is also useful for algorithm producers who wish to gain insight into the benefits of adhering to the XDAIS DMA specifications.

Contents

Figures

Tables

1 Overview

Reference Framework Level 5 (RF5) is intended to enable designers to create extensive applications that use numerous algorithms or channels.

In October 2002, the XDAIS Developer's Kit v2.5 upgraded the specification for using the DMA resource within algorithms. This upgrade introduced:

- IDMA2, an enhancement to IDMA, implemented by the algorithm producer to communicate the algorithm's logical DMA resource requirements to the framework. New parameter fields provide finer granularity in configuring transfers.
- ACPY2, an enhancement to ACPY, called by the algorithm at runtime to perform DMA transfers. New APIs reduce the overhead of DMA reconfiguration.

In particular, the algorithms adhering to these specifications are XDAIS-compliant (XDAIS is also known as the TMS320 DSP Algorithm Standard).

This application note guides the user through an adaptation of RF5.

The primary goal is to add XDAIS algorithms that implement the IDMA2 interface into RF5, with the help of a DMA manager for XDAIS algorithms. Detailed code excerpts are given to help system integrators add IDMA2-based algorithms into RF5-based video applications, but they could similarly apply to integrators working on other types of applications such as speech recognition.

The second objective is to demonstrate the benefits of the new ACPY2 APIs in real video algorithms such as TI's JPEG encoder and decoder. Performance is compared between an existing JPEG implementation utilizing the Chip Support Library (CSL) DAT module to submit DMA transfers, and a newly refitted IDMA2-based JPEG implementation. The enhanced ability to rapidly submit aligned transfers and to use multiple DMA hardware queues provides a healthy reduction in MIPS.

This application note assumes that the user has familiarity with the XDAIS IDMA2 and ACPY2 specifications, and that the user has read the application notes *Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High-Density System* (SPRA795) and *A Video, Audio, Networking System on the C64x NVDK Using eXpressDSP RF5* (SPRA844).

The next section discusses the work done in updating JPEG algorithms to make them adhere to the newest XDAIS DMA specifications, which address performance concerns in the use of DMA. It then walks the user through inserting such IDMA2-based algorithms into TI's Reference Framework Level 5.

2 Adding JPEG Algorithms to RF5 as IDMA2-Based Algorithms

We start with the JPEGENC_TI and JPEGDEC_TI algorithms that directly call the CSL DAT module. Our goal is to have them request DMA resources via IDMA2 instead and then use the resources via ACPY2 at runtime.

The procedure to transform them into IDMA2-based algorithms with the same functionality is then outlined. Suggestions on how to leverage the new ACPY2 optimizations are given.

Finally the algorithms are inserted into RF5. While RF5 does accept traditional processing methods, it has interfaces, APIs, and framework optimizations to specifically handle XDAIS algorithms. Adding a second IDMA2-based algorithm becomes trivial.

2.1 The Software Architecture and Data Flow

Throughout our discussion, diagrams show the data flow. The symbols used in the diagrams are shown in Figure 1.

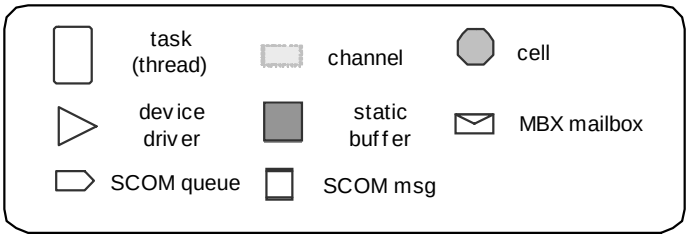

Figure 1. Diagram Symbols

The overall application data flow is shown in Figure 2. Our initial baseline is the RF5-based video application supplied on the Ateme NVDK software release v1.3. It is the trimmed down version of the video, audio, and networking application described in SPRA844. The video processing thread of the trimmed down version is shown in Figure 3.

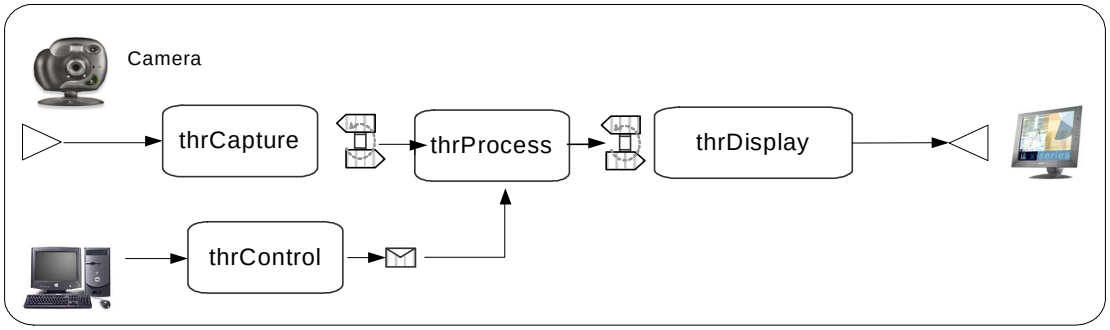

Figure 2. Application Data Flow

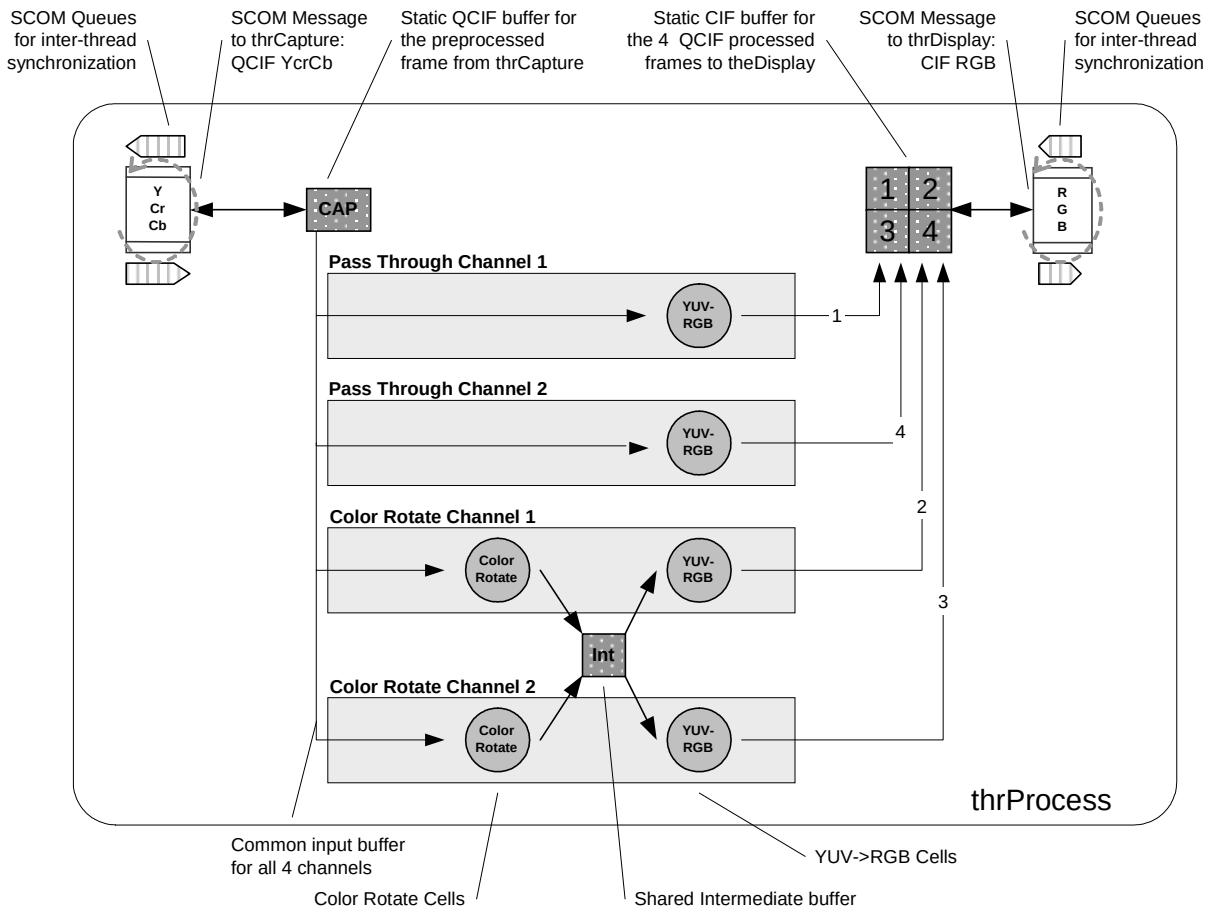

Figure 3. Initial Video Processing Thread

Each cell contains a XDAIS-compliant algorithm. Our goal is to introduce 2 cells to Pass Through Channel 2 to perform JPEG encode and decode operations prior to the YUV2RGB conversion, as shown in Figure 4.

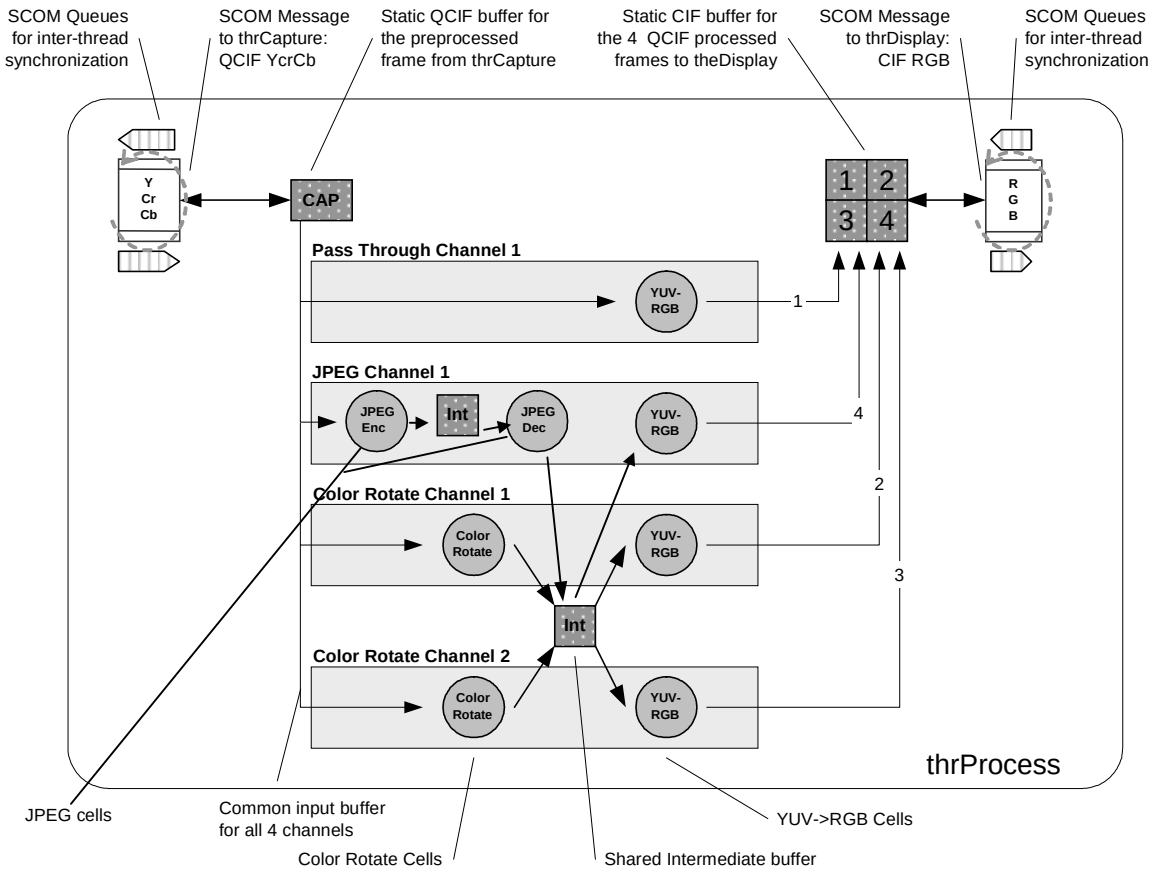

Figure 4. Modified Video Processing Thread

2.2 Suggested Directory Structure

Figure 5 shows the directory structure that could be used for the adaptation.

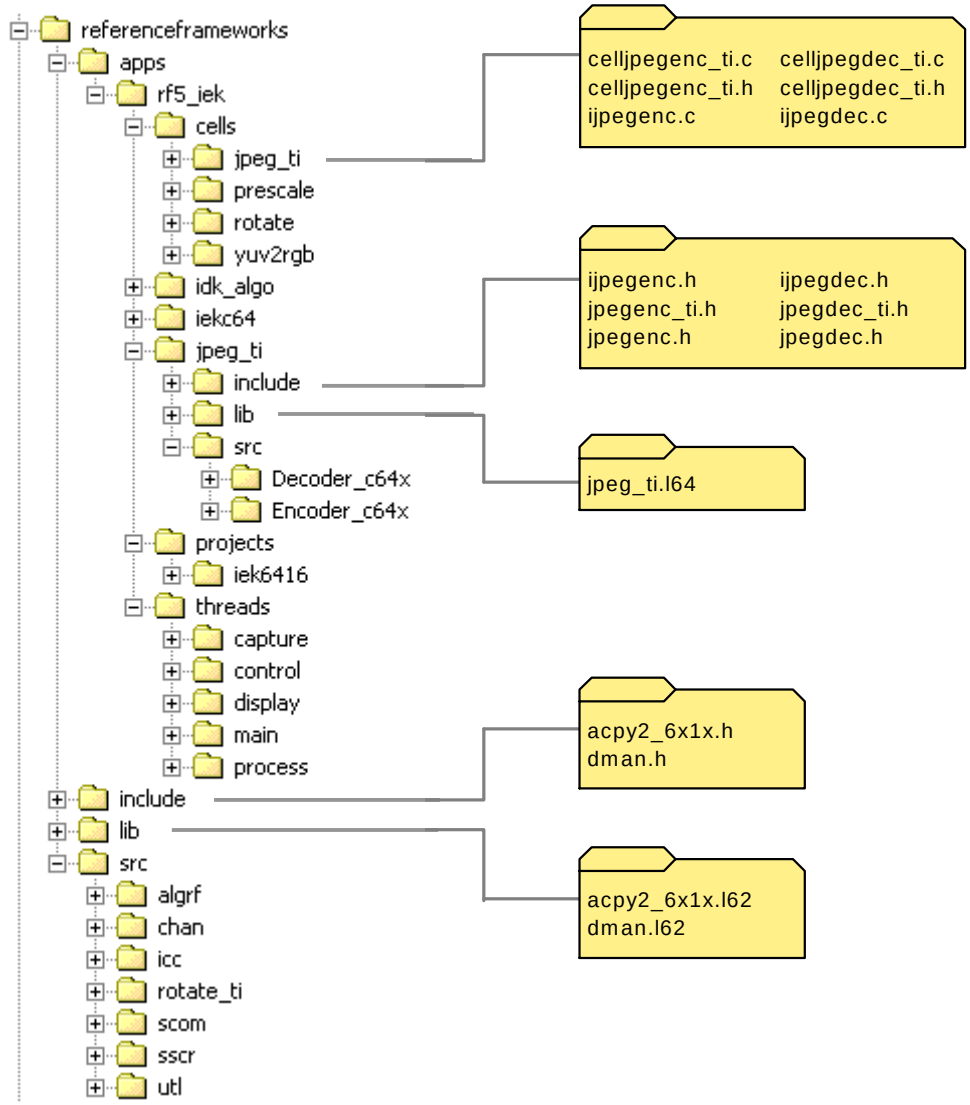

Figure 5. Suggested Directory Structure for DMA Adaptation

Folders to notice include:

- **referenceframeworks\apps\rf5_iek\cells\jpeg_ti.** Contains application-side cell implementation code the jpeg encoder and decoder algorithms. This folder is placed at this location instead of under the referenceframeworks\src folder because in this manner we can keep all application-specific material under the apps folder and make it easier to maintain. For instance, if for some reason we need to pick up the jpeg algorithms as a module, all we have to do is to make a copy of the jpeg_ti folder instead of looking under the referenceframeworks\include, referenceframeworks\lib and referenceframeworks\src folders to seek out the corresponding headers, archives and source files.

- **referenceframeworks\apps\rf5_iek\jpeg_ti.** Contains all source code for the JPEGENC_TI and JPEGDEC_TI algorithms. The lib subfolder contains the archive file linked with the application.
- **referenceframeworks\include.** Contains global header files for the DMA support modules, and RF5 module header files. Added the header files for the DMAN and ACPY2 implementations to support DMA within XDAIS algorithms.
- **referenceframeworks\lib.** Contains archive files for the DMA support modules, and RF5 module archive files. Added the archive files for the DMAN and ACPY2 implementations to support DMA within XDAIS algorithms.

2.3 Using ACPY2 APIs for DMA Transfers Within the Algorithms

The XDAIS specification requires all algorithms using DMA transfers internally to implement the IDMA2 interface and use ACPY2 APIs to configure, submit, and wait on DMA transfers.

2.3.1 Benefits

The main benefits in using the new IDMA2 and ACPY2 specifications versus their previous incarnations include the following:

- The ability to rapidly submit transfers that involve addresses aligned to the element size via the new API ACPY2_startAligned().
- The option of specifying serialization constraints under the IDMA2 interface, giving the flexibility to utilize multiple DMA hardware queues to enhance performance. The C64xx for example, has four queues. An implementation of ACPY2 might assign different priority levels to improve hardware parallelism.
- The requirement that a given DMA transfer is initiated and completed before the next transfer issued at the same serialization level can help eliminate unnecessary ACPY2_wait() calls, thereby reducing footprint and increasing performance.
- Adhering to the new XDAIS DMA rules help eliminate problems of cache coherency with external memory.

2.3.2 Converting Algorithms to Meet New XDAIS DMA Specifications

The JPEGENC_TI and JPEGDEC_TI algorithms originally use the DAT module from the Chip Support Library (CSL) to issue and wait on DMA transfers. To use ACPY2 APIs instead, the algorithm should first configure the DMA logical channels that it requests from the framework. ‘Logical’ channels represent a private DMA channel abstraction describing the type of DMA configuration. Algorithms can get smart with this concept – a different logical handle might be requested for 16 versus 32-bit transfers. Having two logical channels saves on run-time reconfiguration, and thus reduces overhead. For instance, after analyzing the JPEGENC_TI algorithm, we determined that it would benefit from having:

- Two separate 1D to 1D channels with 8-bit transfers
- Two separate 2D to 1D channels with 8-bit transfers

This is an example of how to configure four channels, two IDMA2_1D1D channels, and two IDMA2_2D1D channels:

```
/*
 * ======== Jpeg Encode function ========
 */
unsigned int   jpgenc_ti ( JPG_params  *jpg_params,
               unsigned char     **raw_img,
               unsigned char     *output_area,
               JPEGENC_SCRATCH    *scratch_buff,
               IDMA2_Handle dmaHandle0_1D1D8B,   // input logical channel 1
               IDMA2_Handle dmaHandle1_1D1D8B,   // input logical channel 2
               IDMA2_Handle dmaHandle2_2D1D8B,   // output logical channel 1
               IDMA2_Handle dmaHandle3_2D1D8B    // output logical channel 2
               )
{
    ...
    IDMA2_Params dmaParams;

    /* Configure the 1D logical channel */
    dmaParams.xType = IDMA2_1D1D;            // 1D transfer
    dmaParams.elemSize = IDMA2_ELEM8;        //8-bit element size
    dmaParams.numFrames = 0;                 // Not used in 1D1D transfer
    dmaParams.srcFrameIndex = 0;             // Not used in 1D1D transfer
    dmaParams.dstFrameIndex = 0;             // Not used in 1D1D transfer
    dmaParams.srcElementIndex = 0;
    dmaParams.dstElementIndex = 0;

    ACPY2_configure(dmaHandle0_1D1D8B, &dmaParams);
    ACPY2_configure(dmaHandle1_1D1D8B, &dmaParams);

    /* Initialize the 2D logical channels */
    // Setting the fields in the parameter structure that applies throughout the
    // duration of this encode function. The frame index will be independently modified
    // before each transfer using the corresponding ACPY2 APIs, as it changes on a per
    // transfer basis.
    dmaParams.xType = IDMA2_2D1D;
    dmaParams.numFrames = 8;

    ACPY2_configure(dmaHandle2_2D1D8B, &dmaParams);
    ACPY2_configure(dmaHandle3_2D1D8B, &dmaParams);

    ...
}
```

Alternatively, to reduce run-time overhead, this configuration could also be done at initialization time by putting it into a separate function that is called when opening the cell associated with the JPEG encode algorithm. This can be done as long as the logical channel object instances are not shared across algorithms.

To submit a transfer, the previous DAT implementation used calls such as follows:

```
// 1D to 1D transfer
xfrID_out = DAT_Copy (
               pong_ip,              // Source address
               output_area,          // Destination address
               quant_offset + dcachuff_offset + frhdr_offset + 2  // Byte count
            );

// 2D to 1D transfer
xfrID_in = DAT_Copy2D (
               DAT_2D1D,                    // Type of transfer
               (void *) raw_img[i],         // Source address
               (void *) passive_ip,         // Destination address
               (unsigned short) elem_cnt,   // Number of bytes per line
               8,                           // Number of lines
               num_samples[i] // Number of bytes between start of one line & next line
            );
```

The equivalent code that adheres to the new XDAIS DMA specifications is as follows:

```
// 1D to 1D transfer
ACPY2_start(
    dmaHandle0_1D1D8B,                                  // Logical channel handle
    pong_ip,                                            // Source address
    output_area,                                        // Destination address
    quant_offset + dcachuff_offset + frhdr_offset + 2   // Element count
);

// 2D to 1D transfer
ACPY2_setSrcFrameIndex(
    dmaHandle2_2D1D8B,                                  // Logical channel handle
    num_samples[i] - (unsigned short) elem_cnt          // Source frame index
);
ACPY2_start(
    dmaHandle2_2D1D8B,                                  // Logical channel handle
    (void *) raw_img[i],                                // Source address
    (void *) passive_ip,                                // Destination address
    (unsigned short) elem_cnt                           // Element count per frame
);
```

Calls to ACPY2_setSrcFrameIndex or ACPY2_setDstFrameIndex are necessary only if the frame index changes while the transfers are executed in a loop. Otherwise, it is always a better idea to pre-configure the frame index in the ACPY2_configure call at startup time.

To assist algorithm writers converting legacy DAT or ACPY calls into ACPY2 calls, Table 1 shows examples of equivalent transfers when using the three modules.

Table 1. Examples of Equivalent Transfers

DAT Call	Equivalent ACPY Setup	Equivalent ACPY2 Setup

DAT_copy (src, dst, byteCnt)	IDMA_Handle hdl;
	IDMA_Params params;
	
	params.xType = IDMA_1D1D;
	params.elemSize = IDMA_ELEM8;
	
	ACPY_configure(hdl, params);
	ACPY_start(hdl, src, dst, byteCnt, NULL);
		IDMA2_Handle hdl;
		IDMA2_Params params;
		
		params.xType = IDMA2_1D1D;
		params.elemSize = IDMA2_ELEM8;
		
		ACPY2_configure(hdl, params);
		ACPY2_start(hdl, src, dst, byteCnt);

DAT_copy2D (DAT_1D2D, src, dst, lineLen, lineCnt, linePitch)	IDMA_Handle hdl;
	IDMA_Params params;
	
	params.xType = IDMA_1D2D;
	params.elemSize = IDMA_ELEM8;
	params.stride = linePitch – lineLen;
	params.numFrames = lineCnt;
	
	ACPY_configure(hdl, params);
	ACPY_start(hdl, src, dst, lineLen, NULL);
		IDMA2_Handle hdl;
		IDMA2_Params params;
		
		params.xType = IDMA2_1D2D;
		params.elemSize = IDMA2_ELEM8;
		params.dstFrameIndex = linePitch – lineLen;
		params.numFrames = lineCnt;
		
		ACPY2_configure(hdl, params);
		ACPY2_start(hdl, src, dst, lineLen);

DAT_copy2D (DAT_2D1D, src, dst, lineLen, lineCnt, linePitch)	IDMA_Handle hdl;
	IDMA_Params params;
	
	params.xType = IDMA_2D1D;
	params.elemSize = IDMA_ELEM8;
	params.stride = linePitch – lineLen;
	params.numFrames = lineCnt;
	
	ACPY_configure(hdl, params);
	ACPY_start(hdl, src, dst, lineLen, NULL);
		IDMA2_Handle hdl;
		IDMA2_Params params;
		
		params.xType = IDMA2_2D1D;
		params.elemSize = IDMA2_ELEM8;
		params.srcFrameIndex = linePitch – lineLen;
		params.numFrames = lineCnt;
		
		ACPY2_configure(hdl, params);
		ACPY2_start(hdl, src, dst, lineLen);

DAT_copy2D (DAT_2D2D, src, dst, lineLen, lineCnt, linePitch)	IDMA_Handle hdl;
	IDMA_Params params;
	
	params.xType = IDMA_2D2D;
	params.elemSize = IDMA_ELEM8;
	params.stride = linePitch – lineLen;
	params.numFrames = lineCnt;
	
	ACPY_configure(hdl, params);
	ACPY_start(hdl, src, dst, lineLen, NULL);
		IDMA2_Handle hdl;
		IDMA2_Params params;
		
		params.xType = IDMA2_2D2D;
		params.elemSize = IDMA2_ELEM8;
		params.srcFrameIndex = linePitch – lineLen;
		params.dstFrameIndex = linePitch – lineLen;
		params.numFrames = lineCnt;
		
		ACPY2_configure(hdl, params);
		ACPY2_start(hdl, src, dst, lineLen);

To further utilize the flexibility offered in the ACPY2 APIs, the algorithm developer is urged to make use of the ACPY2_startAligned() API along with the highest element size possible. For example, given the source and destination addresses are aligned on 32-bit boundaries and that the number of elements in a frame is a multiple of 4, an 8-bit transfer of length 'num_dma_bytes' elements can be submitted as a 32-bit transfer using an ACPY2_startAligned() call such as the following:

```
/*
 * The following ACPY2_startAligned call requires that src and dst
 * addresses are aligned on 32-bit boundaries
 */
ACPY2_startAligned(dmaHandle1_1D1D32B, input_addr, pong_ip, num_dma_bytes/4);
```

ACPY2_startAligned() is faster than ACPY2_start() since it avoids run-time checking of source and destination addresses with respect to alignment.

For a complete example of an algorithm using IDMA2/ACPY2, refer to *Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6x1x* (SPRA789), where the source code of an algorithm called FCPY_TI is provided. This algorithm uses ACPY2 API calls to copy 2D buffers.

2.4 Adding an IDMA2 Interface

XDAIS algorithms request their DMA resources through the IDMA2 interface. When building the IDMA2 interface, the following decisions need to be made:

- How many logical channels does the algorithm use?
- Which logical channels should have their transfers serialized, with respect to one another?

Recall that transfers submitted on logical channels sharing the same serializer ID are initiated and completed by observing strong ordering, while transfers on logical channels with different serializer IDs could occur in parallel. Hence it is important to pick the serializer IDs for each logical channel accordingly both to ensure correctness and to maximize performance in the algorithm.

The JPEG encoder's IDMA2 interface implementation follows. It assigns a different serializer ID to each logical channel because this particular algorithm does not require transfers submitted on different logical channels to be serialized, and would prefer to have the transfers occur in parallel on the hardware whenever possible.

```
/*
 *  ======== jpegenc_ti_idma2.c ========
 *  JPEGENC Module - TI implementation of a JPEGENC algorithm
 *
 *  This file contains an implementation of the IDMA2 interface
 */

#pragma CODE_SECTION(JPEGENC_TI_dmaChangeChannels, ".text:dmaChangeChannels")
#pragma CODE_SECTION(JPEGENC_TI_dmaGetChannelCnt, ".text:dmaGetChannelCnt")
#pragma CODE_SECTION(JPEGENC_TI_dmaGetChannels, ".text:dmaGetChannels")
#pragma CODE_SECTION(JPEGENC_TI_dmaInit, ".text:dmaInit")

#include <std.h>

#include <ialg.h>
#include <idma2.h>

#include "jpegenc_ti_priv.h"

#define CHANNEL0 0
#define CHANNEL1 1
#define CHANNEL2 2
#define CHANNEL3 3

#define NUM_LOGICAL_CH 4

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

/*
 *  ======== JPEGENC_TI_dmaGetChannels ========
 *  Declare DMA resource requirement/holdings.
 */
Int JPEGENC_TI_dmaGetChannels(IALG_Handle handle, IDMA2_ChannelRec dmaTab[])
{
    JPEGENC_TI_Obj *jpegenc = (Void *)handle;

    /* Initial values on logical channels */
    dmaTab[CHANNEL0].handle = jpegenc->dmaHandle0_1D1D8B;
    dmaTab[CHANNEL1].handle = jpegenc->dmaHandle1_1D1D8B;
    dmaTab[CHANNEL2].handle = jpegenc->dmaHandle2_2D1D8B;
    dmaTab[CHANNEL3].handle = jpegenc->dmaHandle3_2D1D8B;
```

```
    /*
     * This particular queueId assignment allows transfers submitted on
     * different handles to potentially occur in parallel. If serialization is required
     * between two channels, they should share the same queueId.
     */
    dmaTab[CHANNEL0].queueId = 0;
    dmaTab[CHANNEL1].queueId = 1;
    dmaTab[CHANNEL2].queueId = 2;
    dmaTab[CHANNEL3].queueId = 3;

    return (NUM_LOGICAL_CH);
}

/*
 *  ======== JPEGENC_TI_dmaInit========
 *  Initialize instance object with granted logical channel.
 */
Int JPEGENC_TI_dmaInit(IALG_Handle handle, IDMA2_ChannelRec dmaTab[])
{
    JPEGENC_TI_Obj *jpegenc = (Void *)handle;

    jpegenc->dmaHandle0_1D1D8B = dmaTab[CHANNEL0].handle;
    jpegenc->dmaHandle1_1D1D8B = dmaTab[CHANNEL1].handle;
    jpegenc->dmaHandle2_2D1D8B = dmaTab[CHANNEL2].handle;
    jpegenc->dmaHandle3_2D1D8B = dmaTab[CHANNEL3].handle;

    return (IALG_EOK);
}
```

2.5 Creating the Cells

A cell is a concept introduced in RF5 for ‘wrapping’ a XDAIS algorithm. The uniform abstraction enables modules to build on top of it. For example, the RF5 CHAN module executes algorithms wrapped with a standard cellExecute method in an interface called ICELL. Moreover, the cellOpen and cellClose functions in ICELL can be used respectively to grant and remove DMA resources in the algorithm. The following code excerpt shows how this was achieved using the DMAN module supplied with SPRA789:

```
/*
 *  ======== JPEGENC_cellOpen ========
 *
 */
Bool JPEGENC_cellOpen( ICELL_Handle handle )
{
    // Grant DMA resources
    return (DMAN_addAlg((IALG_Handle)handle->algHandle, &JPEGENC_IDMA2));
}

/*
 *  ======== JPEGENC_cellClose ========
 *
 */
Bool JPEGENC_cellClose( ICELL_Handle handle )
{
    // Withdraw DMA resources from algorithm
    return (DMAN_removeAlg((IALG_Handle)handle->algHandle, &JPEGENC_IDMA2));
}
```

The following code excerpt shows an example of a cellExecute method that is called at run-time to run the algorithm:

```
/*
 *  ======== JPEGENC_cellExecute ========
 */
Bool JPEGENC_cellExecute( ICELL_Handle handle, Arg arg )
{
    int ret_val ;

    // activate instance object
    ALGRF_activate( handle->algHandle );

    ret_val = ((IJPEGENC_Fxns *)(handle->algFxns))->encode
                      (
                         (IJPEGENC_Handle)handle->algHandle,
                         (XDAS_Int8 **)handle->inputIcc[0]->buffer,
                         (XDAS_Int8 *)handle->outputIcc[0]->buffer
                      );

    // deactivate instance object
    ALGRF_deactivate( handle->algHandle );

    return (TRUE);
}
```

If the cache is enabled, cache coherency with the input and output buffers passed in to the algorithms in cellExecute can be ensured by calling the appropriate CACHE APIs. Refer to CSL documentation on how to use these APIs to control the cache.

2.6 Integration of Cells into Overall Framework

The relationships between the modules and interfaces in the framework and the algorithm are shown in Figure 6. Algorithms access DMA hardware via the “logical” DMA channel handles they request and receive from the client application. This request is done via the IDMA2 interface provided by the algorithm. In turn, the framework can use the IDMA2 interface to obtain the DMA requirements by calling the DMAN module, which takes care of the following protocol:

1. Queries the number of logical channels needed by the algorithm using dmaGetChannelCnt().
2. Obtains descriptors of the various logical channels using dmaGetChannels().
3. Instantiates and initializes the logical channel objects.
4. Gives the handles of the logical channel objects to the algorithm using dmaInit().

After obtaining these logical handles, algorithms can then submit DMA transfer requests on the logical channels through a high-performance implementation of ACPY2 run-time APIs provided by the client application.

The ICELL interface is implemented by the application to structure the way algorithms are instantiated and to ease their use by the channel abstraction provided by the framework through the CHAN module. It also provides a placeholder for code to grant and remove DMA resources in the algorithm, as we saw from the previous section.

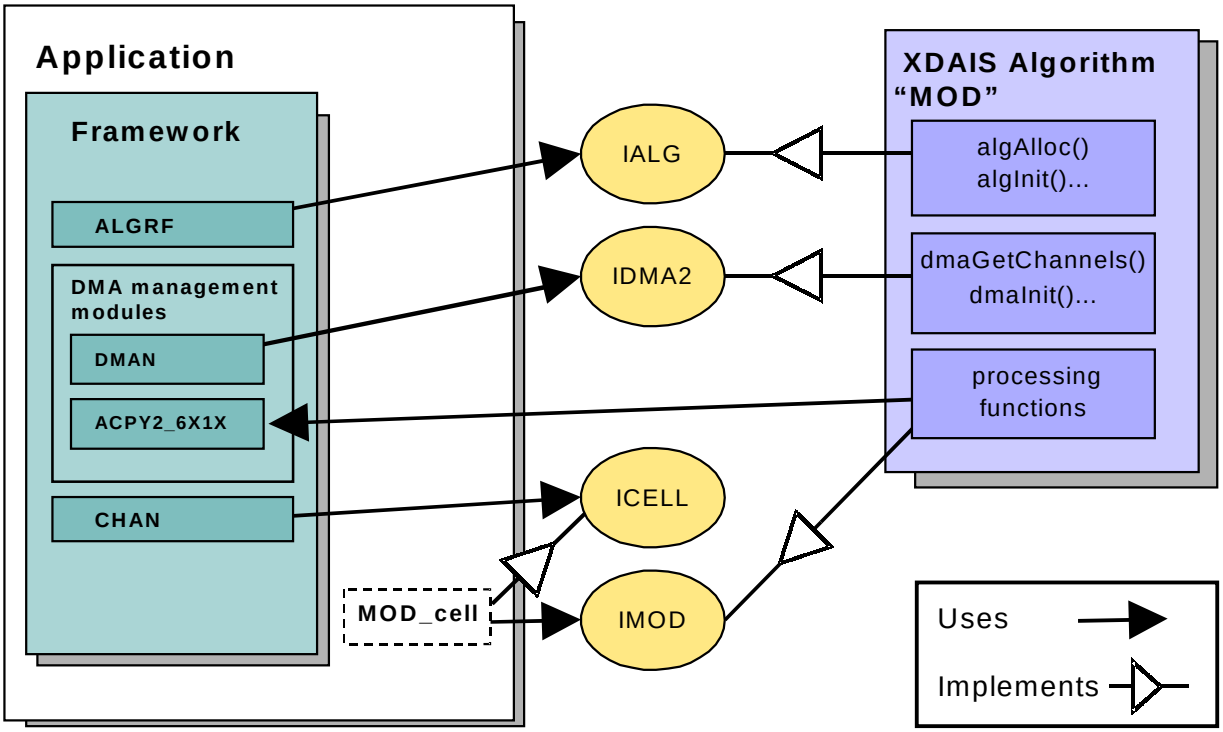

Figure 6. Relationships of Modules and Interfaces Involved in DMA Support

Integrating the new cells and algorithms into the framework requires:

- Initializing all related modules:

```
Void appThreadInit()
{

...

    // Use of DMA in XDAIS algos requires the following modules' init
    ACPY2_6X1X_init();
    DMAN_init();
    DMAN_setup(INTERNALHEAP);

...
}
```

- Setting up the intermediate processing buffers and wrapping them with ICC objects. For example:

```
/*
 *  cell 0 - JPEGENC: create an input and output linear ICC.
 *  The address to the input ICC will be set in the thrProcessRun()
 *  function via ICC_setBuf().
 */
cell = &thrProcess.jpegCells[ chanNum * CHJPEGNUMCELLS ];
*cell                = defaultCell;
cell->name           = "JPEGENC";
cell->cellFxns       = &JPEGENC_CELLFXNS;
cell->cellEnv        = NULL;
cell->algFxns        = (IALG_Fxns *)&JPEGENC_IJPEGENC;
cell->algParams      = (IALG_Params *)&jpegencParams;
cell->scrBucketIndex = VIDEOPROCSCRBUCKET;

//Set size later when input buffer becomes available
inputIcc = (ICC_Handle)ICC_linearCreate( NULL, 0);
UTL_assert( inputIcc != NULL);

outputIcc = (ICC_Handle)ICC_linearCreate(bufCompressed, COMPRESSED_BUF_SIZE);
UTL_assert( outputIcc != NULL);

rc = CHAN_regCell ( cell, &inputIcc, 1, &outputIcc, 1 );
```

- Adding the cell code, as shown in Section 2.5, to the application project and linking in the necessary libraries.

It is preferable to initialize the ACPY2 module after the initialization of other modules that also use the DMA resource. This is because some DMA-based drivers might request specific transfer completion codes (TCCs) on the C6x EDMA. TCCs are centrally managed by the Chip Support Library (CSL), and are granted to the ACPY2 library when the latter makes a call to EDMA_intAlloc(-1) in the ACPY2_init() API. The –1 argument tells the TCC manager to return an unused transfer completion code on the system. Hence, initializing the ACPY2 module last ensures that ACPY2 gets assigned completion codes from the remaining pool of unused TCCs on the system, which is fine as this implementation of ACPY2 is flexible in terms of which TCCs it uses. It requires three TCCs to support up to three outstanding ACPY2 transfer requests at any given time, but they can be any TCC values. These TCCs are dynamically assigned to individual logical channels when submitting each transfer request.

When setting up external memory buffers for the IDMA2-based algorithms, it is a good habit to align the data buffers on 128-byte boundary and to use buffers sized as a multiple of 128 bytes, as per XDAIS DMA Rule 7. This is to ensure correctness when either the L1D or L2 cache is enabled on the system, since the cache line size is 64 bytes for the L1D cache and 128 bytes for the L2 cache. More information on cache coherency issues can be found in *DMA for eXpressDSP-Compliant Algorithm Producers and Consumers* (SPRA445).

2.7 Performance Comparison

Table 2 shows the gain in performance that was obtained by changing the JPEG algorithms into IDMA2-based algorithms optimized through serialization and aligned transfers.

Table 2. Performance Gain of JPEG Channel

	Performance of JPEG Channel Using DAT-Based JPEG Algorithms	Performance of JPEG Channel Using Current XDAIS DMA Specifications
Execution time of JPEG channel (in milliseconds)	7.26	6.82

This represents a 6% performance improvement when moving from a DAT-based implementation to one that uses IDMA2/ACPY2. Clearly, even more significant gains can be made in heavily DMA-based algorithms such as H.263 or MPEG2 by carefully avoiding unnecessary reconfiguration of logical channels. These algorithms typically have to DMA many small blocks, hence any reconfiguration overhead is a big performance hit.

2.8 Some Useful Guidelines

A couple of guidelines that could make life easier for everyone:

- The ACPY2 implementation as shipped in SPRA789 was designed for the C6211/6711, which has only two EDMA priority queues available for asynchronous QDMA transfers. On the other hand, C64x devices have four EDMA priority queues. Because the queues used are not configurable on the application-side with the present implementation of ACPY2, simply dropping the present ACPY2 implementation into a C64x system means algorithms run with ‘hardcoded’ high- and medium-priority queues. Users should modify their ACPY2 implementation (for example, as in SPRA789) to use the appropriate queues and avoid conflict with other modules that use DMA in their system.

  Determining the number of queues to assign to ACPY2 is a game of tradeoffs. Assigning more queues to ACPY2 could potentially improve algorithms’ performance. However, a rule of thumb is to configure ACPY2 to use as few queues as it is necessary to meet real-time requirements. By preserving free queues on the system, adding DMA-based peripherals to the system becomes easier. For example, on the C64x, if ACPY2 uses two queues and the EMAC device driver uses one queue, then adding a video peripheral in the future would not affect the performance of the algorithms since its driver can use the remaining queue. This allows determinism in the timing and benchmark numbers for the system.

- Using the ACPY2_start() API is the simplest way to submit DMA transfers, as it handles all alignment issues behind the scenes. It is the recommended API for users who want to rapidly build a working prototype of the algorithm. However, for those who desire better performance, the JPEG algorithms could be re-evaluated to determine if they could benefit from more aligned transfers. Calling ACPY2_startAligned() in a tight loop instead of ACPY2_start() significantly reduces MIPS.

3 Conclusion

By adhering to the new XDAIS DMA specifications, one can improve the performance of algorithms. Integrating these algorithms into RF5 is a very straightforward procedure.

4 References

Application Notes

1. *Reference Framework for eXpressDSP Software: RF5, An Extensive, High-Density System* (SPRA795)
2. *Reference Frameworks for eXpressDSP Software: API Reference* (SPRA147)
3. *A Video, Audio, Networking System on the C64x NVDK Using eXpressDSP RF5* (SPRA844)
4. *DMA for eXpressDSP-Compliant Algorithm Producers and Consumers* (SPRA445)
5. *Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6x1x* (SPRA789)

User Guides

1. *TMS320 DSP/BIOS User's Guide* (SPRU423B)
2. *TMS320C6000 DSP/BIOS API Reference Guide* (SPRU403E)
3. *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401E)
4. *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352E)
5. *TMS320 DSP Algorithm Standard API Reference* (SPRU360C)
6. *TMS320C64x Two Level Internal Memory Reference Guide* (SPRU610)

Web Resources

1. www.dspvillage.com

IMPORTANT NOTICE

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

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

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

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

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

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

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265