TMS320C6000 Imaging Developer's Kit (IDK) Programmer's Guide

Literature Number: SPRU495A
September 2001

PRINTED WITH SOY INK™

Printed on Recycled Paper

Preface

Read This First

About This Manual

The IDK Programmer's Guide is intended for programmers and application developers who want to develop new applications to extend the capability of the IDK that are eXpressDSP-compliant. The programmer's guide is also intended to ease the learning curve associated with understanding how developers can leverage the different software tools that are provided along with the IDK. In addition the programmer's guide discusses several generic templates, in the form of application drivers that are used in most common image processing tasks, which can be re-used.

How to Use This Manual

This document contains the following chapters:

- ❑ **Chapter 1 – Introduction**, provides a basic overview of the Programmer's Guide.
- ❑ **Chapter 2 – Image Data Manager**, describes the image data manager utility.
- ❑ **Chapter 3 – Development of Application Drivers as Generic Templates for Image Processing**, describes the use of Image Data Manager routines to develop generic templates in the form of application level drivers.
- ❑ **Chapter 4 – Application Development and Prototyping Using Generic Templates**, desribes the use of generic templates to aid users in initial application development and prototyping.
- ❑ **Chapter 6 – Integration of an Application into the Imaging Framework**, describes integrating an application into the imaging framework.
- ❑ **Chapter 7 – Conclusions**, summarizes the concepts discussed in the programer's guide.

Related Documentation From Texas Instruments

The following references are provided for further information:

Documentation

TMS320C6000 Imaging Developer's Kit (IDK) User's Guide (Literature number SPRU494)

TMS320C6000 Imaging Developer's Kit (IDK) Video Device Driver User's Guide (Literature number SPRU499)

C6000 JPEG Information:

- ❑ *TMS320C6000 JPEG Implementation* Application Report (Literature number SPRA704)
- ❑ *Optimizing JPEG on the TMS320C6211 With 2-Level Cache* Application Report (Literature number SPRA705)

C6000 H.263 Information:

- ❑ *H.263 Decoder: TMS320C6000 Implementation* Application Report (Literature number SPRA703)
- ❑ *H.263 Encoder: TMS320C6000 Implementation* Application Report (Literature number SPRA721)

Text Conventions

The following typographical conventions are used in this specification:

- ❑ Text inside back-quotes (‘‘) represents pseudo-code
- ❑ Program source code, function and macro names, parameters, and command line commands are shown in a `mono-spaced font`.

Contents

Figures

Tables

Chapter 1

Introduction

The IDK Programmer’s Guide is intended for programmers and application developers who want to develop new applications to extend the capability of the IDK that are eXpressDSP-compliant. The programmer’s guide is also intended to ease the learning curve associated with understanding how developers can leverage the different software tools that are provided along with the IDK. In addition the programmer’s guide discusses several generic templates, in the form of application drivers that are used in most common image processing tasks, which can be re-used. The re-use of such application drivers to develop new image processing algorithms is demonstrated. The IDK stresses the concept of developing algorithms under an eXpressDSP-compliant framework. The creation of new algorithms that are eXpressDSP-compliant facilitates their integration into the framework under a Channel Manager that creates, runs and deletes multiple instances of different algorithms. In addition the framework allows users to dynamically vary the parameters of the algorithm at run-time. The examples developed for initial testing are integrated into the framework to demonstrate to users how they can plug their algorithms into the Channel Manager framework. The examples shown in the programmer’s guide are intended to represent the data flow found in typical image processing algorithms. While it is impossible to conceive of all the possible data flow situations required for image processing algorithms, a careful study of the examples covered in this guide by the user, should help users in the development of new applications.

Topic **Page**

1.1 IDK as a Rapid Prototyping Platform

In addition to showcasing the included demonstrations, the IDK also serves as a rapid prototyping platform for the development of image and video processing algorithms. Using the software and hardware components provided in the IDK, developers can quickly move from conceptualizing algorithms to high performance working implementations on TMS320C6000 DSP board, with live video input and output to evaluate their algorithms. This rapid prototyping ability is based on the following developments included in the IDK:

1.1.1 Rapid Prototyping Software Suite

The Rapid Prototyping Software Suite consists of the following software packages: ImageLIB, Chip Support Library (CSL), and Image Data Manager.

ImageLIB: This is an optimized Image/Video Processing Functions Library for C programmers on TMS320C6000 devices. It includes many C-callable, assembly-optimized, general-purpose image/video processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical. ImageLIB offers the following advantages to software developers:

❑ By using the routines provided in ImageLIB, an application can achieve execution speeds that are considerably faster than equivalent code written in standard ANSI C language.

❑ By providing ready-to-use DSP functions, ImageLIB can significantly shorten image/video processing application development time.

ImageLIB software and associated documentation is available at:

http://www.ti.com

Chip Support Library (CSL): CSL is a set of Application Programming Interfaces (APIs) used to configure and control all on-chip peripherals. It is intended to make software development easier in getting algorithms operational in a system. The goal of this library is ease of peripheral use, some level of compatibility between devices, shortened development time, code portability, some standardization, and hardware abstraction. CSL offers the following advantages to software developers:

❑ Enables development of DSP application code without having to physically program the registers of peripherals. This helps to make the programming task easier, quicker, and also there is less potential for mistakes.

❑ The availability of CSL for all C6000 devices allows an application to be developed once and run on any member of the TMS320C6000 DSP family.

- ❑ The ability to develop new libraries that use CSL as their foundation to allow for easy data transfers. An example of this is the Image Data Manager Utilities (described below) that use CSL to abstract the details of double buffering in DMAs.

CSL software and associated documentation is available at:

http://www.ti.com

Image Data Manager (IDM): Image Data Manager is a set of library routines that offer abstraction for double buffering of DMA requests, to efficiently move data in the background during processing. They have been developed to help remove the burden from the user of having to perform pointer updates and managing buffers in the code. IDM uses CSL calls to move data back and forth between external and internal memory during the course of processing. IDM utilities offer the following advantages to software developers:

- ❑ The ability to separate and compartmentalize data transfers from the algorithm, leading to software that is easy to understand and simple to maintain.
- ❑ The ability to provide data abstraction routines to perform efficient DMA transfer.

1.1.2 Rapid Prototyping Hardware

The IDK hardware consists of a C6711 DSK and a daughtercard that provides the following capabilities:

- ❑ Video Capture of NTSC/PAL signals (composite video).
- ❑ Display of RGB signals, 640x480 or 800x600 resolution, 16 bits per pixel (565 format).
- ❑ Video data formatting by an on-board FPGA to convert captured interleaved 4:2:2 data to separate Y, Cr, Cb components that may be sent to the DSP for processing.
- ❑ Video capture and display drivers software written using DSP/BIOS and CSL.

This enables users to quickly setup a development environment that includes video input and output capability.

1.2 IDK Documentation from a Programmer’s Perspective

The programmer’s guide is an application of several key software and hardware components that are discussed in complete detail in the IDK documentation. The programmer’s guide is not intended to replace these documents. Rather it is intended to be an application of all the software and hardware technologies discussed in the following documents. As part of discussing all the documents, the concepts that a programmer/developer is expected to gain are elaborated. Understanding these key concepts is essential to appreciating the ease of use that they bring to the development of several examples included as part of the programmer’s guide

The IDK consists of the following pieces of documentation:

❑ *TMS320C6000 IDK Users’ Guide*

  This guide details the software and hardware architecture of the Imaging Developer’s Kit. The software architecture includes at the highest level the IDK Framework (Channel Manager), followed by the Image Processing algorithms using ImageLIB or custom kernels developed by the user at the next level, followed by the use of Image Data Manager and CSL at the lowest level to provide the algorithm with the data it requires. It is important for users to understand the software architecture of the IDK. The hardware architecture for the IDK is also documented in this guide. It would be worthwhile for users to familiarize themselves with the hardware capabilities of the IDK.

  This guide also provides users with all the information regarding the capture and display drivers that were created to form the library “vcard.lib”. This library provides users abstraction from the underlying hardware. It lets users configure by setting/resetting the capture and display hardware to meet their requirements. The device drivers developed as part of the library are modified later to generate application drivers.

  While the video card library vcard.lib provides the abstraction of the hardware from the users, a study of the IDK hardware platform details can always be useful to understand the capabilities of the system, the underlying structure of the pixels, and how they are organized. The application drivers developed as examples later on in this document exploit the structure of the image on the hardware being organized as even and odd fields in separate regions in memory.

  Users are strongly urged to develop their new algorithms under an eXpressDSP-compliant framework, that allows for the seamless integration of multiple algorithms. This application report demonstrates how the Imaging Framework can be used to create, run, delete and modify several instances of algorithms. It demonstrates a powerful software paradigm that eXpressDSP-compliant algorithms can leverage. Programmers are strongly urged to familiarize themselves with the APIs required to interface with the Imaging Framework.

- *IDK Programmer's Guide*

  This document is intended to demonstrate the software and hardware components of the IDK to develop generic templates, in the form of examples that decrease the learning curve of the developer in extending the capabilities of the IDK.

- JPEG and H.263 Application Notes

  As part of the IDK demonstrations, a JPEG encoder and decoder demonstration and a multi channel H.263 decoder demonstration are included. The IDK also includes a H.263 loopback demonstration. These application notes detail all the optimization techniques and organization of the software in both these algorithms. The discussion of the techniques used is intended to aid developers in adopting similar techniques in developing image processing algorithms that conform to standards.

1.3 Overview of the Programmer’s Guide

This section provides the users with an overview of the programmer’s guide. The programmer’s guide is divided into the following chapters.

❑ Chapter 2: Image Data Manager

The Image Data Manager uses the DAT calls in CSL to provide users with an abstraction for double- buffered DMA’s to bring the data required by the algorithm in the background, without impacting the processing of the DSP. The Image Data Manager routines help to relieve the burden of having to maintain the double buffering information. It isolates all the code pertaining to the management of double buffering to one file. The use of Image Data Manager results in code that is easy to maintain and understand. It also helps to restrict any errors associated with double buffering to a single file, allowing for code to be easily debugged and maintained. This chapter discusses the APIs for all the different functions that from a part of the Image Data Manager.

❑ Chapter 3: Development of Application Drivers as Generic Templates for Image Processing

This chapter makes use of the Image Data Manager routines discussed in the earlier chapter to develop generic templates for image processing in the form of application drivers. The use of the Image Data Manager functions, to develop application drivers for grayscale and color image processing are developed in this chapter. The performance of such application drivers is also discussed in this chapter.

❑ Chapter 4: Application Development and Prototyping Using Generic Templates

This chapter makes use of the generic templates developed in the previous chapter, to perform initial development and prototyping of the application. This is demonstrated by leveraging the application drivers to develop a grayscale and a color demonstration. The examples that are developed in this chapter are not as complicated as JPEG or H.263, but are intended to demonstrate the use of the application drivers as generic templates to aid in developing applications.

❑ Chapter 5: Image Processing Using the TMS320C62x ImageLIB

This chapter explains how users can leverage the power of the TMS320C62x ImageLIB, a collection of highly optimized C callable image kernels, to accomplish imaging algorithms. These optimized kernels have a wrapper function that brings in the data required for the kernel by calls to the image data manager.

Trade-offs that the user can perform to improve the performance obtained for performing the algorithm on the entire image are also discussed. The image processing demo makes use of functions from ImageLIB and adapts a generic template application driver to assemble an initial version of an image processing demonstration.

❑ Chapter 6: Integration of an Application into the Imaging Framework

  This chapter explains how users can take a prototyped application and convert it to be eXpressDSP-compliant In addition it demonstrates how users may integrate this application into the Imaging Framework, so that the present application can co-exist with several other applications. In addition the integration of the application into the Imaging Framework allows applications to be created, executed, deleted and modified during run-time. This allows for multichannel implementations of the algorithm, or multiple algorithms to run together side by side. The integration of voice processing and image processing algorithms was used for putting together the demonstration scenarios used in the IDK.

❑ Chapter 7: Conclusions

  This chapter summarizes the key concepts and techniques that software developers can leverage to build new applications on the IDK. Development of applications under that are eXpressDSP-compliant is essential in order for applications to be robust and maintain inter-operability with applications developed by others. The ability to run the developed application in an Imaging Framework to either create multiple instances of the same algorithm or the ability to run multiple algorithms simultaneously are attractive attributes.

Chapter 2

Image Data Manager (IDM)

The Image Data Manager utility is used to provide an abstraction for double-buffered DMA routines that programmers can use to move the data required for the algorithm efficiently in the background. This is done using the DMA controller in the background, while the DSP can continue working away on the data fetched in the previous iteration. The Image Data Manager was built using the CSL DAT routines. CSL stands for Chip Specific Library which in turn was provided to give a standard re-usable set of software routines and provide abstraction from the device hardware. This allows users to develop their application once and port it easily to other C6x devices. The Image Data Manager provides a set of APIs that allow a user to associate a “data stream” with a region of external memory from which data is to be fetched into a region of internal memory, or vice-versa. The direction of the stream decides whether the data is brought into internal memory (get) or sent out from internal memory (put). The Image Data Manager starts to manage the data required for the given application providing the user the pointer to the currently ready buffer of data if the stream is initialized in the input direction, or a pointer to write to if the stream is initialized in the output direction, once stream initialization is completed. The APIs restrict all the DMA management code that is typically written along with the entire application to be localized to one file. This allows for the code to be easily maintained and debugged. In addition it lets the user focus on the details of the underlying algorithm as opposed to having to explicitly include double buffering in the code.

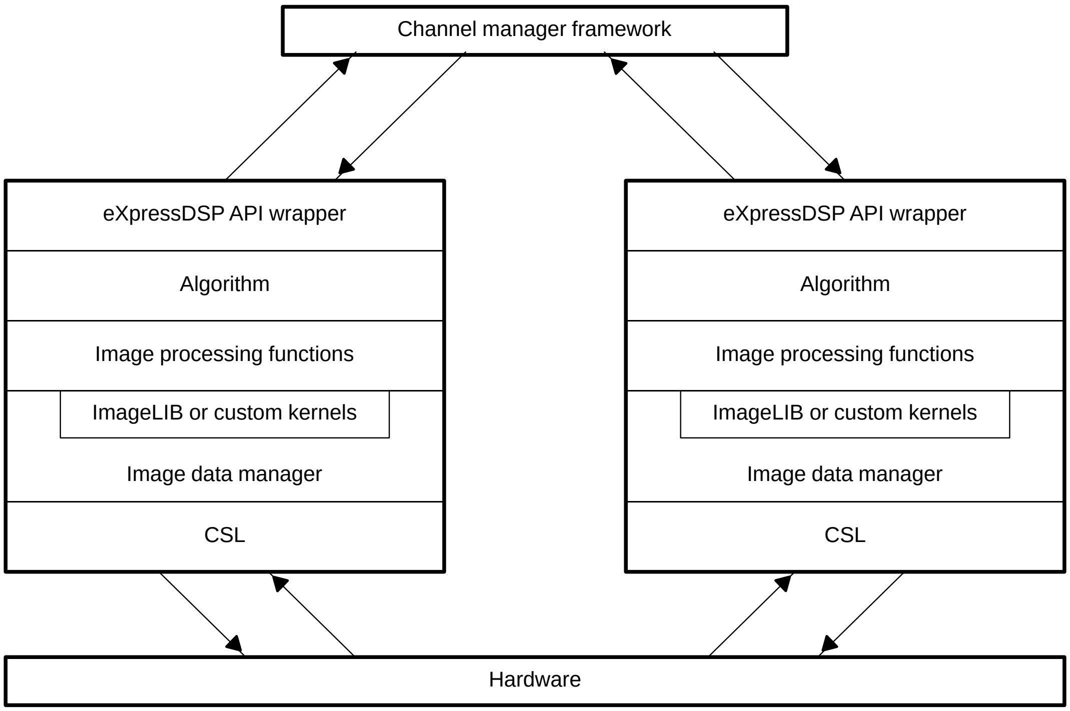
The Image Data Manager is used to develop the generic template routines included in the programmer’s guide in the form of application level drivers, that users can deploy or modify for developing new algorithms.

Topic **Page**

2.1 Software Architecture for the IDK

It is important to understand how Image Data Manager interacts with the various components that form the software architecture for the IDK. Figure 2–1 shows the hierarchy of components.

Figure 2–1. Software Architecture Hierarchy of Components


Channel manager framework
eXpressDSP API wrapper
Algorithm
Image processing functions
ImageLIB or custom kernels
Image data manager
CSL
eXpressDSP API wrapper
Algorithm
Image processing functions
ImageLIB or custom kernels
Image data manager
CSL
Hardware

The different layers of the software architecture and their interaction with the Channel Manager framework and hardware is shown. Figure 2–1 not only details all the different software layers, but also shows how several applications developed with this software architecture in mind eventually integrate into the Channel Manager framework. The software architecture is comprised of six different layers. The following are the levels of the software architecture from the top to the bottom:

❑ The top-most layer of this hierarchical architecture is the **eXpressDSP API Wrapper**. This is the interface available to other algorithms or users of the eXpressDSP-compliant algorithm.

❑ The next layer is the actual **Algorithm**. It typically invokes one or more Image Processing Functions. The ordering of the functions, and data passing between the functions is controlled by the standard algorithm.

- An **Image Processing Function** is a “wrapper” around one or more Imaging Kernels, and is responsible for managing data I/O for the kernels.
- **ImageLIB or Custom Kernels** are the core processing operations. Typically, they are DSP code that has been highly optimized for performance. Many of these kernels are contained in the TI ImageLIB software, while others are custom software for specific applications.
- **Image Data Manager (IDM)** is a set of library routines that offer abstraction for double buffering of DMA requests, to efficiently move data in the background during processing. They have been developed to help remove the burden from the user of having to perform pointer updates in the code. DSTREAM functions use CSL DAT calls to move data between external and internal memory during the course of processing.
- **Chip Support Library (CSL)** provides the users with a key set of routines that provide the user with abstraction from the target hardware. This allows users to specify the target hardware and re-use their software by leveraging the power of CSL. The fact that there is a particular flavor of CSL for every C6000 device, and in future every C5000 device, encourages programmers to use it for application development from a software re-usability perspective.

2.2 Conceptual Details of Image Data Manager (IDM) Implementation

It is important for users to understand how Image Data Manager automatically manages the buffers in internal memory, providing the users the current set of active buffers for the input and output image streams. This section deals with the conceptual aspect without getting into the details of the underlying APIs.

There are two different access modes required by most image processing algorithms:

❑ Double Buffering

❑ Sliding Window

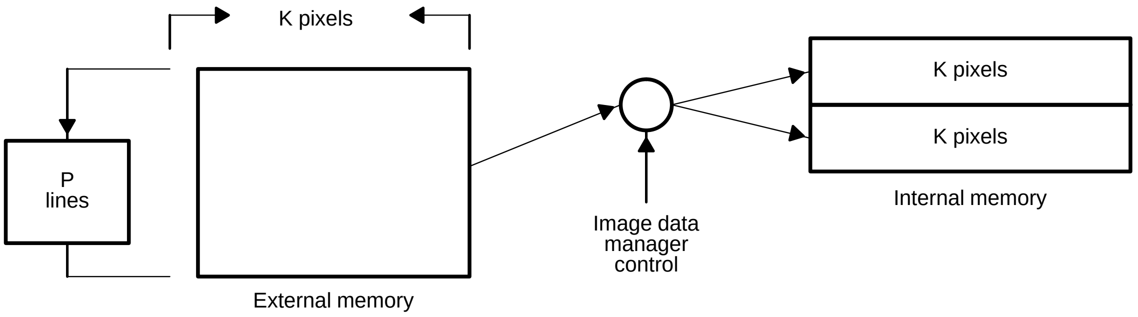
Every useful image processing application requires at least two image streams, one on the input side and one on the output side. This does not account for the class of image processing algorithms that perform in-place transforms like DCTs. Image Data Manager expects the inputs and outputs to be in a separate buffer and as such does not support in place transformations, unless the internal memory regions are aliased.

Image Data Manager however supports both the mechanisms of data transfer shown for input streams. The output stream supports the double buffering mode only. These data transfer mechanisms and how Image Data Manager supports them are specified in the sub-sections under this section. Programmers should take time to understand the details of the buffer requirements in different scenarios, so that they can set up image data streams for different conditions as their application may require.

2.2.1 Double Buffering

Image Data Manager provides for double buffering on input and output streams. Let us consider that a given image in external memory of size “PK” lines is to be processed “K” pixels at a time, where each line has a width of “K” pixels. It can be seen that to process the entire image would require P such iterations. Image Data Manager expects a contiguous region of internal memory of size 2K lines.

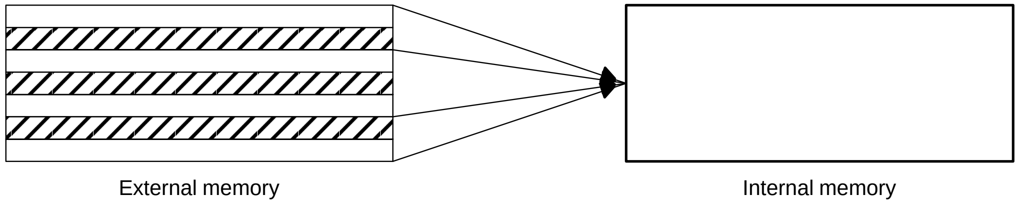
Figure 2–2. Image Data Manager Buffer Requirements for Simple Double Buffering



The buffering requirements that the Image Data Manager requires for this case are shown in Figure 2–2. The internal memory that needs to be allocated for the algorithm should be a contiguous region of size 2K pixels. The IDM controls by multiplexing between the two contiguous regions of K pixels, on a ping-pong basis. In the case shown K pixels form one line of the output image, however it is conceivable that the user may want to work on multiple output lines at a given time. In addition these lines may be contiguous in external memory, or may be spaced apart by a fixed offset. The latter is the case if the lines of the even field are processed first followed by the lines of the odd field. In addition there may be cases where each input line that is fetched from external memory is separated by a fixed offset. Although the data to be fetched from external memory may be sperated by a fixed offset, Image Data Manager always brings data into internal memory so that all lines in the internal memory are contiguous.

Whenever multiple lines are to be fetched on every iteration, with an arbitrary but fixed offset between the individual lines, Image Data Manager makes use of the DAT_2D routines and hence performs a 2D transfer. If the multiple lines are contiguous in external memory, then both the DAT 1D call and DAT_2D call from CSL can be used. In this situation it is preferable to use the 1D DAT call from a speed perspective as it is likely to complete sooner. The situation where each of the lines to be fetched is separated by an offset is shown in Figure 2–3. In any case the internal memory needs to be twice as large as the amount transferred on every iteration.

Figure 2–3. Transfer of Multiple Non-Contiguous Lines in External Memory



The Image Data Manager (IDM) returns to the user the current ready region that can be used for processing. The double buffering mechanism is often used on the output side, to commit back results to external memory. Most common image processing tasks, require context in the form of not only the present line being processed but also the surrounding lines above and below the current line. This leads to the sliding window mechanism discussed in the next section.

2.2.2 Sliding Window Mechanism

The sliding window mechanism is best illustrated by a 3x3 convolution algorithm. The following table shows the set of input lines (in_line_x) that are required to produce one output line (out_line_y). The first few iterations of this algorithm are traced to represent the data flow required by this algorithm.

Table 2–1. Data Flow in a 3x3 Convolution Algorithm

Iteration Number	Input Lines Required	Output Line Produced
1	in_line_ 0, in_line_1, in_line_2	out_line_o
2	in_line_1, in_line_2, in_line_3	out_line_1
3	in_line_2, in_line_3, in_line_4	out_line_2
4	in_line_3, in_line_4, in_line_5	out_line_3

The data flow shown in Table 2–1, is a sliding window mechanism, in which the oldest line is dropped. It can be seen that every iteration of the algorithm requires three lines. The three lines are composed of two lines from the previous iteration and 1 new line that has been fetched for the current iteration.

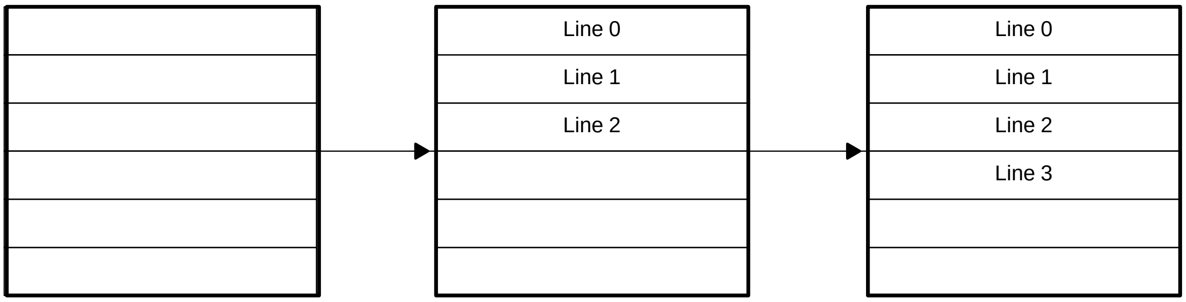
Sliding window mechanism of bringing in data is supported by IDM for input streams only. The output stream supports only the double buffered mode. The pointer to the current buffer returned by IDM always points to a contiguous region of memory. This allows all the pointer information required for algorithms to be computed using the current pointer curr_ptr and the width of the line x_dim.

Table 2–2. Relation of Pointers to Lines in Sliding Window

Pointer to First Line	Pointer to Second Line	Pointer to Third Line
curr_ptr	curr_ptr + x_dim	curr_ptr + (2 *x_dim)

The fact that the curr_ptr points to a contiguous region of internal memory, is beneficial to the user in that an array of pointers need not be passed. In addition the explicit relationship of the line number to the pointer for that line, is also beneficial to optimizing compilers as it provides them the freedom to re-order the load instructions. The internal memory is expected to be twice as large as the quantum of data being fetched, in this case it needs to hold up to six lines of the input image. The sequence in which this six line buffer is filled by the IDM is illustrated by Figure 2–4.

Figure 2–4. Progression of Events in Internal Memory For Sliding Window Mechanism



The progression of events is continued in the next figure. Image Data Manager issues copies to start copying lines in internal memory once more than half the internal memory is filled by IDM. This situation allows the data that is being fetched to wrap back to the appropriate location in internal memory and fetch the next line.

Figure 2–5. Progression of Events in Internal Memory for Sliding Window

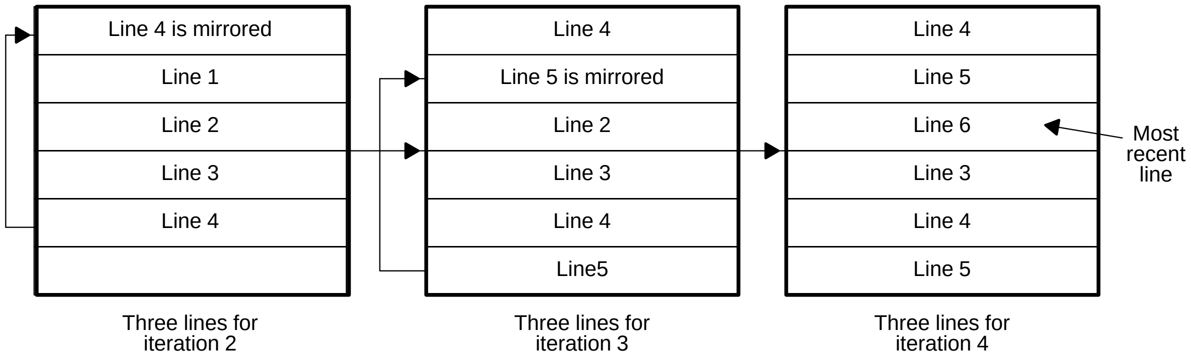


Figure 2–5, shows further progression of events, with respect to how copy transfers are issued, to mirror lines as soon as a line past half the internal memory is used. In addition the wrap around point is changed to be one line above the half of the internal memory. Thus it is only on the first iteration that three new lines are fetched and written into the start of the internal memory. For every iteration after this, one new line is fetched and copied into the appropriate location. For every user query of the IDM “curr_ptr” a pointer to the current contiguous section of three lines is returned, so that users can begin processing their algorithm.

The API details that users can call to take advantage of the double buffered and sliding window mechanisms are discussed in the next section. It is imperative that users understand the internal memory buffering requirements and the different situations that IDM offers support for.

2.2.3 Data Transfers that Image Data Manager Does Not Support

The Image Data Manager does not support transfers of multiple lines with variable widths. It does not support transfer of multiple lines with fixed width, that have variable offsets between them. However these cases are not common for image processing tasks IDM does not allow multiple lines of the image brought into internal memory to be written with either a fixed or variable offset. The multiple lines of the image are written into internal memory to be located contiguously. This kind of data transfer may be required occasionally to take care of boundary or edge conditions. However this assumes a specific boundary handling condition namely one of reflection across the boundaries and hence is not provided for in the IDM routines as other boundary handling conditions are possible. IDM supports only the double buffered mode on the output streams.

2.3 Image Data Manager API Documentation

Image Data Manager (IDM) is a set of library routines that offer abstraction for double buffering DMA requests, to efficiently move data in the background during processing. They have been developed to help remove the burden from the user of having to perform pointer updates in the code. IDM functions use DAT Calls from CSL to move data between external and internal memory. They can be extended in future to use EDMA/DMA calls as appropriate based on the device. All the functions detailed below assume that a DMA channel has been opened to support either 1D or 2D transfers using a DAT_open call. In addition each call to the IDM functions makes sure that the pointer handed off to the user points to a region in which data transfer has been completed, by waiting on the DMAs. Since the DMAs are done in a double-buffered fashion the wait call does not actually block the CPU, but merely verifies that the data transfer has indeed been completed. This is done internally by the IDM through the use of DAT_wait calls that make sure that the data transfer issued by a DAT_copy completes.

The following IDM functions are currently defined:

- **dstr_open**: Open an input/output image data stream to bring data from external to internal memory or viceversa.

- **dstr_get**: Bring data from external to intenal memory allowing for either one line at a time or multiple lines at a time without any offset between them. This function should only bre used on ainput stream. The behaviour of this function when used on an output stream cannot be guaranteed.

- **dstr_get_2d**: Bring data from external to internal memory allowing for etither one line at a time, or multiple lines at a time, with no/fixed offset between the lines. This function should only be used on an input stream. The behaviour of this function when used on an output stream cannot be guaranteed.

- **dstr_put**: Commit data from internal memory to external memory either one line at a time, or multiple lines without any offset between them. This function should only be used on an output stream. The behaviour of this function when used on an input stream cannot be guaranteed.

- **dstr_put_2d**: Commit data from internal memory to external memory either one line at a time, or multiple lines with no/fixed offset between successive lines. This function should only be used on an output stream. The behaviour of this function when used on an output stream cannot be guaranteed.

- **dstr_rewind**: This function performs a stream rewind, by resetting the pointer to the external memory to the new location. The number of iterations that have been executed is not reset. Hence when the stream is initialized, the size of the external memory should be the sum of all the regions in external memory from which data will be feteched.

- **dstr_close**: This function closes the streams opened using dstr_open. This function waits for any previous DMAs to complete and then closes the stream. This function should only be called on a stream that has already been opened.

dstr_open — *Initializes input/output stream*

Prototype

```
int dstr_open
{
     dstr_t *dstr,
     void    *x_data,
     int        x_size,
     void    *i_data,
     unsigned short i_size,
     unsigned short quantum,
     unsigned  short  multiple,
     unsigned  short  stride,
     unsigned  short  w_size,
     dstr_t_dir_t         dir
};
```

Arguments

dstr_t *dstr	DMA Stream Structure
void *x_data	“External” data buffer
int x_size	Size of external data buffer
void *i_data	“Internal” data buffer
unsigned short i_size	Size of internal data buffer
unsigned short quantum	Size of single transfer get/put
unsigned short multiple	Number of lines
unsigned short stride	Stride amount for external pointer
unsigned short w_size	Window size, 1 for double buffering
dstr_t dir	Direction Input/Output

Return Value

Int	0 – function succeeded
	{–1,–2,–3}– function failed

Description Initializes input/output stream. Must be used before dstr_put/dstr_get or dstr_put_2d/dstr_get_2d calls are used. dstr_close should be used only on a stream that has been opened using dstr_open.

dstr_get — *Returns pointer to current area in internal memory*

Prototype (void *) dstr_get();

Arguments none

Return Value (void *) Returns a pointer to current input buffer.

Description Returns a pointer to the current area in internal memory that contains valid data.

dstr_get_2d

Returns pointer to current area in internal memory

Prototype (void *) dstr_get_2d();

Arguments none

Return Value (void *) Returns a pointer to current input buffer.

Description Returns a pointer to the current area in internal memory that contains valid data. This function is called on an input stream, when succesive lines in external memory are seperated by a fixed offset.

dstr_put

Returns pointer to current buffer

Prototype (void *) dstr_put();

Arguments none

Return Value (void *) Pointer to current buffer in which output results can be stored.

Description Returns a pointer to current buffer in which output results can be stored. It also commits the results of the previous output buffer to external memory.

dstr_put_2d

Returns pointer to current buffer

Prototype (void *) dstr_put_2d();

Arguments none

Return Value (void *) Pointer to current buffer in which output results can be stored.

Description Returns a pointer to current buffer in which output results can be stored. It also commits the results of the previous output buffer to external memory. This function should be used when the output lines need to be written to external memory either with zero/fixed offset between successive lines.

dstr_rewind — *Rewinds input/output streams*

Prototype

```
int dstr_rewind
(
    dstr_t          *dstr,
    void            *x_data,
    dstr_dir_t       dir,
    unsigned short  w_size
)
```

Arguments

dstr_t *dstr	DMA stream structure
void *x_data	Pointer to “external buffer” to which stream is reset.
dstr_dir_t dir,	Direction of stream, input/output
unsigned short w_size	Window size 1, for double buffering

Return Value

int	0 for succesful rewind

Description

Rewinds input/output streams to start fetching data from new location in external memory. The external offset is reset to 0. This resets the number of external transfers completed to 0.

dstr_close — *Closes stream*

Prototype

```
void dstr_close(dstr_t *dstr);
```

Arguments

dstr_t *dstr	Pointer to DMA stream structure

Return Value

void	none

Description

This function closes the stream that was opened using dstr_open

Direction Structure Definitions — *Defines directions input/output*

Prototype

```
typedef enum dstr_dir_t
{
    DSTR_INPUT,
    DSTR_OUTPUT
} dstr_dir_t;
```

Arguments

none

Return Value none

Description Structure that defines directions input/output. User can use the above defined symbolic names to set direction of image stream.

DMA Stream Definition

Maintains state information

Prototype

```
typedef struct dstr_t
{
    char              *x_data;
    int                x_ofs;
    unsigned           x_size;
    char              *i_data;
    unsigned short     i_ofs;
    unsigned short     i_size;
    unsigned short     w_size;
    unsigned short     quantum;
    unsigned short     multiple;
    unsigned short     stride;
    unsigned           xfer_id;
} dstr_t;
```

Arguments

char *x_data	Pointer to external data
int x_ofs	Current offset to external data
unsigned x_size	Length of external data buffer
char *i_data	Pointer to internal buffer
unsigned short i_ofs	Offset to internal buffer
unsigned short i_size	Size of internal buffer
unsigned short w_size	Size of window
unsigned short quantum	Amount transferred by a single get/put call
unsigned short stride	Byte offset between succesive lines in external memory that need to be fetched.
unsigned xfer_id	Transfer id of the previous DMA

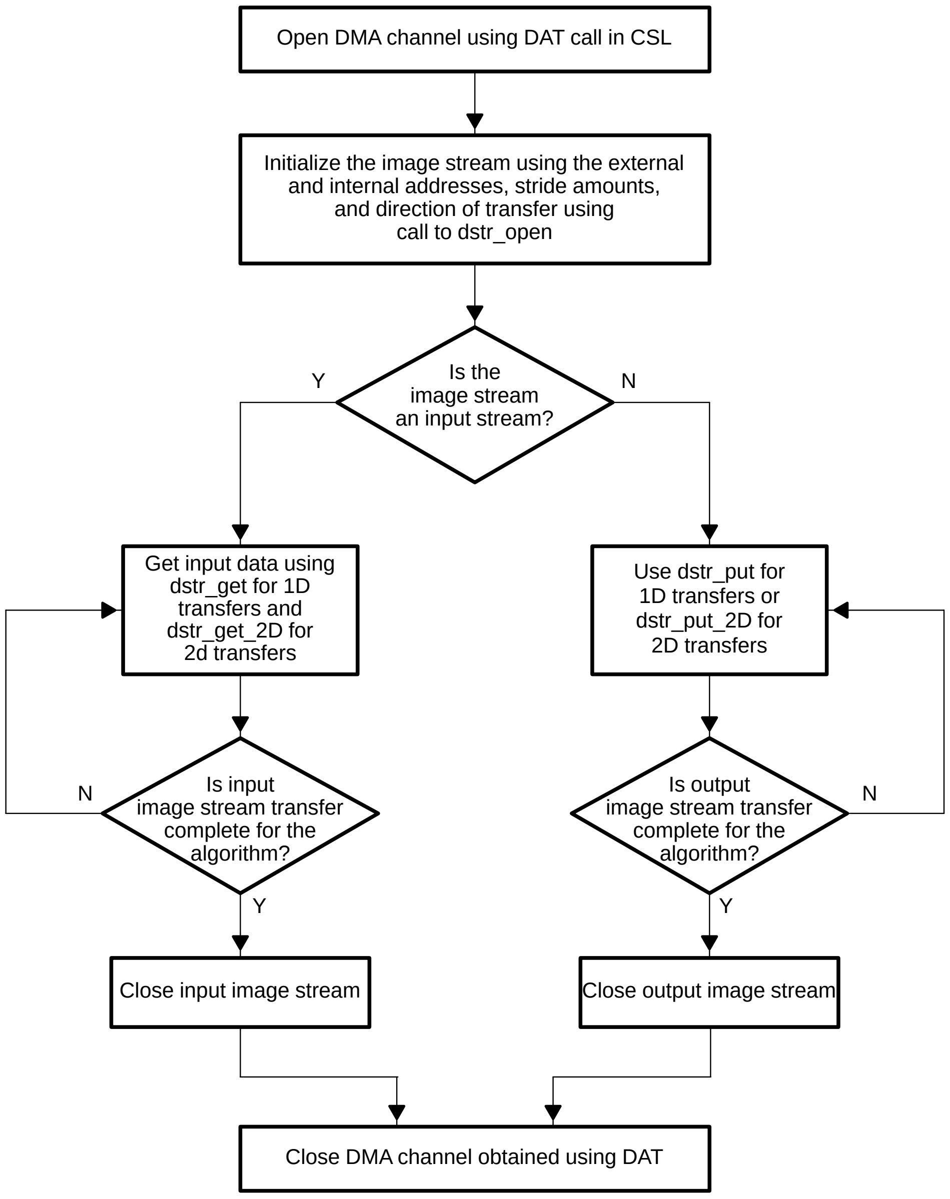
Return Value none

Description Internal structure that IDM uses to maintain state information. User declares input and output streams of type dstr-t for using IDM.

2.4 Programming Model for the Image Data Manager

The typical steps involved in utilizing an image stream from the Image Data Manager are summarized in Figure 2–6. These steps need to be followed for creating new image streams and for utilizing them in bringing the required data by the algorithm.

Figure 2–6. Steps Involved in Creating and Using Image Streams


Open DMA channel using DAT call in CSL
Initialize the image stream using the external and internal addresses, stride amounts, and direction of transfer using call to dstr_open
Is the image stream an input stream?
Y
N
Get input data using dstr_get for 1D transfers and dstr_get_2D for 2d transfers
Use dstr_put for 1D transfers or dstr_put_2D for 2D transfers
Is input image stream transfer complete for the algorithm?
N
Y
Is output image stream transfer complete for the algorithm?
N
Y
Close input image stream
Close output image stream
Close DMA channel obtained using DAT

2.5 Conclusions

This chapter introduced the Image Data Manager, a set of utilities used in seamlessly bringing the data required for the algorithm. Image Data Manager uses the DAT module from CSL to issue the calls required for 1D and 2D transfer of data. The conceptual implementation of the Image Data Manager and how it provides the abstraction of double buffering and sliding window were discussed. In addition the benefits of software re-usability and maintainability of the resulting code were also discussed. The APIs that form the IDM and their invocation sequence along with their capabilities were also discussed in this chapter.

Concepts discussed in this chapter will be used in the next chapter to develop generic templates, in the form of application level drivers that users can leverage for application development. The Image Data Manager and its associated functions are used in the development of each of these application drivers, and hence programmers will find it useful to familiarize themselves with the capabilities provided by IDM.

Chapter 3

Development of Application Drivers as Generic Templates for Image Processing

This chapter makes use of the Image Data Manager routines discussed in the previous chapter to develop generic templates in the form of application level drivers. The use of these application drivers to develop new applications is discussed in Chapter 4. While it is impossible to provide generic templates that cover all possible situations, the templates provided here are meant to be reasonably representative of common image processing algorithms. This chapter discusses the development of gray scale and color application drivers. Different approaches to generating these application drivers are discussed and their resulting change in performance are also summarized. A brief overview of the hardware is provided so that programmers can gain a better understanding of the hardware they are programming for.

Topic **Page**

3.1 Imaging TDK Hardware

The Imaging TDK hardware consists of a C6711 DSK with 16MB SDRAM, and a daughtercard that provides video capture, display, and formatting capabilities. The daughtercard (Figure 3–1) includes:

- ❑ NTSC/PAL digital video decoder IC (TI TVP5022)
- ❑ NTSC/PAL digital video encoder IC (TI TVP6000)
- ❑ Video Palette IC (TI TVP3026C)
- ❑ Xilinx FPGA that includes the following functions: card controller, FIFO buffer manager, front/back end interfaces. Details of the interfaces served by the FPGA are provided in Appendix I
- ❑ 16Mbit SDRAM (capture frame memory), with option to support 64Mbit devices

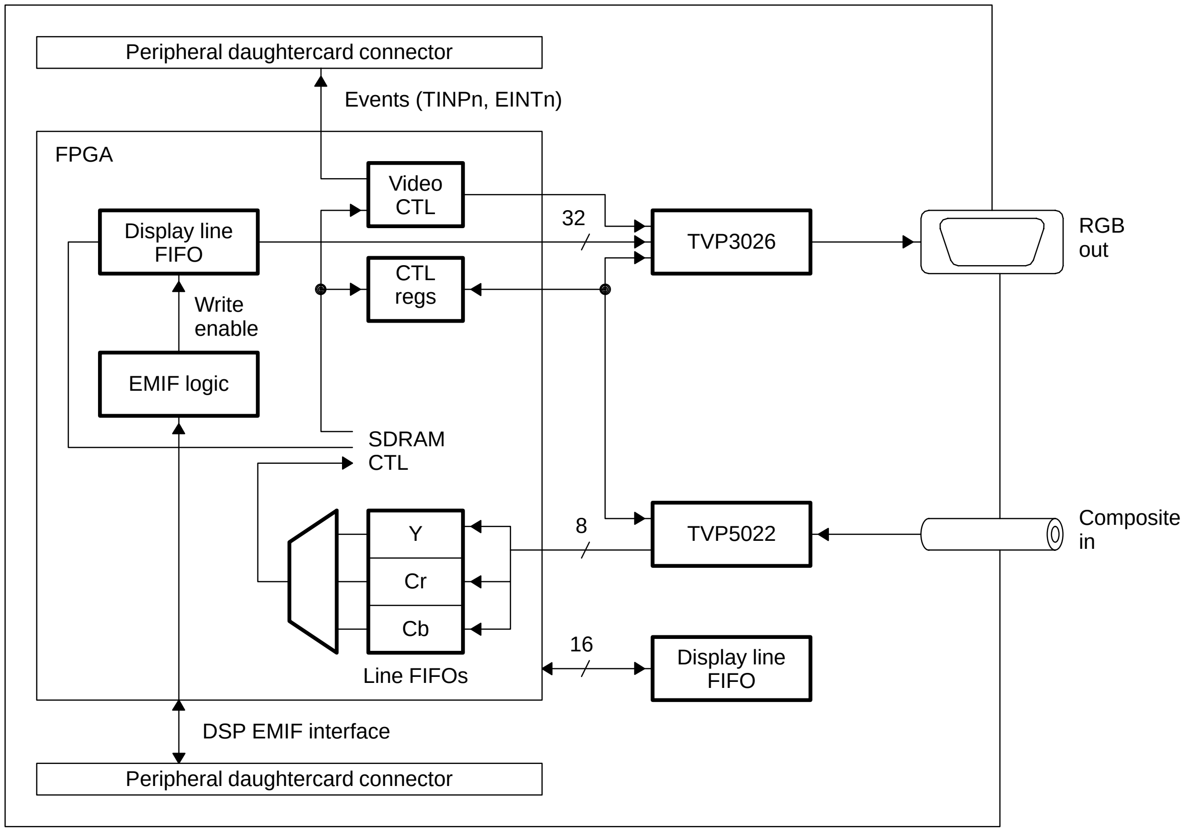
The daughtercard provides the ability for the following types of video capture and display:

- ❑ Input video signal capture is limited to a single NTSC/PAL signal
- ❑ Input signal should of composite video format
- ❑ Display output may be in the form of a 16-bit RGB (565) signal

The daughtercard hardware includes the following:

- ❑ 1 set of TMS320C6000 daughtercard connectors (male, solder side)
- ❑ Female RCA connector for composite video input (NTSC/PAL)
- ❑ Female S-Video connector for component (Y-C) video input (NTSC/PAL)
- ❑ Female RCA connector for composite video output (NTSC/PAL)
- ❑ Female S-Video connector for component video output (NTSC/PAL)
- ❑ Female 15-pin VGA connector for RGB monitor output

Figure 3–1. Imaging TDK Daughtercard Block Diagram



3.2 Video Capture

The Imaging TDK daughtercard includes two video input ports for NTSC/PAL video, of which one may be active at any time. The NTSC/PAL inputs consist of an industry standard RCA jack for composite video input, and an S-video jack for component (Y-C) input. Both inputs are routed to the TVP5022 video decoder, and may be configured for square-pixel or ITU standard resolutions. The TVP5022 performs digitization and minimal filtering of the video inputs. All video input data is digitized in the 4:2:2 format, to produce a standard YCrCb pixel stream. Since most DSP algorithms operate on input data as separate Y, Cr, and Cb blocks, the FPGA interface performs separation of the digital stream before writing it to the capture frame buffer. Captured data is stored as two separate fields, in three separate blocks in the frame buffer.

Data is expected from the TVP5022 in the Cr0-Y0, Cb0-Y1, Cr2-Y2, Cb2-Y3, … format. The FPGA internally adjusts the data stream for endian, and stores it into the capture frame memory as shown in Figure 2–2. The FPGA manages a capture frame buffer in an on-board SDRAM memory bank. SDRAM was chosen due to its low cost for the required memory bank size (2MB), however, the DSP interface to this buffer is of the ASRAM type. The FPGA performs this translation autonomously. It is noted that the capture frame memory is read only to the DSP interface. Any writes attempted to the frame memory by the DSP are discarded.

The FPGA SDRAM controller supports both 2MB and 8MB configurations of SDRAM. The 8MB option can be used in the event that 2MB devices become scarce and also in the cases where additional capture memory is required. For example, in the case of ITU-sampled PAL video, 625 lines of 768 pixels must be captured, which exceeds the 2MB capacity of the capture buffer in that configuration. The FPGA supports both memory types and is controllable via software. Table 3–1 outlines the capture formats vs memory requirements.

Table 3–1. Video Capture Memory Requirements

Format	Required Memory
NTSC, square pixel	2MB
PAL, square pixel	8MB
NTSC, ITU601	2MB
PAL, ITU601	8MB

Note:

The TVP5022 chipset and FPGA support sampling of all versions of the PAL standard, though stuffing options of the TVP5022 crystal may be required.

Read accesses to the frame memory are throttled as appropriate using the DSP EMIF ARDY signal. Since the SDRAM memory is faster than the ASRAM interface, this is generally only necessary at the beginning of a burst of reads, and possibly when refreshes of the SDRAM bank are required. The FPGA includes a small read FIFO to minimize the effect of this. It should be noted however that the frame memory management is most efficient when accessed linearly. It is suggested that the application software access the memory in a linear fashion, to minimize SDRAM page misses which will slow the memory transactions. The ARDY signal is also asserted when bank conflicts occur, resulting from arbitration effects with the capture line FIFOs. The effect is minimized by the existence of the FIFOs, plus a priority scheme implemented in the FPGA controller.

All video input timing is provided by the TVP5022. This includes a vertical synchronization pulse, plus a composite blanking signal which indicates the presence of active data on the pixel bus. A pixel clock is also provided, which is used by the FPGA to latch data into the aforementioned line FIFOs. Data is routed to the FPGA over an 8-bit video input port. Data may be captured in either the square pixel (640x480 or 768x576) or ITU (720x480 or 720x576) format. The format is determined via a control register bit in the TVP5022, which must be programmed to denote line length divisibility by 64 or 72 (all formats fit into one of these two categories). The setting of the input mode, as well as complete configuration of the TVP5022, is provided via an I^2C interface. A complete list of the addressable registers and their functions in the TVP5022 is available at:

http://www.ti.com

Captured data is stored as two separate fields (odd and even fields), in three separate blocks (Y, Cr, Cb) in the frame buffer memory on the daughtercard. Note that the memory locations of the fields, as well as the blocks within the fields, are not necessarily contiguous. Up to 3 frames of captured data may be stored in the daughtercard memory. At any given time, the FPGA controls two of the buffers to which it writes captured video data in a ping-pong fashion. The application has access to the third buffer, which typically has the most recently captured data. If the application falls behind in processing, the two buffers that the FPGA controls can be toggled and the application simply runs at a processing rate less than the captured 30 frames/sec. If the application can maintain the full processing rate, the buffers are physically walked through by both the FPGA and the application in a circular fashion. See Figure 2–3 for an explanation of the capture buffer management.

The FPGA directly controls all the capture data management, without any DSP resource (specifically, a DMA channel). The FPGA provides a capture frame interrupt to the DSP, which is used to inform the driver that a new frame is available for processing. The capture event may be mapped to one of the DSP events as shown in Table 3–2.

Table 3–2. Capture Events

DSP Event	**Mapped to System Event** …	**Intended Use** …
$\overline{\text{EINTn}}$ (n = 4–7)	Vertical sync falling (end of captured frame)	Interrupt to CPU driver

Any DSP event line not tied to a capture (or display) event is tri-stated, such that it may be used by another daughtercard or motherboard interface.

To maintain this buffer scheme, it is necessary for the IDK driver software to inform the FPGA when the application has completed use of its buffer, and that it may be returned to the pool of capture buffers which the FPGA owns.

3.3 Video Display

The Imaging TDK daughtercard includes three video output ports, of which one may be active at any time. Two of the ports output NTSC video, while the third provides RGB output for a standard computer monitor. The NTSC outputs consist of an industry standard RCA jack for composite video output, and an S-video jack for component (Y-C) output. Both outputs are driven by the TVP6000 video encoder, and may be configured for square-pixel or the ITU standard resolutions. Both outputs operated in the standard TV manner of interlaced frames. The RGB output is driven by the TVP3026, and can drive any of the standard monitor resolutions, in a non-interlaced mode.

In the case of RGB output the FPGA provides the video timing to the output. Consequently, the DSP display driver software must also program the FPGA integrated video controller, which drives the timing information to the TVP3026 RGB palette.

Video data is built up in buffers in system memory on the C6711 DSK. Frame buffer memory is of the SDRAM type, with a read CAS latency of three. The imaging daughtercard does not include any addressable amount of video display memory. Video output data is transferred in real time from the frame buffer to the imaging daughtercard. This data service can be provided by the DSP EDMA controller and EMIF resources. Figure 3–1 shows a transfer of data from the DSK frame buffer to the imaging daughtercard and display hardware.

The FPGA monitors the display device and generates events to the DSP motherboard. The events supported by the FPGA for display are shown in Table 3–3.

Table 3–3. Display Events

Event/Signal	May be Mapped to Daughtercard Signal …	Intended Use …
Pixel clock (active pixels only)	TOUT0 or TOUT1	Timer period set to pixels per line, TINT drives DMA line event
Composite blank falling (end of active line)	$\overline{\text{EINT7}}$, $\overline{\text{EINT6}}$, $\overline{\text{EINT5}}$, $\overline{\text{EINT4}}$	$\overline{\text{EINTn}}$ drives DMA line event; $\overline{\text{EINTn}}$ drives CPU interrupt
Vertical sync falling (end of frame)	$\overline{\text{EINT7}}$, $\overline{\text{EINT6}}$, $\overline{\text{EINT5}}$, $\overline{\text{EINT4}}$	$\overline{\text{EINTn}}$ drives DMA frame event; $\overline{\text{EINTn}}$ drives CPU interrupt

The preferred use of the above events is that the pixel clock be routed to one of the timer inputs, and a single interrupt is used on the vertical synchronization pulse to synchronize the DSP to the display. In this configuration, the selected timer must be configured in pulse mode with a period equal to the number of active pixels per line.

The FPGA is capable of driving to all DSP event lines, which include the four processor edge-triggered interrupts (/EINTn, n = 4–7) and the two timer inputs (TINPn, n= 0 or 1). Any DSP event line not selected for one of the above event sources is tri-stated by the FPGA, allowing it to be used by another daughter-card or motherboard interface.

Based on the above event selection, the IDK Display Driver configures the DSP DMA (or EDMA) and timer module (if appropriate) to service display events. The intended operation is that one DMA channel will be dedicated to servicing line events (once per horizontal sync pulse), and a separate DMA or CPU event per vertical sync pulse will be used for synchronization. The horizontal event forces the DMA to transfer a line of data to the FPGA display FIFO, via the aforementioned read of the motherboard SDRAM. The FPGA latches this data into the FIFO autonomously, which feeds the output display devices in real time.

Display events are scheduled such that data is ready for the display devices before it is needed. Specifically, this is achieved by scheduling the first event at the end of the vertical synchronization period. At this point, several lines of blanked display (for which no data is needed) must still be timed, so the DMA has time to perform the required accesses. In the case of an interrupt being used for the horizontal line events, generation of this event is straightforward. In the case of a timer however, generation is slightly more complicated, because the FPGA does not always source the horizontal video timing. In this case, special hardware inside the FPGA inserts additional TINPn pulses to 'fake' a first line of video display, to force a DMA of data to the FPGA line FIFO. The following diagram outlines the operation in both cases.

Since the FPGA is always one line ahead of the display, the last line event reads data that is off the end of the display buffer. This does not have any adverse effects, as the line FIFO is automatically reset during the vertical synchronization period. The data read is discarded, and the first line event generation described above re-synchronizes the display properly.

3.4 Application Drivers for Gray Scale Processing

Display drivers supporting these video display modes are included in the Imaging TDK. The drivers are written using DSP/BIOSII and CSL. Refer to the *TMS320C6000 IDK User's Guide* (Literature number SPRU494) for additional details. The application drivers developed in this section differ from the display drivers in that they bring the image data into internal memory and allow the user to focus on algorithm development. The application driver performs the merging of the even and odd fields in external memory, during the course of the algorithm. The device driver on the other hand uses the DMA/EDMA model to perform merging of the image data, while the DSP waits for the DMA transfers to complete. The merging combines the data stored in two separate fields in the memory area allocated for the image display. Both the device drivers and application drivers make use of the video capture library and video display library routines available in "vcard.lib". Hence the mechanism for getting the pointers to these external memory regions is the same, how the merging is performed in the two drivers is different.

Captured data on the daughtercard is stored as two separate fields (odd and even fields), in three separate blocks (Y, Cr, Cb) in the frame buffer memory on the daughtercard. This necessitates the creation of two application drivers one that works on the image in progressive order namely a line from the even field followed by a line from the odd field, and another that works in field order namely the even field followed by the odd field. Image processing applications in general can work on images either in progressive order or in fields order. However certain image processing applications require the data to be in progressive order. Hence both types of application drivers are provided for the display of a 640 by 480 image.

3.4.1 Gray Scale Driver

In all application drivers developed in this programmer's guide, the driver is labeled with an "odd_even" nomenclature if it works on the even field followed by the odd field. All other application drivers work on the image in normal or progressive order namely one line from the even field followed by one line from the odd field. It may be useful for programmers to review the steps involved in opening, utilizing and closing an image stream in IDM that was discussed in section 2.4.

The application driver that works on images in normal or progressive order requires two input image streams, one for the lines of the even field and one for the lines of the odd field. The successive lines within any field, are contiguous in memory although the two fields themselves are not. The application driver also requires an output stream to write the lines to the display area one after the other. Since the lines of the even and odd field are being merged, there is no offset between the successive output lines. This application driver can be used by programmers to put together applications that have the following data flow model.

Figure 3–2. Data Flow For Gray Scale Driver with Progressive Order

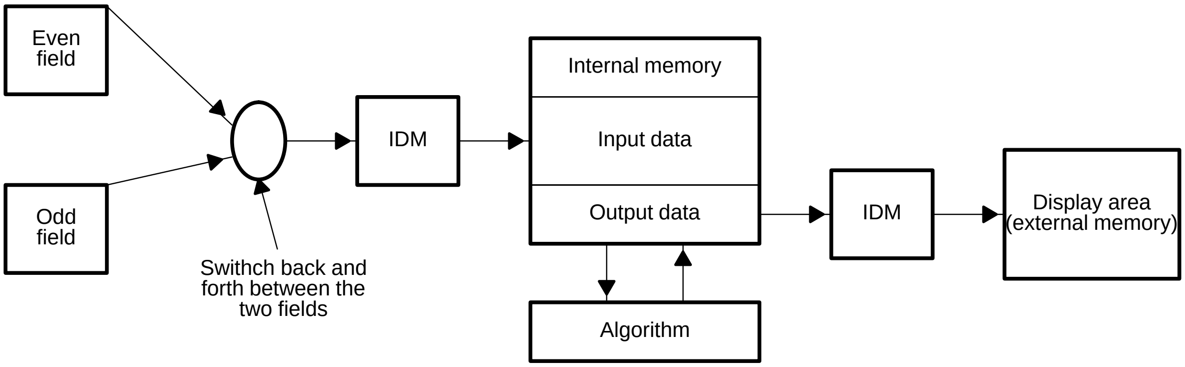

Even field
Odd field
Swithch back and forth between the two fields
IDM
Internal memory
Input data
Output data
Algorithm
IDM
Display area (external memory)

Figure 3–2, shows the interaction of IDM in bringing data to the application by switching between the even and odd fields back and forth to allow users to work on the image in progressive order, although the data on the Imaging TDK daughtercard is organized as two separate fields. The code to initialize the input image stream for the even field is shown below:

Step 1: Open input image stream for even field

```
 /*-------------------------------------------------------------*/
/* Initizlize input stream to start fetching from even field,   */
/* of size rows * cols, into int_mem1 (internal memory) of      */
/* size 2 * cols, 1 line of ”cols” pixels every time.          */
/* Check returned error code if any                             */
/*-------------------------------------------------------------*/
err_code  = dstr_open ( &iev_dstr,
                        in_image_ev->img_data,
                        rows * cols,
                        int_mem1,
                        2 * cols,
                        cols,
                        1,
                        cols,
                        1,
                        DSTR_INPUT);
```

Step 2: Open another input stream for the odd field

```
/*-------------------------------------------------------------*/
/* Initizlize input stream to start fetching from odd field,    */
/* of size rows * cols, into int_mem1 (internal memory) of      */
/* size 2 * cols, 1 line of ”cols” pixels every time.          */
/* Check returned error code if any                             */
/*-------------------------------------------------------------*/
```

```
err_code  = dstr_open (  &iod_dstr,
                         in_image_od–>img_data,
                         rows * cols,
                         int_mem2,
                         2 * cols,
                         cols,
                         1,
                         cols,
                         1,
                         DSTR_INPUT);
```

Step 3: Open output image stream for writing data to display buffer

```
 /*-------------------------------------------------------------*/
 /* Initialize output stream to write to out_image–>img_data,   */
 /* the merged fields back and the results of the algorithm.   */
 /* into int_mem3 (internal memory), 1 line at a time.         */
 /* Check for any error codes if any                            */
 /*-------------------------------------------------------------*/

err_code = dstr_open  ( &o_dstr,
                        out_image–>img_data,
                        out_rows * out_cols,
                        int_mem3,
                        2 * cols,
                        cols,
                        1,
                        cols,
                        1,
                        DSTR_OUTPUT);
```

This completes the first stage of setting up the image streams for the input and output as defined in the flow chart in section 2.4.

Step 4: This step requests data through the dtsr_get and dtsr_put calls on the input and output image streams for performing the algorithm in this case “copy_word”. The code makes use of the even and odd input image streams back to back, and writes data to the output stream, which in turn transfers it to the display buffer. The copy_word routine copies data from input to output using word wide copies. The intent of this function, was to merely highlight the setting up of the image streams required to support an algorithm that requires input lines in normal or progressive order

```
 /*----------------------------------------------------------------*/
 /* For all rows of the input even and odd field, merge the       */
 /* results by using the copy routine. The input data is obtained  */
 /* by calls to dstr_get and the output calls are obrtained using  */
 /* calls to dstr_put.                                            */
 /*----------------------------------------------------------------*/

for ( i = 0; i < ( rows + 1); i++)
```

```
{
    /*-------------------------------------------------------------*/
    /* Obtain input and output pointers. This call to dstr_get    */
    /* uses the even field. The call to dstr_put always operates  */
    /* on the output stream.                                     */
    /*-------------------------------------------------------------*/

    in_data  = (unsigned char *) dstr_get(&iev_dstr);
    out_data = (unsigned char *) dstr_put(&o_dstr);

    /*-------------------------------------------------------------*/
    /* Perform copy routine using word wide copies. The in_data   */
    /* and out_data are input and output pointers that the algo–  */
    /* rithm can use to perform the copy. Other more advanced     */
    /* user algorithms should go here.                           */
    /*-------------------------------------------------------------*/

    copy_word(in_data, out_data, cols);

    /*-------------------------------------------------------------*/
    /* Obtain input and output pointers. This call to dstr_get    */
    /* uses the odd field. The call to dstr_put always operates   */
    /* on the output stream.                                     */
    /*-------------------------------------------------------------*/

    in_data  =  (unsigned char *) dstr_get(&iod_dstr);
    out_data =  (unsigned char *) dstr_put(&o_dstr);

    /*-------------------------------------------------------------*/
    /* Perform copy routine using word wide copies. The in_data   */
    /* and out_data are input and output pointers that the algo–  */
    /* rithm can use to perform the copy. Other more advanced     */
    /* user algorithms should go here.                           */
    /*-------------------------------------------------------------*/

    copy_word(in_data, out_data, cols);
}
```

Step 5: This completes the stage of checking if all the data required by the algorithm is complete or not. The first call to put, informs the user of the output buffer space, that can be used to store results. Therefore one additional put call is required to commit the last set of output results to external memory

```
 /*-------------------------------------------------------------*/
 /* Commit last set of results to memory                        */
 /*-------------------------------------------------------------*/

 dstr_put(&o_dstr);
```

Step 6: The last step in the flow chart is to close all the image streams that were opened. The code to implement this step is shown:

```
dstr_close(&iev_dstr);
dstr_close(&iod_dstr);
dstr_close(&o_dstr);
```

This step convludes all the required steps that need to be followed when users interact with image streams.

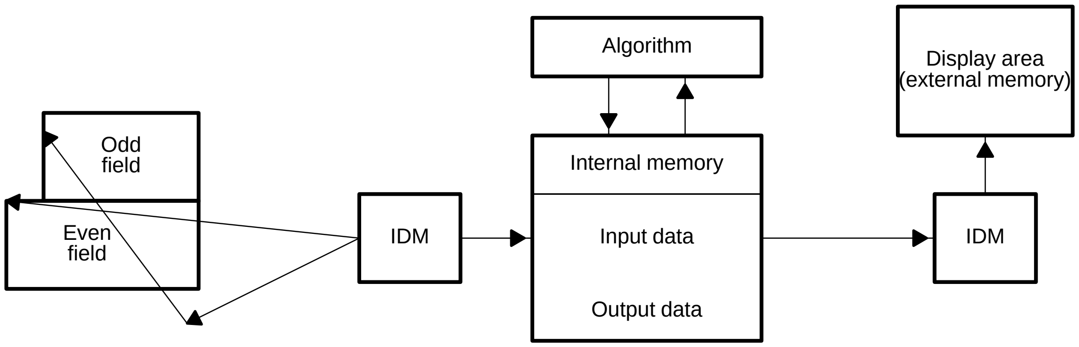
The algorithm itself in this case is very simple, but is shown for the sake of completeness:

```
/*-------------------------------------------------------------*/
/* Simple user algorithm illustrating word wide optimization on    */
/* a simple memcpy like operation. More advanced algorithms may    */
/* be written here.                                               */
/*-------------------------------------------------------------*/

void copy_word(unsigned char *inp, unsigned char *out, int cols)
{

    int i;

    /*-------------------------------------------------------------*/
    /* Cast incoming pointers to word aligned data. Image Data     */
    /* Manager will return aligned pointers so long as internal    */
    /* and external start addresses are aligned.                  */
    /*-------------------------------------------------------------*/

    unsigned int *input =  (unsigned int *) inp;
    unsigned int *output = (unsigned int *) out;

    /*-------------------------------------------------------------*/

    /* Use word wide accesses to load and store from input and     */
    /* output arrays respectively.                                */
    /*-------------------------------------------------------------*/

    for ( i = 0; i < (cols >> 2); i++)
    {
        output[i] = input[i];
    }
}
```

3.4.2 Gray Scale Application Driver for Odd-Even Field Based Processing

The application driver for field based processing is presented in this section. The captured data on the Imaging daughtercard is stored as two separate fields (odd and even fields), in three separate blocks (Y, Cr, Cb) in the frame buffer memory on the daughtercard. The next application driver is intended as a generic template for image processing applications that can use field based processing.

Figure 3–3. Data Flow for Field-Based Processing-Based Application Driver



The steps required to implement this code follow the steps shown in the flow chart described in section 2.4. Unlike the previous application driver that used separate input image streams for both the even and odd field, the present application driver, uses one input image stream and rewinds to the start of the odd output stream once the even image stream is exhausted. The following steps accomplish the different steps in the flowchart shown in section 2.4:

Step 1: This step opens an input image stream to fetch “num_lines” lines of the even field into internal memory. The variable num_lines allows the user to work on multiple lines at once for a given algorithm. The internal memory needs to be of size 2 * num_lines * cols, to hold the data as discussed earlier. Every access fetches “num_lines” lines each “cols” pixels wide and strides external memory by “cols” pixels as successive lines of the even field are contiguous in memory. The example included with the IDK sets num_lines to be 2

```
.  /*-------------------------------------------------------------*/
    /* Initizlize input stream to start fetching fom even field,    */
    /* of size 2 *rows * cols, including the even and odd fields    */
    /* into int_mem1 (internal memory) of size 2 * num_lines * cols,*/
    /* ”num_lines” lines of ”cols” pixels every time.              */
    /* Check returned error code if any                             */
    /*-------------------------------------------------------------*/

    err_code  = dstr_open ( &i_dstr,
                            in_image_ev->img_data,
                            (2 * rows * cols),
                            int_mem1,
                            2 * num_lines * cols,
                            cols,
                            num_lines,
                            cols,
                            1,
                            DSTR_INPUT);
```

Step 2: This stage opens an output image stream to write out the results obtained by processing multiple lines of the even or odd field. Hence successive lines need to be separated by a fixed offset in external memory. The output image stream is set up using the “dstr_open” call as shown. In this particular call note that the stride amount in external memory is set to “2*cols” to provide the fixed offset between successive lines that are written to external memory. The fact that multiple lines (num_lines in this case) need to be written out every time is also specified in the “dstr_open” call

```
/*-------------------------------------------------------------*/
/* Initialize output stream to write to out_image->img_data,   */
/* to fetch ”num_lines” lines of pixels ”cols” wide at a time  */
/* and paste into the output area 2 * cols apart. Once the     */
/* even input stream is exhausted, the input stream rewinds    */
/* the odd field and the output stream rewinds to the second   */
/* line and pastes 2 lines apart every time.                  */
/*-------------------------------------------------------------*/
err_code = dstr_open (  &o_dstr,
                        out_image–>img_data,
                        out_rows * out_cols,
                        int_mem2,
                        2 * num_lines * cols,
                        cols,
                        num_lines,
                        2 * cols,
                        1,
                        DSTR_OUTPUT);
```

Step 3: The following code implements two stages, namely the stage of getting the necessary data by the appropriate calls to dstr_get_2D and the dstr_pur_2D, and rewinding to the start of the odd field once the even field has been exhausted. The outer j loop implements the two passes, the first for the even field and the next for the odd field. The inner “i” loop iterates for all the lines within a field

```
/*--------------------------------------------------------------------*/
/* Process even field in one pass and odd field in a separate pass    */
/*--------------------------------------------------------------------*/
for (j = 0; j < 2; j++)
{
   /*-----------------------------------------------------------------*/
   /* Since ”num_lines” lines of pixels are brought in every time, the */
   /* inner loop runs for rows/num_lines. During this time pixels in   */
   /* all ”num_lines” are processed.                                  */
   /*-----------------------------------------------------------------*/
    for ( i = 0; i < (rows + num_lines)/num_lines; i++)|
    {
         /*-----------------------------------------------------------*/
         /* Obtain input and output pointers. This call to dstr_get_2D */
         /* uses the even field when j = 0, and the odd field when j   */
         /* = 1. The call to dstr_put_2D on the first time indicates   */
         /* to the user where to begin pasting into the output area.  */
         /*-----------------------------------------------------------*/
```

```
        in_data  = (unsigned char *) dstr_get_2D(&i_dstr);
        out_data = (unsigned char *) dstr_put_2D(&o_dstr);

        /*------------------------------------------------------------*/
        /* Perform copy routine using word wide copies. The in_data   */
        /* and out_data are input and output pointers that the algo–  */
        /* rithm can use to perform the copy.                        */
        /*------------------------------------------------------------*/

        copy_word(in_data, out_data, num_lines * cols);

    }

    /*----------------------------------------------------------------*/
    /* Commit last set of results to memory                           */
    /*----------------------------------------------------------------*/

    dstr_put_2D(&o_dstr);

   /*----------------------------------------------------------------*/
   /*  Rewind input and output streams                               */
   /*----------------------------------------------------------------*/

      dstr_rewind(&i_dstr, in_rewind, DSTR_INPUT, 1);
      dstr_rewind(&o_dstr, out_rewind, DSTR_OUTPUT, 1);
  }
```

The first call to dstr_put_2D informs the user, where the output results for the present iteration of the algorithm need to be written. Since this is issued, before any real outputs are available, one extra call to dstr_put_2D is required outside the loop to commit the last set of results to external memory. Once this has been accomplished, the image streams on the input and output side are re-wound. The input side image stream is reset to point to the start of the odd field, while the reset point for the output stream is to the first odd line of the output. The rewind of pointers is accomplished by calls to "dstr_rewind".

Step 4: Once the input and output streams have finished bringing all the data required for the algorithm, and committed all the processed results to memory, the image streams need to be closed. This is accomplished by the following code

```
.    /*------------------------------------------------------------------*/
     /* Close input and output stream.                                 */
     /*------------------------------------------------------------------*/

     dstr_close(&i_dstr);
     dstr_close(&o_dstr);
```

This completes the application driver development for the gray scale mode. The application code included is identical to the previous one and hence is not repeated here. It copies the data in the input line using word wide loads.

3.4.3 DSP Loading for Gray Scale Application Drivers

The current set of application drivers provided with the IDK work for a 640 by 480 image. The size of the output image, the application drivers work with, can be changed by programmers. This is illustrated by way of two applications that are included in the IDK. The performance of the application drivers for a 640 by 480 image is summarized below, in terms of CPU loading.

	Gray Scale	**Gray Scale Odd Even**
Percentage DSP Loading	21 %	23 %

This leaves about 75% of the DSP cycles for algorithmic processing. The DSP loading can be minimized further by processing 4 lines from either field at any given time. Changing the IDM data transfer calls to use EDMA/DMA will also help in reducing the DSP Loading. The DSP Loading is measured by using the RTDX capability of BIOSII, and using the CPU Utilization Graph.

3.5 Development of Color Application Drivers

The development of color application drivers is discussed in this section. The color, application drivers are also written to provide users with both progressive and field based processing. They make use of IDM to bring in the YCRCB data and perform the color space conversion routine and convert the data to RGB565 format. Therefore they make use of at least three input image streams and one output image stream. The current settings in the examples, provided with IDK are for a 640 by 480 image. This can be changed by the programmer to work for other resolutions that may be needed.

3.5.1 Color Application Driver for Progressive Order Using EDMAs to Merge Even/Odd Fields

The first color application driver that is developed relies on the EDMA/DMA transfer mechanism to perform the merge of the lines in the odd/even field in external memory, while the DSP blocks for completion of the DMAs. The data flow required in this case is similar to the one required for gray scale processing as discussed under Section 3.4.1. While this is not the way in which the final application driver is written, it is definitely worth the effort to consider the different possibilities. The application driver fetches the data from the merged buffer and performs color space conversion and writes out the results to the display area.

The code to merge the lines of the even and odd field is shown below:

```
/*-------------------------------------------------------------------*/
/* The input frame pointer contains six pointers, two each for the  */
/* Y, Cr and Cb data. These pointers are used to copy/capture data  */
/* (both fields) from daughter card to SDRAM (external memory)      */
/* by firing DMA transfers from these locations to arrays in        */
/* external memory. Wait on the transfer id of the last transfer    */
/* id to make sure tyhat transfers completed.                      */
/*-------------------------------------------------------------------*/

xfrid = DAT_Copy2D(DAT_1D2D, input–>y1,  ybuff, 640, 240, 640*2);
xfrid = DAT_Copy2D(DAT_1D2D, input–>cr1, crbuff, 320, 240, 320*2);
xfrid = DAT_Copy2D(DAT_1D2D, input–>cb1, cbbuff, 320, 240, 320*2);
xfrid = DAT_Copy2D(DAT_1D2D, input–>y2,  ybuff+640, 640, 240, 640*2);
xfrid = DAT_Copy2D(DAT_1D2D, input–>cr2, crbuff+320, 320, 240, 320*2);
xfrid = DAT_Copy2D(DAT_1D2D,  input–>cb2, cbbuff+320, 320, 240, 320*2);
DAT_Wait(xfrid);
```

This code makes use of the DAT transfer routines, to perform 2D copies, that merge the lines from the even field and the odd field into external buffers ybuff, crbuff, cbbuff. The IDM is used to perform color space conversion by setting up image streams from ybuff, crbuff, cbbuff to internal memory to perform the color space conversion routines and to write them out to the display area. The code required to do this, once again follows the steps as discussed under section 2.4.

Step 1: Open image streams to fetch data from ybuff, crbuff, cbbuff into internal memory. This is done using the following code and uses “dstr_open” calls. The addresses of ybuff, crbuff, and cbbuff are set in a structure which defines the image size and location. The luma data derives the address of the external buffer by reading the structure in_image and obtaining the address in_image_luma–>img_data

```
/*-------------------------------------------------------------------------*/
/*  This initializes a luma stream of data, and brings a line worth of    */
/* luma data to internal memory using Image Data Manager                  */
/*-------------------------------------------------------------------------*/

err_code = dstr_open(&i_luma,
                     in_image_luma–>img_data,
                     rows * cols,
                     int_luma,
                     (2 *cols),
                     cols,
                     1,
                     cols,
                     1,
                     DSTR_INPUT);
```

This completes the opening of the input image stream to fetch the luma data into internal memory. Similar code is uded to set up the image streams for CR and CB and the output stream and is shown below. Since all input streams operate out of merged buffers (that have the even and odd fields merged), there is no offset between successive luma (Y), CR and CB lines. The CB image data stream is similar to that of the CR and is not shown here, for conciseness

```
 *-----------------------------------------------------------------------*/
/* This initializes a Cr data stream to bring data into internal memory */
/* using the Image Data Manager                                          */
/*-----------------------------------------------------------------------*/

err_code = dstr_open(&i_cr,
                     in_image_Cr–>img_data,
                     (rows * cols) >> 1,
                     int_cr,
                     (cols),
                     cols >> 1,
                     1,
                     cols >> 1,
                     1,
                     DSTR_INPUT);
```

The output stream is opened, using the following lines of code:

```
/*-----------------------------------------------------------------------*/
/* Initialize output stream to transfer data from internal memory to    */
/* display buffer.                                                      */
/*-----------------------------------------------------------------------*/
```

```
err_code = dstr_open(&o_dstr,
                     out_image–>img_data,
                     (4 * rows * cols),
                     out_data,
                     (4 * cols),
                     2 * cols,
                     1,
                     2 * cols,
                     1,
                     DSTR_OUTPUT);
```

Step 2: This completes the opening and setup of all the input and output streams. The color space conversion routine is called to convert each line of YCRCB data to RGB565 format for display. This routine is hand optimized and maximizes the multiplier bandwidth of the architecture. The algorithm is implemented by converting each line of input through the color space conversion routine. The resulting output line of RGB565 is committed to external memory. The code shown, implements the color space conversion algorithm on the entire image

```
/*-------------------------------------------------------------------*/
/* For all rows of the image, ontain working luma (Y), Chroma CR, CB   */
/* signals and call color space conversion routine to perform trans–   */
/* formation from YCrCb to RGB data.                                  */
/*-------------------------------------------------------------------*/

for( i = 0; i < rows; i++)
{
    /*-----------------------------------------------------------------*/
    /* The call to dstr_get returns the current set of working Y, Cr   */
    /* and Cb pointers                                                 */
    /*-----------------------------------------------------------------*/

    Y  = (unsigned char *) dstr_get(&i_luma);
    CR = (unsigned char *) dstr_get(&i_cr);
    CB = (unsigned char *) dstr_get(&i_cb);

    /*-----------------------------------------------------------------*/
    /* The call to dstr_put specifies where the output RGB results are */
    /* written to.                                                    */
    /*-----------------------------------------------------------------*/

    RGB = (unsigned char *) dstr_put(&o_dstr);

    /*-----------------------------------------------------------------*/
    /* Call color space conversion routine to work on current data     */
    /*-----------------------------------------------------------------*/

    ycbcr422pl_to_rgb565_asm(coeffs, Y, CR, CB, RGB, cols);

}
```

Once all the input data required for the algorithm and the processed output data of the algorithm is committed back to external memory, the input and output streams are closed. The code that implements this is shown below. The first call to dstr_put is performed so that the user knows where the processed results are to be written in internal memory. Hence the last set of results need to be committed by performing a separate dstr_put call outside the loop, before closing the streams

```
/*----------------------------------------------------------------------*/
/* Commit last set of results. This is done by one extra put call out- */
/* side the main loop.                                                 */
/*----------------------------------------------------------------------*/

dstr_put(&o_dstr);

/*----------------------------------------------------------------------*/
/* Close the input Y, Cr and Cb streams                                 */
/*----------------------------------------------------------------------*/

dstr_close(&i_luma);
dstr_close(&i_cr);
dstr_close(&i_cb);

/*----------------------------------------------------------------------*/
/* Close output stream                                                  */
/*----------------------------------------------------------------------*/

dstr_close(&o_dstr);
```

3.5.2 Color Application Drivers for Progressive Order with DSP Doing Even/Odd Field Merge

The examples included as part of the IDK also include another application driver, that works on the image in progressive order. It is different from the previous approach where the DSP blocked on completion of the DMA transfer requests. The DSP performs the merge of the even and odd fields by opening six input streams, three for the even field and three for the odd field and does the merge as part of the transfer. It is exactly analogous to the gray scale driver that worked on the even and odd field back to back.

The code to initialize the six different input streams is not shown here for conciseness. The code that implements the color space conversion over the entire image is shown. The calls to the get routine switch between the even and odd fields so that they are processed back to back. There is no offset between successive output lines that are processed in this driver

```
/*----------------------------------------------------------------------*/
/* For all rows of the even and odd image field obtain luma Y, Cr and */
/* Cb data and call optimized color space conversion routine. Commit  */
/* results to display buffer. Update luma Y, Cr and Cb with pointers  */
/* for odd fields and commit results to external memory. In this ex-  */
/* ample 1 line of the image from the even field and 1 line of the    */
/* image from the odd field are processed back to back.              */
/*----------------------------------------------------------------------*/

for ( i = 0; i < rows; i++)
{
    /*---------------------------------------------------------------*/
    /* Obtain Y, Cr, Cb pointers for even field. The call to the put  */
    /* routine returns information on where the output results are to */
    /* be written.                                                    */
    /*---------------------------------------------------------------*/

    Y  = (unsigned char *) dstr_get(&i_luma_ev);
    Cr = (unsigned char *) dstr_get(&i_cr_ev);
    Cb = (unsigned char *) dstr_get(&i_cb_ev);
    RGB = (unsigned char *) dstr_put(&o_dstr);

   /*---------------------------------------------------------------*/
   /* Call optimized color space conversion routine for even field  */
   /*---------------------------------------------------------------*/

   ycbcr422pl_to_rgb565_asm(coeffs,  Y, Cr, Cb, RGB, cols);

   /*---------------------------------------------------------------*/
   /* Obtain Y, Cr, Cb pointers for odd  field. The call to the put  */
   /* routine returns information on where the output results are to */
   /* be written.                                                    */
   /*---------------------------------------------------------------*/

    Y  = (unsigned char *) dstr_get(&i_luma_od);
    Cr = (unsigned char *) dstr_get(&i_cr_od);
    Cb = (unsigned char *) dstr_get(&i_cb_od);
    RGB = (unsigned char *) dstr_put(&o_dstr);

    /*---------------------------------------------------------------*/
    /* Call optimized color space conversion routine for odd  field  */
    /*---------------------------------------------------------------*/

     ycbcr422pl_to_rgb565_asm(coeffs,  Y, Cr, Cb, RGB, cols);
}
/*-------------------------------------------------------------------*/
/* Commit  last set of results to memory by extra put call outside   */
/* the main loop.                                                    */
/*-------------------------------------------------------------------*/

dstr_put(&o_dstr);
```

Once all the data that is required by the algorithm has been fetched, and all processed results have been committed to external memory, the image streams need to be closed. The code that implements this is shown. This concludes this application driver that works on the even field and odd field back to back to produce the color application driver. The difference in performance between these two drivers is discussed in the performance section for color application drivers

```
/*-------------------------------------------------------------------*/
/* Close streams for evan and odd Y, Cr and Cb                       */
/*-------------------------------------------------------------------*/

dstr_close(&i_luma_ev);
dstr_close(&i_cr_ev);
dstr_close(&i_cb_ev);

dstr_close(&i_luma_od);
dstr_close(&i_cr_od);
dstr_close(&i_cb_od);
```

```
/*--------------------------------------------------------------------*/
/* Close output stream handle                                         */
/*--------------------------------------------------------------------*/

dstr_close(&o_dstr);
```

Notice that since the application driver works on both the even and odd field at the same time, six input streams are required. These six input streams are closed in this section once all the processing has been completed.

3.5.3 Color Application Drivers for Odd/Even Field Based Processing

The following color application driver is analogous to the second gray scale application driver discussed in this chapter. It works on multiple lines of the even field, and finishes processing all lines of the even field. It then starts to work on the odd field, by fetching a multiple number of lines from the odd field, till it finishes processing all lines from the odd field. Successive output lines that are produced need to be offset by one line. The code to initialize the input streams is not shown for conciseness. The code to initialize the output image stream is shown below.

Step 1: Open output stream

```
/*--------------------------------------------------------------------*/
/* Initizlize stream for output data                                  */
/*--------------------------------------------------------------------*/

err_code = dstr_open( &o_dstr,
                      out_image->img_data,
                      (2 * out_rows * out_cols),
                      out_data,
                      4 * num_lines * out_cols,
                      2 * out_cols,
                      num_lines,
                      4 * out_cols,
                      1,
                      DSTR_OUTPUT);
```

This application driver makes use of only three input image streams and one output stream. Once all the lines of the even field have been processed, the dstr_rewind field rewinds all image streams to point to the start of the odd field. Notice that every call to dstr_put for the output image stream will commit “num_lines” lines, each of size “2 * out_cols” (RGB 565 every pixel is represented as 16 bits), each of which is offset by “4 * out_cols” amongst each other.

Step 2: Implement color space conversion for all lines of the image. This is achieved by issuing calls to dstr_get_2D and dstr_put_2d. There are two loops in the code required to implement this. The outer loop iterates two times, performing the algorithm on all lines of the even field on the first iteration, and rewinding and performing the algorithm on all lines of the odd field on the next iteration. The inner loop implements the algorithm on all lines of a given field. As usual, the first call to dstr_put_2D is performed before any output has been processed, to inform the user of the pointer in internal memory to which output results can be written. The code that implements this stage is shown below. The ability to fetch multiple lines allows users to balance their data bandwidth to their processing load. The call to the color space conversion routine is issued once to work on multiple lines.

```
/*------------------------------------------------------------------------*/
/* The outer loop iterates twice, because it processes all the lines of   */
/* the even field first, followed by the lines of the odd field next.    */
/*------------------------------------------------------------------------*/
for ( j = 0; j < 2; j++)
{
   /*---------------------------------------------------------------------*/
   /* For all rows of the even and odd image field obtain luma Y, Cr and */
   /* Cb data and call optimized color space conversion routine. Commit  */
   /* results to display buffer. Update luma Y, Cr and Cb with pointers  */
   /* for odd fields and commit results to external memory. In this ex–  */
   /* ample 1 line of the image from the even field and 1 line of the    */
   /* image from the odd field are processed back to back.              */
   /*---------------------------------------------------------------------*/
   for ( i = 0; i < (rows + num_lines)/num_lines; i++)
   {
       /*-----------------------------------------------------------------*/
       /* Obtain Y, Cr, Cb pointers for even/odd field. The call to put  */
       /* routine returns information on where the output results are to */
       /* be written. These calls need to be to the 2D routines as mul–  */
       /* tiple lines need to be read and written at the same time.     */
       /*-----------------------------------------------------------------*/
       Y  = (unsigned char *) dstr_get_2D(&i_luma_ev);
       Cr = (unsigned char *) dstr_get_2D(&i_cr_ev);
       Cb = (unsigned char *) dstr_get_2D(&i_cb_ev);
       RGB = (unsigned char *) dstr_put_2D(&o_dstr);
      /*-----------------------------------------------------------------*/
      /* Call optimized color space conversion routine for even field  */
      /*-----------------------------------------------------------------*/
      ycbcr422pl_to_rgb565_asm(coeffs,  Y, Cr, Cb, RGB, cols * num_lines);
   }
   /*---------------------------------------------------------------------*/
   /* Commit last set of results to output buffer for both even/odd      */
   /* field.                                                            */
   /*---------------------------------------------------------------------*/
   dstr_put_2D(&o_dstr);
   /*---------------------------------------------------------------------*/
   /* Rewind luma, Cr and Cb streams to work on the odd field for the    */
   /* next iteration.                                                   */
   /*---------------------------------------------------------------------*/
   dstr_rewind(&i_luma_ev, luma_rewind, DSTR_INPUT, 1);
   dstr_rewind(&i_cr_ev,   cr_rewind,   DSTR_INPUT, 1);
   dstr_rewind(&i_cb_ev,   cb_rewind,   DSTR_INPUT, 1);
   dstr_rewind(&o_dstr,    out_rewind,  DSTR_OUTPUT, 1);

}
```

Step 3: Close all input and output image streams. This step completes the application driver by closing all input and output image streams that have been opened. This section of the code closes the three input streams and one output stream. Unlike the previous application driver that opened six input streams, three for the even field and three for the output field, the present application driver opens three input image streams and shares them between the two fields using the dstr_rewind function

```
   /*-----------------------------------------------------------------*/
   /* Close streams for Y, Cr and Cb                                  */
   /*-----------------------------------------------------------------*/

   dstr_close(&i_luma_ev);
   dstr_close(&i_cr_ev);
   dstr_close(&i_cb_ev);

   /*-----------------------------------------------------------------*/
   /* Close output stream handle                                      */
   /*-----------------------------------------------------------------*/

   dstr_close(&o_dstr);
```

3.6 DSP Loading for Color Application Drivers

The current set of color application drivers provided with the IDK work for a 640 by 480 image. The size of the output image, the application drivers work with, can be changed by programmers. The performance of the application drivers for a 640 by 480 image is summarized below, in terms of CPU loading.

	Progressive Using EDMA	**Progressive Using DSP**	**Field Based Processing**
DSP Loading	75%	61%	51%

This leaves about 75% of the DSP cycles for algorithmic processing. The DSP loading can be minimized further by processing 4 or more lines from either field at any given time. The current set of application drivers provided with the IDK process two lines at a time. Changing the IDM data transfer calls to use EDMA/DMA will also help in reducing the DSP Loading. The DSP Loading is measured by using the RTDX capability of BIOSII, and using the CPU Utilization Graph. All the color application drivers perform color space conversion. This allows for up to 50% of the DSP cycles to implement other algorithms using the color application drivers. The purpose of providing the application that works in progressive order using EDMAs, is to allow users to work on possibly some other algorithm using the DSP while the EDMA performs the merge. In the present application driver since there are no other algorithms, the DSP blocks.

3.7 Conclusions

This chapter provided several examples of generic templates that programmers can use to develop generic application drivers. The application drivers allow users to work with images in progressive order or to implement field based processing. The application drivers allow users to work on multiple lines, when field based processing is used. The application drivers for progressive order can be extended to work on multiple lines as well. The ability to work with multiple lines allows users to balance the data transfer bandwidth to the processing load of the algorithm. The use of these generic templates in developing new applications will be examined in the next chapter.

Chapter 4

Application Development and Prototyping Using Generic Templates

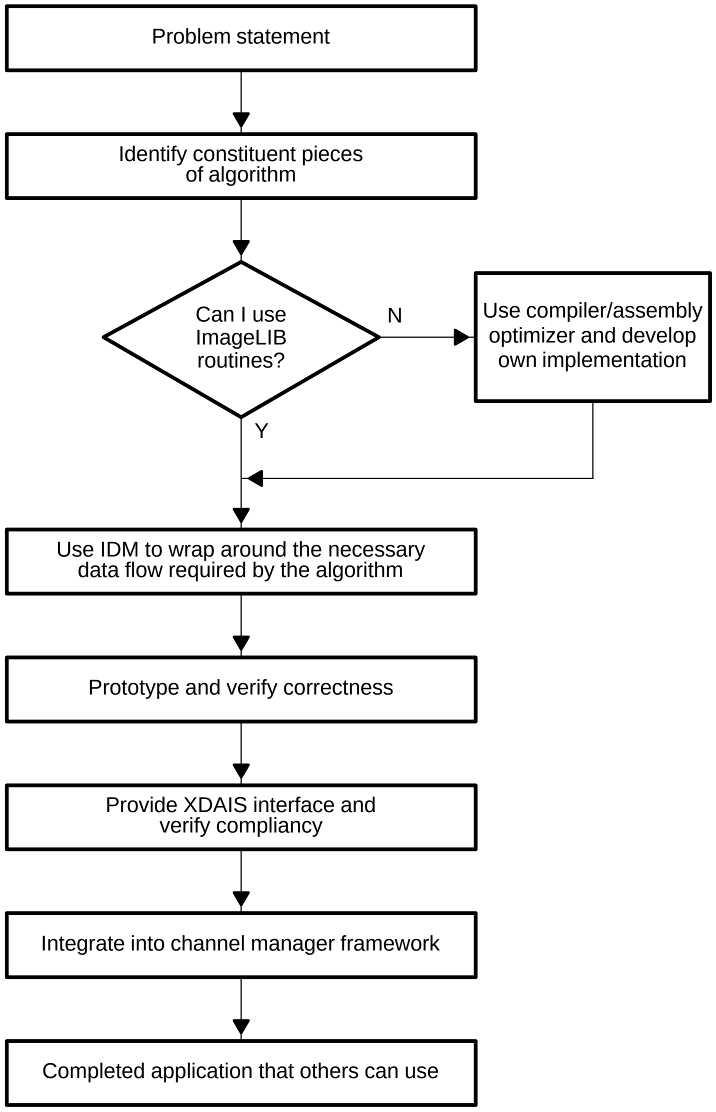
This chapter makes use of the generic templates developed in the previous chapter to aid users in initial application development and prototyping. This is the first stage of application development. This chapter presents a typical application development flow that is recommended based on our experience. Two examples are presented in this chapter as part of application development that leverage the application drivers developed in the previous chapter. These are the median filtering application and the color space rotation applications. These applications are not as complicated as JPEG or H.263 demonstrations included with the IDK. They are reduced in complexity, as they are intended to demonstrate the steps involved in the development of new applications as opposed to demonstrating the capabilities of the DSP on a complex algorithm.

Topic **Page**

4.1 Recommended Application Development Flow for IDK

Figure 4–1 shows the recommended algorithm development flow for the IDK. The flow chart shown is recommended based on the experience gained in implementing the more complicated demonstration scenarios that form a part of the IDK.

Figure 4–1. Recommended Algorithm Development Flow for IDK


Problem statement
Identify constituent pieces of algorithm
Can I use ImageLIB routines?
N
Use compiler/assembly optimizer and develop own implementation
Y
Use IDM to wrap around the necessary data flow required by the algorithm
Prototype and verify correctness
Provide XDAIS interface and verify compliancy
Integrate into channel manager framework
Completed application that others can use

4.2 Development of the Non-Linear Median Filtering Algorithm

4.2.1 Problem Statement

Develop an application that demonstrates the power of non-linear median filtering. The application should show the benefits of non-linear filtering by showing an image corrupted with salt and pepper noise. It should demonstrate the power of median filtering by filtering the noisy image with the median filter and displaying the cleaned up image side by side with the noisy image.

4.2.2 Use of ImageLIB in Developing an Application

The problem statement shown above forms the inspiration for the development of the included, grayscale application that forms part of the IDK. The recommended application flow chart shown in the previous section is followed to develop this application. The median_3x3 is a key image processing kernel. ImageLIB provides a collection of highly optimized functions for key image processing tasks that are C callable. The median_3x3 function from ImageLIB has an API that is obtained from the header file “median_3x3_h.h”. The API for this function is shown:

```
*     NAME                                                                  *
*          median_3x3                                                       *
*     USAGE                                                                 *
*           This routine is C–callable and can be called as:                *
*                                                                           *
*      void median_3x3_asm(unsigned char * in_data, int cols ,              *
*                          unsigned char * out_data);                       *
*       in_data  = pointer to input array of unsigned chars                 *
*       cols     = width of in_data                                         *
*       out_data = pointer to output array of unsigned chars                *
*                                                                           *
*           (See the C compiler reference guide.)                           *
*                                                                           *
*     DESCRIPTION                                                           *
*           The benchmark performs a 3x3 median filtering algorithm. It     *
*     comes under the class of non–linear signal processing algorithms.     *
*     Rather than replace the gray level at a pixel by a weighted average   *
*     of the nine pixels including and surrounding it, the gray level at    *
*     each pixel is replaced by the median of the nine values. The median   *
*     of a set of nine numbers is the middle element so that half of the    *
*     elements in the list are larger and half are smoother.  Median filt–  *
*     removes the effect of extreme values from data. Using a wide mask to  *
*     reduce the effect of noise results in un–acceptable blurring of sharp *
*     edges in the original image.                                          *
*                                                                           *
```

IMAGELIB routines are written to provide the user with the flexibility of varying the data bandwidth required to bring the data required for the algorithm with the processing load. It can be seen that the median_3x3 kernel can work on either a single line or multiple lines at a given time. A more comprehensive application that uses multiple ImageLIB function is discussed in the next chapter. The next step in the flow chart is to wrap around the raw ImageLIB kernel the necessary data flow required to implement the algorithm on the entire image.

4.2.3 Using IDM to Implement Data Flow for Algorithm

The median filter requires a sliding window mechanism, where it works on three input lines at a given time. It requires three new lines, for the first iteration, with each successive iteration dropping the oldest input line. This mechanism is already available in IDM. Following the steps that were shown in the application driver, IDM is used to implement the data flow required for the algorithm. This is implemented as a series of steps as shown in the flow chart in section 2.4. The complete code that provides the necessary data flow is shown.

Step 1:

```
/*------------------------------------------------------------------*/
/* Open input stream with the following parameters                  */
/*                                                                  */
/* External address: ext_ptr                                        */
/* External size:    ext_size                                       */
/* Internal memory:  int_mem                                        */
/* Internal size:    6 * cols                                       */
/* Number of bytes / line: cols                                     */
/* Number of lines/fetch:  1                                        */
/* External stride/line: 2 * cols                                   */
/* Sliding window:   3 lines                                        */
/* Direction: Input stream                                          */
/*------------------------------------------------------------------*/

err_code = dstr_open ( &i_dstr,
                      ext_ptr,
                      ext_size,
                      int_mem,
                    ( 6 * cols ),
                      cols,
                      1,
                      2 * cols,
                      3,
                      (DSTR_INPUT));
```

This opens an input stream to fetch data from an external memory location, into internal memory of size “2 * 3 * cols”, fetching one new line at a time and implementing the sliding window protocol. The opening of the image output stream for this application is done with the following lines of code.

```
/*------------------------------------------------------------------*/
/*  Open output stream with the following parameters                */
/*                                                                  */
/* External address: ext_ptr + cols                                 */
/* External size:    rows * cols                                    */
/* Internal memory:  corr_ptr                                       */
/* Internal size:    2 * cols                                       */
/* Number of bytes/line: cols                                       */
/* Number of lines:      1                                          */
/* External stride:      2 * cols                                   */
/* Direction:            output                                     */
/*------------------------------------------------------------------*/

err_code = dstr_open ( &o_dstr,
                       ext_ptr + (cols),
                      (2 * rows * cols),
                       corr_ptr,
                      (2 * cols),
                       cols,
                       1,
                       (2 * cols),
                       1,
                       (DSTR_OUTPUT));
```

where the start of internal memory is pointed to by int_mem, and the following pointers are defined.

```
/*------------------------------------------------------------------*/
/* ext_ptr is start of external memory for input data               */
/* corr_ptr is intenal memory where correlation result is stored    */
/* Since double buffering is used for 3x3 correlation, internal     */
/* memory needs to hold upto 6 columns                              */
/*------------------------------------------------------------------*/

char  *ext_ptr  = (char *) out_image->img_data;
char  *corr_ptr = int_mem + ( 6 * cols);
```

Notice that corr_ptr points to the location in internal memory, in which output results will be written into is allocated to be 6 lines past the start of internal memory pointed to by “int_mem”. This takes care of the buffering requirements for the input side as specified by the IDM. Recall that this requirement specifies that the size of the internal memory required for a stream needs to be twice as large as the transferred amount, namely (2 * (3*cols)). This data flow covers many common image processing algorithms such as correlation, convolution, object detection.

Step 2: Implement the core algorithm by repeated calls to the ImageLIB kernel

```
/*------------------------------------------------------------------*/
/* The following loop iterates rows times and computes 1 line of    */
/* median output.                                                   */
/*------------------------------------------------------------------*/
```

```
for ( i = 0; i < rows; i++)
{
    /*--------------------------------------------------------------*/
    /* in_data is pointer to input buffer                            */
    /* out_data is pointer to output buffer                          */
    /*--------------------------------------------------------------*/

    in_data  = (unsigned char *) dstr_get(&i_dstr);
    out_data = (unsigned char *) dstr_put(&o_dstr);

    /*--------------------------------------------------------------*/
    /* Perform 3 x 3 median algorithm to implement low pass         */
    /*--------------------------------------------------------------*/

    median_3x3_asm(in_data, cols, out_data);
    out_data[cols – 1 ] = out_data[cols – 2];
}
```

Step 3: Recall that the first call to the put routine, merely returns to the programmer the location where the output results can be stored. Hence an extra put call is required outside the main loop to commit the results. In addition once the output and input image streams have completed their jobs, they need to be closed.

```
/*------------------------------------------------------------------*/
/* Commit last output buffer and close input and output streams     */
/*------------------------------------------------------------------*/

dstr_put(&o_dstr);
dstr_close(&o_dstr);
dstr_close(&i_dstr);
```

4.2.4 Using Generic Templates to Feed Algorithm with Image Data

The next step is to actually feed the image processing algorithm with the required data. This is done by taking the generic templates developed in the previous chapter and identifying the one that best fits the present application. The problem definition states, that both the corrupted image and the median filtered image need to be shown side by side. This requires a split screen mode. The existing application level drivers work on a 640 by 480 image resolution for NTSC data. This needs to be modified to accommodate the split screen mode, in which the first image is the corrupted image created by random salt and pepper noise. The even_odd application driver template is used and modified as shown. The output stream is changed to send out lines which are half the display width. The declaration for the output image stream for the modified code is shown along with the code that creates a noisy reference image. The noisy image forms the input to the median filtering algorithm. Hence care should be taken to make sure that the output of the application driver is written to the correct address for the median filter’s input image stream to bring in. Once the noise has been injected on the reference image, and all lines have been processed, the streams are closed as before. The noise, itself is generated by calls to the rand() function in the run time support library. The code that generates the noise is not shown here for conciseness.

```
/*-------------------------------------------------------------*/
/* Open output stream to write to out_image–>img_data,         */
/* the merged fields back and the results of the algorithm.    */
/* into int_mem3 (internal memory), 1 line at a time.          */
/* Check for any error codes if any                            */
/*-------------------------------------------------------------*/

err_code = dstr_open  ( &o_dstr,
                        out_image–>img_data,
                        out_rows * out_cols,
                        int_mem3,
                        2 * cols,
                        cols >> 1,
                        1,
                        cols,
                        1,
                        DSTR_OUTPUT);

/*----------------------------------------------------------------*/
/* For all rows of the input even and odd field, merge the        */
/* results by using the copy routine. The input data is obtained  */
/* by calls to dstr_get and the output calls are obrtained using  */
/* calls to dstr_put.                                             */
/*----------------------------------------------------------------*/

for ( i = 0; i < ( rows + 1); i++)
{
    /*------------------------------------------------------------*/
    /* Obtain input and output pointers. This call to dstr_get    */
    /* uses the even field. The call to dstr_put always operates  */
    /* on the output stream.                                      */
    /*------------------------------------------------------------*/

    in_data  = (unsigned char *) dstr_get(&iev_dstr);
    out_data = (unsigned char *) dstr_put(&o_dstr);

    /*------------------------------------------------------------*/
    /* The copy routine copies the even pixels contained in       */
    /* ”in_data” into ”out_data”. Thus when the width of the      */
    /* input is ”cols”, the width of the output data produced     */
    /* by copy is ”cols >> 1”. To this output data, noise is      */
    /* injected.                                                  */
    /*------------------------------------------------------------*/

    copy_half(in_data, out_data, cols);
    noise(out_data, cols >> 1);

    /*------------------------------------------------------------*/
    /* Obtain input and output pointers. This call to dstr_get    */
    /* uses the odd field. The call to dstr_put always operates   */
    /* on the output stream.                                      */
    /*------------------------------------------------------------*/

    in_data  =  (unsigned char *) dstr_get(&iod_dstr);
    out_data =  (unsigned char *) dstr_put(&o_dstr);
```

```
        /*------------------------------------------------------------*/
        /* The  copy routine copies the even pixels contained in      */
        /* ”in_data” into ”out_data”. Thus when the width of the      */
        /* input is ”cols”, the width of the output data produced     */
        /* by copy is ”cols >> 1”. To this output data noise is       */
        /* injected.                                                  */
        /*------------------------------------------------------------*/

        copy_half(in_data, out_data, cols);
        noise(out_data, cols >>1);
    }

    /*----------------------------------------------------------------*/
    /* Commit last set of results to memory                           */
    /*----------------------------------------------------------------*/

    dstr_put(&o_dstr);

    /*----------------------------------------------------------------*/
    /* Close even and odd input streams and the output stream         */
    /*----------------------------------------------------------------*/

    dstr_close(&iev_dstr);
    dstr_close(&iod_dstr);
    dstr_close(&o_dstr);
```

4.2.5 Putting the Modules Together for Initial Testing

The modules that create the split screen mode driver, and the module that implements median filtering and writes the result comprehending the pitch have been developed. These two modules are called one after another in the code for the main task. The code that calls these two modules from the main task is shown.

```
/*-------------------------------------------------------------------------*/
/* Algorithm to be executed is setup as a task under BIOS which gets scheduled*/
/* by the TASK Manager.                                                    */
/*-------------------------------------------------------------------------*/

void tskMainFunc()
{
     /*--------------------------------------------------------------------*/
     /* Perform greyscale algorithm                                        */
     /*--------------------------------------------------------------------*/

     greyscale();
}

/*-------------------------------------------------------------------------*/
/* This is the greyscale algorithm where the luma channel alone is made use of*/
/*-------------------------------------------------------------------------*/

void greyscale()
{

    int frame_cnt;
    int hres;
    int vres;
```

```
IMAGE    in_image_ev;
IMAGE    in_image_od;
IMAGE    out_image;
SCRATCH_PAD scratch_pad;
/*------------------------------------------------------------------------*/
/* Configure capture hardware for square pixel and siplay hardware for    */
/* gray scale images.                                                     */
/*------------------------------------------------------------------------*/
VCAP_config(VCAP_SQP);
VDIS_config(VDIS_640X480_GS);
/*------------------------------------------------------------------------*/
/* Obtain the default horizontal resolution, vertical resolution, bits    */
/* per pixel and pitch settings for gray scale mode                       */
/*------------------------------------------------------------------------*/
hres  = VDIS_settings.hres;
vres  = VDIS_settings.vres;
/*-------------------------------------------------------------------------*/
/* Set the rows and columns parameter in the even and odd field image st-  */
/* ructures. The actual pointers to the image data will be returned by     */
/* calls to the imaging hardware using the library routines                */
/*-------------------------------------------------------------------------*/
in_image_od.img_cols = in_image_ev.img_cols = hres;
in_image_od.img_rows = in_image_ev.img_rows = vres >> 1;
/*-------------------------------------------------------------------------*/
/* Set the output rows and columns for the output image structure          */
/*-------------------------------------------------------------------------*/
out_image.img_cols = hres;
out_image.img_rows = vres;
/*-----------------------------------------------------------------------*/
/* The following loop, iterates  for a fixed period of one day, in which */
/* (24 hrs * 60 mts * 60 secs * 30 frmes/sec) are processed. After this  */
/* period the board hardware reests by a call to the cature and display  */
/* reset routines                                                        */
/*-----------------------------------------------------------------------*/
 for (frame_cnt = 0; frame_cnt <= ( 24 * 60 * 60 * 30); frame_cnt ++ )
{
    /*-----------------------------------------------------------------*/
    /* The current set of input and output pointers are obtained by    */
    /* calls to the getFrame and toggleBuffs routine. By using the     */
    /* SYS_FOREVER flag the function blocks until a new frame arrives  */
    /* By setting the output side argument to 0, the next available    */
    /* buffer is returned independent of the display event.            */
    /* ”input” and ”output” are pointers to input frame and output     */
    /* data.                                                           */
    /*-----------------------------------------------------------------*/
    input  = VCAP_getFrame(SYS_FOREVER);
    output = VDIS_toggleBuffs(0);
    /*-----------------------------------------------------------------*/
    /* Set the pointer for the even, odd input fields. Also set the    */
    /* pointer for image data on the output end                        */
    /*-----------------------------------------------------------------*/
```

```
        in_image_ev.img_data = input–>y1;
        in_image_od.img_data = input–>y2;
        out_image.img_data   = output;

        /*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
        /* Also set parameters for the scratch–pad data structure to point */
        /* to appropriate amounts of external and internal memory as requ– */
        /* ired by application.                                            */
        /*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/

        scratch_pad.ext_data = (char *) ext_mem;
        scratch_pad.ext_size = sizeof(ext_mem);
        scratch_pad.int_data = (char *) int_mem;
        scratch_pad.int_size = sizeof(int_mem);

        /*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
        /* User algorithm goes here. In this algorithm, the first block     */
        /* copy_image copies the even pixels of a given line, to form       */
        /* the split screen mode. It then corrupts the original data with   */
        /* salt and pepper noise. The median algorithm then uses the corr–  */
        /* upted input as source and produces a cleaned up image shown on   */
        /* the right half of the screen, while the noisy image is produced  */
        /* on the left half.                                                */
        /*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/

        copy_image(&in_image_ev, &in_image_od, &out_image, &scratch_pad);
        median3x3_image(&scratch_pad, &out_image);
    }

    /*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
    /* Reset the capture and display hardware.                             */
    /*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/

    VDIS_config(VDIS_RESET);
    VCAP_config(VCAP_RESET);
}
```

The programmer is asked to refer to the *TMS320C6000 Imaging Developer's Kit (IDK) Video Device Driver's User's Guide* (Literature number SPRU499) for more details on how the task is set up to run under BIOSII. This is set up using the BIOSII .cdb file, which is included as part of the project. The main task gets scheduled and runs periodically performing the algorithm. This same algorithm could have been split into two tasks, namely one of creating the half screen mode with the corrupted image and another that runs the median filter on this corrupted image to produce a cleaned up image.

This step completes the initial development and testing of the algorithm. It is by no means a complete solution. The algorithm still needs the eXpressDSP wrapper to make it eXpressDSP-compliant. This allows this algorithm to be integrated with other algorithms. The Channel Manager can then be used to manage all these algorithms and even configure them at run time.

4.3 Development of Color Plane Rotation Algorithm

4.3.1 Problem Definition

Develop a color plane rotation algorithm that performs rotation of the color plane using the following equations:

```
cr_temp = Cr[i] – 128;
cb_temp = Cb[i] – 128;

cr_int = (cr_temp * cosine) – (cb_temp * sine);
cb_int = (cr_temp * sine)   + (cb_temp * cosine);

cr_fin = (cr_int >> 15) + 128;
cb_fin = (cb_int >> 15) + 128;
```

This problem requires the manipulation of the components CR and CB. The image size is not in a split screen mode. Therefore the generic templates defined for color can be used without any changes to the IDM in the opening of the streams. The CR, CB components for each line are rotated on a per-frame basis. The code that sets up an initial angle and keeps it modulo (2*pi) is shown:

```
short cosine;
short sine;

static double ang = 0.0;
const  double PI  = 3.1415927;
ang += (PI / 120.0);
if (ang >= (2 * PI)) ang =  0;

cosine = 126*256 * cos(ang);
sine   = 126*256 * sin(ang);
```

The code required to provide the data flow for the algorithm is identical to that of the generic templates provided for color. Therefore the code that performs the opening of the image . The odd-even color application driver is chosen for this application as it is the fastest of all the application drivers available. The code that implements the calls to the rotate function before the color space conversion is shown.

```
 for ( j = 0; j < 2; j++)
{
   /*-------------------------------------------------------------------*/
   /* For all rows of the even and odd image field obtain luma Y, Cr and */
   /* Cb data and call optimized color space conversion routine. Commit  */
   /* results to display buffer. Update luma Y, Cr and Cb with pointers  */
   /* for odd fields and commit results to external memory. In this ex–  */
   /* ample 1 line of the image from the even field and 1 line of the    */
   /* image from the odd field are processed back to back.               */
   /*-------------------------------------------------------------------*/
   for ( i = 0; i < (rows + num_lines)/num_lines; i++)
   {
       /*----------------------------------------------------------------*/
       /* Obtain Y, Cr, Cb pointers for even field. The call to the put  */
       /* routine returns information on where the output results are to */
       /* be written.                                                     */
       /*----------------------------------------------------------------*/

       Y  = (unsigned char *) dstr_get_2D(&i_luma_ev);
       Cr = (unsigned char *) dstr_get_2D(&i_cr_ev);
       Cb = (unsigned char *) dstr_get_2D(&i_cb_ev);
       RGB = (unsigned char *) dstr_put_2D(&o_dstr);

       /*---------------------------------------------------------------*/
       /* Rotate Cr. Cb color plane using rotate routine                */
       /*---------------------------------------------------------------*/

       rotate(Cr, Cb, (cols * num_lines) >> 1, cosine, sine);

      /*---------------------------------------------------------------*/
      /* Call optimized color space conversion routine for even field  */
      /*---------------------------------------------------------------*/

      ycbcr422pl_to_rgb565_asm(coeffs,  Y, Cr, Cb, RGB, cols * num_lines);

   }

   dstr_put_2D(&o_dstr);

   dstr_rewind(&i_luma_ev, luma_rewind, DSTR_INPUT, 1);
   dstr_rewind(&i_cr_ev,   cr_rewind,   DSTR_INPUT, 1);
   dstr_rewind(&i_cb_ev,   cb_rewind,   DSTR_INPUT, 1);
   dstr_rewind(&o_dstr,    out_rewind,  DSTR_OUTPUT, 1);
}
   /*-------------------------------------------------------------------*/
   /* Close streams for odd Y, Cr and Cb                                 */
   /*-------------------------------------------------------------------*/

   dstr_close(&i_luma_ev);
   dstr_close(&i_cr_ev);
   dstr_close(&i_cb_ev);

   /*-------------------------------------------------------------------*/
   /* Close output stream handle                                         */
   /*-------------------------------------------------------------------*/

   dstr_close(&o_dstr);
}
```

This completes the modifications to the generic template required for implementing the color plane rotation. The code that implements the actual rotation as defined by the equations is also shown.

```
/*-------------------------------------------------------------------*/
/* Function that rotates Cr, Cb color plane using cosine and sine   */
/* values in Q15 math                                                */
/*-------------------------------------------------------------------*/

void rotate   (unsigned char *Cr, unsigned char *Cb, int x_dim,
               short cosine, short sine)
{
    int i;

    short cr_temp;
    short cb_temp;

    int cr_int;
    int cb_int;

    short cr_fin;
    short cb_fin;

    for(i = 0; i < x_dim; i++)
    {
       /*----------------------------------------------------------*/
       /* Center pixels around 0, by deducting 128 from both Cr and */
       /* Cb.                                                       */
       /*----------------------------------------------------------*/

       cr_temp = Cr[i] – 128;
       cb_temp = Cb[i] – 128;

       /*-----------------------------------------------------------*/
       /* Rotate Cr and Cb by multiplying using cosine and sine      */
       /*-----------------------------------------------------------*/

       cr_int = (cr_temp * cosine) – (cb_temp * sine);
       cb_int = (cr_temp * sine)   + (cb_temp * cosine);

       /*-----------------------------------------------------------*/
       /* Convert Q15 number and add 128 back to center chroma       */
       /* values around 128.                                         */
       /*-----------------------------------------------------------*/

       cr_fin = (cr_int >> 15) + 128;
       cb_fin = (cb_int >> 15) + 128;

       /*-----------------------------------------------------------*/
       /* Limit values to the range of 0–255                         */
       /*-----------------------------------------------------------*/

       Cr[i] = cr_fin & 0xFF;
       Cb[i] = cb_fin & 0xFF;
    }
}
```

This completes the initial development of this application for testing and verification. The step of converting this algorithm into a eXpressDSP-compliant algorithm and integration into the channel manager are not shown. These steps are illustrated in the Chapter 5 for the image processing demonstration developed in Chapter 4 using ImageLIB.

4.4 Conclusions

This chapter discussed by providing two examples, how the generic templates discussed in Chapter 3, allow programmers to develop applications for initial testing and verification. In particular it showed that in some cases the generic application drivers may have to be modified slightly to accommodate the data flow for the required algorithm. In other cases the generic application drivers may prove to be an exact match for the data flow expected by the algorithm. None the less, the presence of all these application drivers, should help programmers considerably in reducing their learning curve with the IDK.

Chapter 5

Image Processing Using ImageLIB

This chapter deals with the use of the C6000 ImageLIB in developing image processing applications. One of the demonstration scenarios developed as part of the IDK is the image processing demonstration, that makes use of the ImageLIB routines to perform image processing functions such as filtering, thresholding and edge detection. The median filtering algorithm developed in the previous chapter is another example of the use of ImageLIB. This chapter intends to focus on a more concrete example that follows the recommended application development flow from start to finish. The application that has been developed and prototyped in this chapter will be integrated into the Channel Manager framework.

Topic **Page**

5.1 Problem Definition

This demonstration is intended to highlight the following:

- ❑ The availability of several easy to use algorithms like Image Filtering, Image “thresholding”, Sobel Edge detection using TMS320C62x ImageLIB [2].
- ❑ The ease of use that the software architecture provides for algorithms to be customized by the user.

Figure 5–1 shows the standard algorithms used to create four channels. The Channel Manager sequences the operation of these channels. These algorithms were chosen for the purposes of this demonstration as they are fairly representative of a large class of common image processing tasks. The input image as well as results of the image processing functions will be simultaneously displayed as shown in Figure 5–2.

Figure 5–1. Image Processing Demonstration

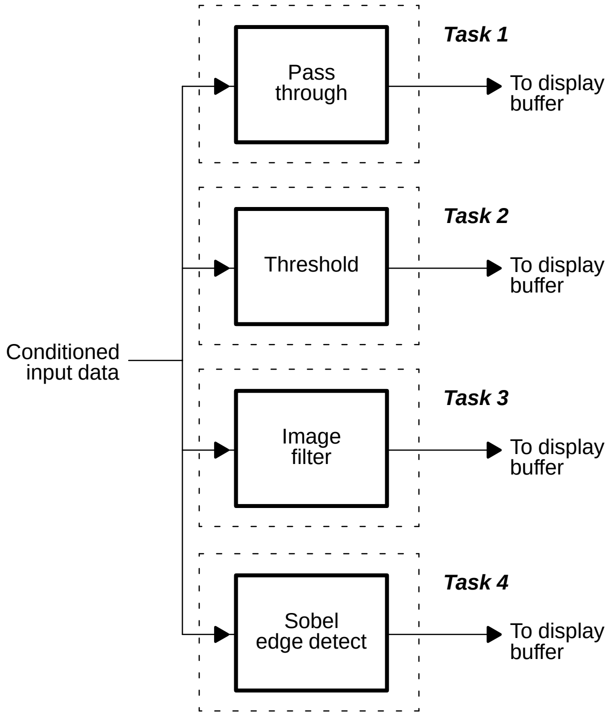
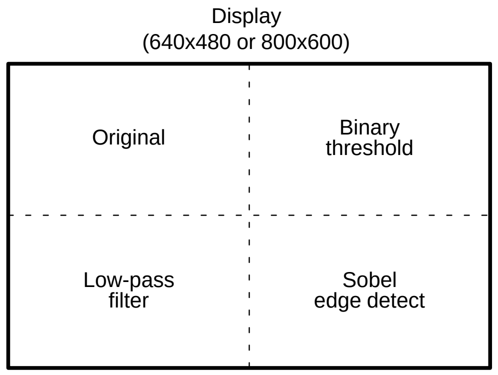


Figure 5–2. Image Processing Demonstration Display



5.1.1 Use of ImageLIB Components

The TMS320C62x Image/Video Processing Library provides a rich set of commonly used image processing functions that have been optimized for performance and code-size. As discussed earlier, the use of these routines either, in their existing form or with minor variations allow the user to perform several image processing algorithms. The following code example taken form ImageLIB shows the piped loop kernel for 3 by 3 correlation, where the input mask to be correlated is assumed to be a part of the image. The 3 by 3 correlation sum, is computed and then shifted into the byte range by a user defined shift amount, to display the output result in the byte range. The motivation behind reproducing the piped loop kernel here, is to impress the fact that these routines offer high performance, smaller code-size, flexibility and C callability for the user to exploit, without having to re-code these routines. The correlation, median_3x3 and threshold kernels have been used in the programmer's guide, to illustrate this philosophy. In the following piped loop kernel for corr_3x3 it can be seen that 18 multiplies, are performed in 9 cycles, maximizing the multiplier bandwidth of the architecture. The correlation kernel allows for the implementation of low pass filters that sum to 1, and have positive filter coefficients. The shift amount needs to be adjusted though, in accordance with the filter coefficients. It will be shown later, how users can develop their own convolution routine and plug it into the existing framework, if more general filters are desired. The ImageLIB library has been augmented with a convolution kernel, that performs all types of filters, saturating the output pixel result to either 0 or 255, since the first release of the library with CCS 1.2.

Figure 5–3. ImageLIB Code for corr_3x3 Kernel

```
LOOP:    ; PIPED LOOP KERNEL

   [!A1]   ADD     .D2     SP,B9,B1            ;  sum1 = (c00+c01+c02)
|| [!A1]   ADD     .L2X    A0,B13,B9           ;  sum1 = (c00+c01)
||         MV      .L1X    B1,A0               ;  a20  = a21
||         MPY     .M1     A3,A9,A3            ;  c00  = (a30*mask30)
||         MV      .S2     B3,SP               ;  a31  = a32
||         MPY     .M2     B11,B8,B9           ;  c01  = (a01*mask01)
||         LDBU    .D1T2   *–A11(1),B11        ;  a02  = pix02

   [!A1]   MPY     .M1X    A12,B5,A12          ;  c01  = (a31*mask31)
|| [!A1]   ADD     .L1     A13,A3,A12          ;  sum2 = c00+c01+c02
|| [!A1]   ADD     .L2X    B1,A14,B1           ;  sum1 = sum1+sum0
||         ADD     .S2     B9,B13,B13          ;  sum1 = (c00+c01+c02)
||         MPY     .M2     SP,B5,B3            ;  c01  = (a31*mask31)
||         LDBU    .D2T2   *B4++(2),B1         ;  a22  = pix22

   [!A1]   MV      .L1     A4,A5               ;
||         MPY     .M1X    B3,A15,A2           ;  c02  = (a32*mask32)
||         ADD     .S1     A5,A12,A14          ;  sum = sum+sum2
||         ADD     .L2     B13,B12,B12         ;  sum1 = sum1+sum0
||         MPY     .M2X    SP,A9,B10           ;  c00  = (a30*mask30)
||         LDBU    .D2T2   *–B4(1),B9          ;  a22  = pix22

   [ B0]   B       .S1     LOOP                ;  if(count) B LOOP
|| [!A1]   ADD     .L1X    B10,A12,A12         ;  sum2 = c00+c01
|| [!A1]   ADD     .L2X    A14,B1,B1           ;  sum = sum+sum1
|| [ B0]   ADD     .S2     0xfffffffe,B0,B0    ;  count––
||         MPY     .M2     DP,B7,SP            ;  c01  = (a21*mask21)
||         MPY     .M1     A2,A6,A12           ;  c00  = (a00*mask00)
||         LDBU    .D1T1   *A10++(2),A12       ;  a32  = pix32

           ADD     .L1     A12,A2,A2           ;  sum2 = c00+c01+c02
|| [!A1]   SHRU    .S2     B1,0xc,B1           ;  sum = sum>>shiftval
||         MPY     .M1X    B11,A6,A14          ;  c00  = (a00*mask00)
||         LDBU    .D1T2   *–A10(1),B3         ;  a32  = pix32
||         MPY     .M2X    A13,B8,B9           ;  c01  = (a01*mask01)

   [!A1]   ADD     .S1     A4,A2,A12           ;
|| [!A1]   STB     .D2T2   B1,*B2++(2)         ; *OUTP= sum
||         MPY     .M1     A13,A7,A12          ;  c02  = (a02*mask02)
||         ADD     .L1X    A12,B9,A2           ;  sum0 = (c00+c01)
||         MPY     .M2X    B11,A7,B13          ;  c02  = (a02*mask02)

           ADD     .L2X    A12,B12,B12         ;  sum  = sum1+sum
||         MPY     .M1     A0,A8,A0            ;  c00  = (a20*mask20)
||         ADD     .L1X    A14,B9,A0           ;  sum0 = (c00+c01)
||         MPY     .M2     B1,B7,B13           ;  c01  = (a21*mask21)
```

Figure 5–3. ImageLIB Code for corr_3x3 Kernel (Continued)

```
           SHRU    .S2     B12,0xc,DP          ;  sum2 = sum2>>shiftval
||         MPY     .M1X    DP,A8,A0            ;  c00  = (a20*mask20)
||         ADD     .L1     A2,A12,A14          ;  sum0 = (c00+c01+c02)
||         ADD     .L2X    A0,B13,B12          ;  sum0 = (c00+c01+c02)
||         MPY     .M2     B1,B6,B9            ;  c02  = (a22*mask22)
||         LDBU    .D1T1   *A11++(2),A13       ;  a02  = pix02
||         MV      .S1     A13,A2              ;  a00  = a01
||         MV      .D2     B11,B11             ;  a01  = a02

   [ A1]   SUB     .S1     A1,1,A1             ;
|| [!A1]   STB     .D2T2   DP,*–B2(1)          ;  *OUTP= sum2
||         ADD     .L2X    A0,SP,SP            ;  sum1 = (c00+c01)
||         ADD     .L1X    A3,B3,A13           ;  sum2 = c00+c01
||         MPY     .M1     A12,A15,A3          ;  c02  = (a32*mask32)
||         MPY     .M2     B9,B6,B13           ;  c02  = (a22*mask22)
||         MV      .S2     B9,DP               ;  a21  = a22
||         MV      .D1     A12,A3              ;  a30  = a31
```

5.2 Use of Image Data Manager to Manage Data Flow for ImageLIB Components

The availability of Image Data Manager makes the integration of the ImageLIB components into actual algorithms that work on the whole image a simple task. The following code shows an example for the ImageLIB correlation corr_3x3_image rotuine, that requires three lines of the input image as input at any given time, sliding down one line at a time. The output needs to be double buffered so that processing can continue while DMA transfers take place in the background. In later sections a custom convolution kernel that can be developed and integrated in to the present system will be discussed. Since both these algorithms require the same number of input lines and produce the same nuber of output lines, the code shown below can also be used for accomplishing convolution on the whole image.

```
/*----------------------------------------------------------------------*/
/* Include header files for functions to be called from this routine     */
/*----------------------------------------------------------------------*/

#include ”img_proc.h”
#include ”dstr_2D.h”
#include ”corr3x3_wr_p.h”

/*----------------------------------------------------------------------*/
/* If debug is defined include log objects for verifying correct init–   */
/* ialization of streams.                                              */
/*----------------------------------------------------------------------*/

#ifdef DEBUG
#include <std.h>
#include <log.h>
extern LOG_Obj trace;
#endif

/*----------------------------------------------------------------------*/
/* void corr3x3_image( SCRATCH_PAD * scratch_pad, IMAGE * out_image,     */
/*                     char        * out_ptr,     int   pitch,           */
/*                     LPF_PARAMS  * lpf_params )                        */
/*                                                                       */
/* This function sets up the input and output streams for performing     */
/* 3 x 3 correlation  and accepts the following parameters               */
/*                                                                       */
/* Arguments:                                                            */
/*          scratch_pad: external and internal scratch pad memory       */
/*          out_image:   pointer to output image                         */
/*          out_ptr:     pointer location to start pasting into          */
/*          pitch:       pitch argument to be used                       */
/*          lpf_params:  low pass filter parameters                      */
/*----------------------------------------------------------------------*/

void corr3x3_image(SCRATCH_PAD *scratch_pad, IMAGE *out_image,
                   char        *out_ptr,     int   pitch,
                   LPF_PARAMS  *lpf_params)
{
```

```
/*-------------------------------------------------------------------*/
/* Obtain external and internal start addresses and their respective */
/* sizes.                                                           */
/*-------------------------------------------------------------------*/

char *int_mem   = scratch_pad->int_data;
char *ext_mem   = scratch_pad->ext_data;
int   int_size  = scratch_pad->int_size;
int   ext_size  = scratch_pad->ext_size;

/*-------------------------------------------------------------------*/
/* Set rows and cols to be half of the output image rows and cols    */
/*-------------------------------------------------------------------*/

int   rows      =  (out_image->img_rows >> 1);
int   cols      =  (out_image->img_cols >> 1);

/*------------------------------------------------------------------*/
/* ext_ptr is start of external memory for input data               */
/* corr_ptr is intenal memory where correlation result is stored    */
/* Since double buffering is used for 3x3 correlation, internal     */
/* memory needs to hold upto 6 columns                              */
/*------------------------------------------------------------------*/

char  *ext_ptr  = ext_mem;
char  *corr_ptr = int_mem + ( 6 * cols);

/*-------------------------------------------------------------------*/
/* The low pass filter in internal memory is set to the last line    */
/* The external low pass filter coefficient address is lpf_ext       */
/*-------------------------------------------------------------------*/

unsigned char *lpf_ptr = (unsigned char *)(int_mem + int_size
                                        – ( cols >> 1));
unsigned char *lpf_ext = lpf_params–>mask;

unsigned char *in_data, *out_data;
int            shift;
dstr_t        i_dstr, o_dstr;
int           err_code;
int i, j;

/*-------------------------------------------------------------------*/
/* Copy low pass filter coefficients from external memory to inter–  */
/* nal memory.                                                      */
/*-------------------------------------------------------------------*/

lpf_ptr[0] = lpf_ext[0];
lpf_ptr[1] = lpf_ext[1];
lpf_ptr[2] = lpf_ext[2];
lpf_ptr[3] = lpf_ext[3];
lpf_ptr[4] = lpf_ext[4];
lpf_ptr[5] = lpf_ext[5];
lpf_ptr[6] = lpf_ext[6];
lpf_ptr[7] = lpf_ext[7];
lpf_ptr[8] = lpf_ext[8];

/*------------------------------------------------------------------*/
/* Read shift argument from user passed location.                  */
/*------------------------------------------------------------------*/

shift = lpf_params–>shift;
```

```
/*------------------------------------------------------------------*/
/* Initialize input stream with the following parameters          */
/*                                                                  */
/* External address: ext_ptr                                        */
/* External size:    ext_size                                       */
/* Internal memory:  int_mem                                        */
/* Internal size:    6 * cols                                       */
/* Number of bytes / line: cols                                     */
/* Number of lines/fetch:  1                                        */
/* External stride/line: cols                                       */
/* Sliding window:   3 lines                                        */
/* Direction: Input stream                                          */
/*------------------------------------------------------------------*/

err_code = dstr_open( &i_dstr,
                      ext_ptr,
                      ext_size,
                      int_mem,
                    ( 6 * cols ),
                      cols,
                      1,
                      cols,
                      3,
                      (DSTR_INPUT));

/*------------------------------------------------------------------*/
/* If DEBUG is defined check for any error messages in LOG object   */
/*------------------------------------------------------------------*/

#ifdef DEBUG
if (err_code)
{
    LOG_printf(&trace,”Error opening input stream %d \n”,
       err_code);
}
#endif

/*------------------------------------------------------------------*/
/*  Initialize output stream with the following parameters         */
/*                                                                  */
/* External address: out_ptr                                        */
/* External size:    rows * cols                                    */
/* Internal memory:  corr_ptr                                       */
/* Internal size:    2 * cols                                       */
/* Number of bytes/line: cols                                       */
/* Number of lines:      1                                          */
/* External stride:      pitch                                      */
/* Direction:            output                                     */
/*------------------------------------------------------------------*/
```

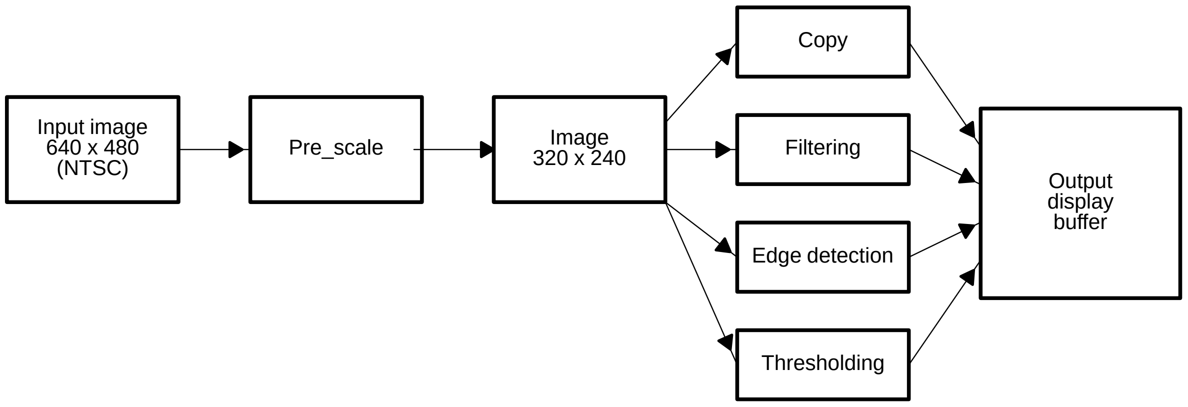
```
     err_code = dstr_open( &o_dstr,
                           out_ptr,
                          (2 * rows * cols),
                           corr_ptr,
                          (2 * cols),
                           cols,
                           1,
                           pitch,
                           1,
                           (DSTR_OUTPUT));

    /*-----------------------------------------------------------------*/
    /* If DEBUG is defined then echo any error messages to output log   */
    /*-----------------------------------------------------------------*/

    #ifdef DEBUG
    if (err_code)
    {
         LOG_printf(&trace,"Error opening output stream %d \n",
            err_code);
    }
    #endif

    /*-----------------------------------------------------------------*/
    /* The following loop iterates rows times and computes 1 line of    */
    /* correlation output.                                             */
    /*-----------------------------------------------------------------*/

    for ( i = 0; i < rows; i++)
    {

        /*-------------------------------------------------------------*/
        /* in_data is pointer to input buffer                           */
        /* out_data is pointer to output buffer                         */
        /*-------------------------------------------------------------*/

        in_data  = (unsigned char *) dstr_get(&i_dstr);
        out_data = (unsigned char *) dstr_put(&o_dstr);

        /*-------------------------------------------------------------*/
        /* Perform 3 x 3 correlation algorithm to implement low pass    */
        /*-------------------------------------------------------------*/

        corr3x3_wr(in_data, out_data, 0, cols, (char *)lpf_ptr, shift);
        out_data[cols – 1 ] = out_data[cols – 2];
    }

    /*-----------------------------------------------------------------*/
    /* Commit last output buffer and close input and output streams     */
    /*-----------------------------------------------------------------*/

    dstr_put(&o_dstr);
    dstr_close(&o_dstr);
    dstr_close(&i_dstr);

}
```

As shown above, every ImageLIB component requires a component_image file that is used to call it several times to accomplish the algorithm on the entire image. The number of times, the component is called needs to be tuned based on the data bandwidth required for the algorithm to the processing bandwidth. In this particular case one line of output is processed every time, and the sliding window moves the input data up one line at a time, bringing the most recent input line at the bottom of the buffer.

5.2.1 API interface for Image Processing Algorithms

The following code demonstrates the high level function which calls out to each image processing algorithm. The function img_proc may be called as such or each individual algorithm within this function can be converted to eXpressDSP so that it can be invoked by the Imaging Framework [3]. This issue will be examined later. However, an examination of the code provided below shows the simplicity with which all algorithms can be relocated on the display window either at run-time or compile time. The input image from the IDK hardware is a 640x480 image. Pre-scaling changes the image size to be 320x240. The vertical resizing is achieved by using only the even field, while the horizontal resizing is achieved by averaging. The information regarding the location and size of the image is passed in a structure as shown. In addition the pointer to the internal and external memory space is also stored in a SCRATCH_PAD structure which is shown below. The code is structured to accomplish the solution for the stated problem definition using the hierarchy as shown in Figure 5–4.

Figure 5–4. Structuring Algorithms to Accomplish Image Processing Demonstration



```
typedef struct image
{
    unsigned char *img_data;
    int            img_cols;
    int            img_rows;
}IMAGE;
```

```
typedef struct
{
    char          *ext_data;
    int            ext_size;
    char          *int_data;
    int            int_size;
}SCRATCH_PAD;

typedef enum img_type
{
    FLDS,
    PROG
} IMG_TYPE;
```

With, these structure definitions in place, the code that calls out to each image processing operation can now be examined. The code shown below sets up the algorithms to follow the structure outlined in Figure 5–4.

```
/*-----------------------------------------------------------------------------*/
/* Specify the header files of the functions to be used for image processing    */
/*-----------------------------------------------------------------------------*/

#include ”pre_scale_image.h”
#include ”copy_image.h”
#include ”corr3x3_image.h”
#include ”sobel_image.h”
#include ”threshold_image.h”

/*-----------------------------------------------------------------------------*/
/* Specify statistics object conditionally if debug is to be implemented using  */
/* BIOS.                                                                        */
/*-----------------------------------------------------------------------------*/

#ifdef _DSP_BIOS_DBG
#include <std.h>
#include <clk.h>
#include <trc.h>
#include <sts.h>
extern far STS_Obj stsLogPrintf;
#endif

/*-----------------------------------------------------------------------------*/
/* The following function sets the pointer pointed to by ptr_ptr. It is used by */
/* all the image processing algorithms to set their output pointer.            */
/*                                                                              */
/* void set_ptr(int quadrant, IMAGE *out_img, char **ptr_ptr)                  */
/*                                                                              */
/* Arguments:                                                                   */
/*            quadrant: Quadrant number {0, 1, 2, 3} allowed values            */
/*            out_img:  Structure containing parameters of output image        */
/*            ptr_ptr:  Pointer to pointer to store out the output pointer     */
/*-----------------------------------------------------------------------------*/

void set_ptr(int quadrant, IMAGE *out_img, char **ptr_ptr)
{
  /*---------------------------------------------------------------------------*/
  /* Read the number of columns and rows for the output image from the structure*/
  /* In addition set the pitch to be cols wide.                                */
  /*---------------------------------------------------------------------------*/
```

```
   int  cols     =  out_img->img_cols >> 1;
   int  rows     =  (out_img->img_rows >> 1);
   int  pch      =  out_img->img_cols;

  /*----------------------------------------------------------------------------*/
  /* By default assume that quadrant is 0 and set output pointer to point here. */
  /* Otherwise compute the cortrect pointer location  and set it in the output  */
  /* pointer                                                                    */
  /*----------------------------------------------------------------------------*/

   *ptr_ptr = (char *) out_img->img_data;
   if (quadrant == 1) *ptr_ptr += cols;
   if (quadrant == 2) *ptr_ptr += (rows * pch);
   if (quadrant == 3) *ptr_ptr += ((rows * pch) + cols);
}

/*------------------------------------------------------------------------------*/
/* Core image processing routine.                                              */
/*                                                                             */
/* void img_proc( IMAGE *in_image, IMAGE *out_img, SCRATCH_PAD *scratch_pad,    */
/*                img_type img_type_val,   LPF_PARAMS *lpf_params,             */
/*                THRESH_PARAMS *thresh_params )                               */
/*                                                                             */
/* in_image: Structure to the input image                                      */
/* out_img:  Structure to the output image                                     */
/* img_type_val: Type of image FLDS or PROG                                    */
/* lpf_params: Parameters to low pass filter                                   */
/* thresh_params: Threshold arguments                                          */
/*------------------------------------------------------------------------------*/

void img_proc( IMAGE *in_image,
               IMAGE *out_img,   SCRATCH_PAD *scratch_pad,
               img_type img_type_val, LPF_PARAMS *lpf_params,
               THRESH_PARAMS *thresh_params)
{

   int time;
   int  quadrant;
   char *out_ptr;
   int pitch, rows, cols;

   /*---------------------------------------------------------------------------*/
   /* Set output pointer to point to start of image. Read pitch value , cols    */
   /* and rows from the output image.                                          */
   /*---------------------------------------------------------------------------*/

   out_ptr = (char *) out_img->img_data;

   pitch=  out_img->img_cols;
   cols =  out_img->img_cols >> 1;
   rows =  (out_img->img_rows >> 1);

   /*---------------------------------------------------------------------------*/
   /* If DSP_BIOS debug is defined use time object to get statistics            */
   /*---------------------------------------------------------------------------*/
```

```
    #ifdef _DSP_BIOS_DBG
    if (TRC_query(TRC_USER0) == 0)
    {
        time = CLK_gethtime();
        STS_set(&stsLogPrintf, time);
    }
    #endif

    /*-----------------------------------------------------------------------------*/
    /* Perform pre-scaling of image by specifying type of image.                  */
    /*-----------------------------------------------------------------------------*/

    pre_scale_image(in_image, out_img, scratch_pad, FLDS);

    /*----------------------------------------------------------------------------*/
    /* Copy original image to the third quadrant                                 */
    /*----------------------------------------------------------------------------*/

    quadrant = 3;
    set_ptr(quadrant, out_img, &out_ptr);
    copy_image(scratch_pad, out_img, out_ptr, pitch);

    /*----------------------------------------------------------------------------*/
    /* Perform low pass filtering and copy results to zeroeth quadrant           */
    /*----------------------------------------------------------------------------*/

    quadrant = 0;
    set_ptr(quadrant, out_img, &out_ptr);
    conv3x3_image(scratch_pad, out_img, out_ptr, pitch, lpf_params);

     /*----------------------------------------------------------------------------*/
     /* Perform sobel and copy to second quadrant                                 */
     /*----------------------------------------------------------------------------*/

    quadrant = 2;
    set_ptr(quadrant, out_img, &out_ptr);
    sobel_image(scratch_pad, out_img, out_ptr, pitch);

    /*----------------------------------------------------------------------------*/
    /*  Perform thresholding and copy to 1st quadrant                            */
    /*----------------------------------------------------------------------------*/

    quadrant = 1;
    set_ptr(quadrant, out_img, &out_ptr);
    threshold_image(scratch_pad, out_img, out_ptr, pitch, thresh_params);

    /*----------------------------------------------------------------------------*/
    /* If DSP_BIOS_DBG is defined stop accumulating statistics                   */
    /*----------------------------------------------------------------------------*/

    #ifdef _DSP_BIOS_DBG
    if (TRC_query(TRC_USER0) == 0)
    {
        time = CLK_gethtime();
        STS_delta(&stsLogPrintf, time);
    }
    #endif
}
```

The relocation of the algorithm results to the appropriate quadrant is performed by the function set_ptr that accepts a quadrant and then sets the appropriate pointer. Pitch is set to the column width of the display. However this allows for display techniques like overlay. The quadrant can be passed in as an array of values that can be changed either at compile time or run time.

5.3 Modifying the Application Driver for the Algorithm

Chapter 3 examined the development of application drivers as generic templates that could be used for image processing algorithms. The problem definition for the image processing demonstration requires four different images to be displayed at the same time, with one channel being the pass through. This requires modifying the generic application driver to produce a 320x240 image in the case of NTSC. The code to do this is examined in the function pre_scale_image.c and pre_scale.c. The output image stream needs to be modified to produce half as many pixels as the input stream. Further, only the even field or odd field is processed to achieve the reduction in size in the vertical direction. Pixels are averaged along the horizontal direction to get better quality in the output image. The modified application driver uses the flow chart defined in section 2.4. the code to open the output stream is changed as follows, from the application driver for field based processing. Recall that this application driver allows users to process multiple lines from a given field at once

```
/*----------------------------------------------------------------------*/
/* Initialize output stream with the following parameters              */
/*                                                                      */
/* External address: ext_scale_start                                    */
/* External memory size: ext_size                                       */
/* Internal address: ptr_scale                                          */
/* Internal size:    cols * num_lines                                   */
/* Number of bytes/line: (cols * num_lines) >> 1                        */
/* Number of lines: 1                                                   */
/* External memory offset: (cols * num_lines) >> 1                      */
/* Window size:  1                                                      */
/* Direction: DSTR_OUTPUT                                               */
/*----------------------------------------------------------------------*/

err_code = dstr_open ( &o_dstr,
                       (void *) (ext_scale_start),
                       (ext_size),
                       (void *) (ptr_scale),
                       (cols * num_lines),
                       (cols * num_lines) >> 1,
                        1,
                       (cols * num_lines) >> 1,
                       (1),
                       (DSTR_OUTPUT));
```

Notice that this modified initialization accounts for the fact that the output stream produces half as much data as the input stream.

The next step is to issue calls to the input and output stream to accomplish the task of scaling down the image. The code to accomplish this is shown below:

```
    /*----------------------------------------------------------------------*/
    /* The following loop iterates through all the lines and calls the      */
    /* pre_scale code that scales the image from 640 by 480 to 340 by 240   */
    /* in_data is the pointer to the present input buffer and out_data is   */
    /* the pointer to the present output buffer                             */
    /*----------------------------------------------------------------------*/

    for ( i = 0; i < (rows / num_lines); i++)
    {
        in_data =  (unsigned char *) dstr_get_2D(&i_dstr);
        out_data = (unsigned char *) dstr_put(&o_dstr);

        pre_scale(in_data, out_data, (cols * num_lines));
    }
```

Once all the required data for the input and output stream has been fetched and processed, the input and output streams need to be closed. This is done using the following code. As in previous application driver examples, one extra call to the put function is required to commit the last set of results to memory

```
    /*----------------------------------------------------------------------*/
    /* Commit the last output buffer by isssuing dstr_put on output side    */
    /* Also close the input and output streams                              */
    /*----------------------------------------------------------------------*/

    dstr_put(&o_dstr);
    dstr_close(&i_dstr);
    dstr_close(&o_dstr);
```

This completes the process of modifying the generic templates as required for this application. The actual function that does the scaling in the horizontal dimension is shown below:

```
/*----------------------------------------------------------------------*/
/* void pre_scale(unsigned char *in_data, unsigned char *out_data,      */
/*                int num_lines)                                        */
/* in_data: pointer to input buffer                                     */
/* out_data: pointer to output buffer                                   */
/* num_lines: number of pixels to process                               */
/*----------------------------------------------------------------------*/

void pre_scale(unsigned char *in_data, unsigned char *out_data,
               int num_lines)
{

    /*--------------------------------------------------------------------*/
    /* in_ptr: int pointer to load pixels as integer values               */
    /* iters:  num_lines >> 2 since four pixels are processed together    */
    /*--------------------------------------------------------------------*/

    unsigned int *in_ptr  = (unsigned int *) (in_data);
    int iters    = (num_lines >> 2);
    int i;

    /*--------------------------------------------------------------------*/
    /* Mask to extract pairs of bytee. mask_3x1x extracts third and first */
    /* byte. Similarly mask_x2x0 extracts 2nd and 0th byte                */
    /*--------------------------------------------------------------------*/
    unsigned int mask_3x1x = 0xFF00FF00;
    unsigned int mask_x2x0 = 0x00FF00FF;

    /*-------------------------------------------------------------------*/
    /* pix_x3210: int containing four pixels                             */
    /* pix_x3x1: integer containing third and first pixels               */
    /* pix_x2x0: integer containing second and 0th pixels                */
    /*-------------------------------------------------------------------*/

    unsigned int pix_3210;
    unsigned int pix_x3x1;
    unsigned int pix_x2x0;
    unsigned int pix_avg;

    unsigned short pix_lo;
    unsigned short pix_hi;

    /*-------------------------------------------------------------------*/
    /* The following loop iterates and performs pre–scaling              */
    /*-------------------------------------------------------------------*/

    for ( i = 0; i < iters; i++)
    {
        /*---------------------------------------------------------------*/
        /* pix_x3210 performs word wide loads of the pixels              */
        /* pix_x3x1 contains the extercated 3rd and 1st pixels           */
        /* pix_x2x0 contains second and 0th pixels                       */
        /*---------------------------------------------------------------*/

        pix_3210 = in_ptr[i];
        pix_x3x1 = (pix_3210 & mask_3x1x) >> 8;
        pix_x2x0 = (pix_3210 & mask_x2x0);

        /*---------------------------------------------------------------*/
        /* pix_avg: perform averaging by add2 instruction                */
        /* Extract lower and higher halves into pix_lo and pix_hi and    */
        /* scale by 0.5                                                  */
        /*---------------------------------------------------------------*/

        pix_avg  = _add2(pix_x3x1, pix_x2x0);
        pix_lo   = (pix_avg & 0x0000FFFF) >> 1;
        pix_hi   = _mpyhuls(pix_avg, 1) >> 1;

        /*---------------------------------------------------------------*/
        /* Store out the lower and higher averages into output array     */
        /*---------------------------------------------------------------*/

        out_data[2*i]   = (unsigned char) pix_lo;
        out_data[2*i+1] = (unsigned char) pix_hi;
     }
}
```

The pre_scale function accesses the input data using word wide accesses and performs averaging of successive pixels to produce an output image. The resulting output pixels are limited to a byte range and stored out.

5.4 Algorithm Integration for Initial Testing

With all the kernel wrapper functions and the modified application driver file completed, the different pieces are now ready to be put together to accomplish the algorithm. For the purposes of initial testing all four iamge processing algorithms are scheduled as one task under BIOSII. This will be modified in the next chapter when they are integrated to be eXpressDSP-compliant. The function “img_proc” shown in section 5.3.2 needs to be called for every new frame from the capture hardware. This is implemented by calling it from the main task “tskmainFunc” that has been set up. The code discussed in this section is intended for merely performing an initial evaluation and testing of the image processing algorithm. The code that implements this is shown below.

```
/*********************************************************************************/
/*          Copyright (C) 2000 Texas Instruments Incorporated.                  */
/*                           All Rights Reserved                                */
/*------------------------------------------------------------------------------*/
/* FILENAME...... main.c                                                        */
/*                                                                              */
/* Function:                                                                    */
/*          This file is intended to serve as a generic template file for eval– */
/* uating various imaging applications, on the Imaging TDK hardware. This code  */
/* was initially written to demonstrate a straight pass–through demo for gray   */
/* scale and color images.                                                      */
/*                                                                              */
/* There are four main functions:                                               */
/*                                                                              */
/* a) void main() :                                                             */
/*                    The main function initializes Chip Support Library. It opens */
/* a DMA channel for 2D transfers using the DAT_open call. It also resets the cap– */
/* ture and display drivers.                                                    */
/*                                                                              */
/* b) void tskMainFunc:                                                         */
/*                      This is a task that is scheduled in BIOS to run using the  */
/*    task manager. In this example two functions are placed within this task.  */
/*    These are the grayscale() function and color() function.                  */
/*                                                                              */
/* c) void grayscale(): A function that displays captured image as gray scale   */
/*    image.                                                                    */
/*                                                                              */
/* d) void color(): A function that displays captured image in color as a packed */
/*    RGB in 565 format                                                         */
/*                                                                              */
/* NOTES:                                                                       */
/*                                                                              */
/* Make sure that DSP/BIOS HWI is configured for the video interrupts.          */
/* HWI5 –> _VCAP_isr                                                            */
/* HWI6 –> _VDIS_isr                                                            */
/* Make sure ”Use Dispatcher” is checked for each one.                          */
/*                                                                              */
/*********************************************************************************/

/*------------------------------------------------------------------------------*/
/* DSP/BIOS includes hedaer files                                               */
/*------------------------------------------------------------------------------*/
```

```
#include <std.h>
#include <log.h>
#include <swi.h>
#include <sem.h>
#include <clk.h>

/*-----------------------------------------------------------------------------------*/
/* CSL includes                                                                      */
/*-----------------------------------------------------------------------------------*/

#include <csl.h>
#include <irq.h>
#include <dat.h>
#include <cache.h>

/*-----------------------------------------------------------------------------------*/
/* Capture/Display Hardware Application includes                                     */
/*-----------------------------------------------------------------------------------*/

#include ”vcap.h”
#include ”vdis.h”

/*-----------------------------------------------------------------------------------*/
/* ImageLIB component colour space conversion header file includes                   */
/*-----------------------------------------------------------------------------------*/

#include ”ycbcr422pl_to_rgb565_h.h”
#include ”img_proc.h”
#include ”lpf_params.h”
#include ”thresh_params.h”

/*-----------------------------------------------------------------------------------*/
/* VCAP_Frame is the capture hardware structure that contains pointers to luma/chr */
/* oma. output array is a pointer that will be used to write output results for    */
/* display.                                                                          */
/*-----------------------------------------------------------------------------------*/

static VCAP_Frame *input;
static void *output;

/*-----------------------------------------------------------------------------------*/
/* The following arrays contain Y, Cr and Cb info and are in external memory as    */
/* the .data section is defined to be in external memory in the .cdb file. They    */
/* are declared to be aligned on a 128 byte boundary.                                */
/*                                                                                   */
/* ybuff: contains luma data of size 640 by 480                                      */
/* crbuff: chroma red data   of size 320 by 480                                      */
/* cbbuff: chnroma blue data of size 320 by 480                                      */
/*                                                                                   */
/*-----------------------------------------------------------------------------------*/

#pragma DATA_ALIGN(ybuff,128);
#pragma DATA_ALIGN(crbuff,128);
#pragma DATA_ALIGN(cbbuff,128);
static char ybuff[640*480];
static char crbuff[320*480];
static char cbbuff[320*480];
#pragma DATA_SECTION(ext_mem,”.image:ext_sect”);
#pragma DATA_SECTION(int_mem,”.chip_image:int_sect”);
#pragma DATA_ALIGN(ext_mem, 8);
#pragma DATA_ALIGN(int_mem, 8);
char ext_mem[246 * 320];
char int_mem[16  * 240];
```

```
/*---------------------------------------------------------------------------------*/
/* Function declarations for greyscale routines                                    */
/*---------------------------------------------------------------------------------*/

static void greyscale();

/*----------------------------------------------------------------------------*/
/* Main function                                                              */
/*----------------------------------------------------------------------------*/

void main()
{

  /*--------------------------------------------------------------------------*/
  /*  Initialize CSL and open the next available DMA channel for data transfer*/
  /*--------------------------------------------------------------------------*/

  CSL_Init();
  DAT_Open(DAT_CHAANY, DAT_PRI_LOW, DAT_OPEN_2D);

  CACHE_Flush(CACHE_L2ALL, 0x0, 0x0);

  /*--------------------------------------------------------------------------*/
  /* Reset the display and capture hardware by calls to vdis and vcap         */
  /*--------------------------------------------------------------------------*/

  VDIS_config(VDIS_RESET);
  VCAP_config(VCAP_RESET);
}

/*----------------------------------------------------------------------------*/
/* Algorithm to be executed is setup as a task under BIOS which gets scheduled*/
/* by the TASK Manager.                                                      */
/*----------------------------------------------------------------------------*/

void tskMainFunc()
{

     /*-----------------------------------------------------------------------*/
     /* Perform greyscale algorithm followed by color algorithm               */
     /*-----------------------------------------------------------------------*/

     greyscale();

}

/*----------------------------------------------------------------------------*/
/* This is the greyscale algorithm where the luma channel alone is made use of*/
/*----------------------------------------------------------------------------*/

void greyscale()
{
    int hres;
    int vres;
    int frame_cnt;

    IMAGE in_image;
    IMAGE out_image;
    SCRATCH_PAD scratch_pad;
    LPF_PARAMS  lpf_params;
    THRESH_PARAMS thresh_params;

    /*------------------------------------------------------------------------*/
    /* Configure capture hardware for square pixel and siplay hardware for    */
    /* gray scale images.                                                     */
    /*------------------------------------------------------------------------*/
```

```
VCAP_config(VCAP_SQP);
VDIS_config(VDIS_640X480_GS);

/*-------------------------------------------------------------------------*/
/* Obtain the default horizontal resolution, vertical resolution, bits    */
/* per pixel and pitch settings for gray scale mode                        */
/*-------------------------------------------------------------------------*/

hres  = VDIS_settings.hres;
vres  = VDIS_settings.vres;

/*------------------------------------------------------------------------*/
/* The following loop, iterates  for a set number of frames (300) and    */
/* implements the gray scale display.                                     */
/*------------------------------------------------------------------------*/

for(frame_cnt = 0; frame_cnt < 24 * 60 * 60; frame_cnt ++ )
{
    /*--------------------------------------------------------------------*/
    /* The current set of input and output pointers are obtained by    */
    /* calls to the getFrame and toggleBuffs routine. By using the     */
    /* SYS_FOREVER flag the function blocks until a new frame arrives  */
    /* By setting the output side argument to 0, the next available    */
    /* buffer is returned independent of the display event.            */
    /* ”input” and ”output” are pointers to input frame and output     */
    /* data.                                                           */
    /*--------------------------------------------------------------------*/

    input  = VCAP_getFrame(SYS_FOREVER);
    output = VDIS_toggleBuffs(0);

    in_image.img_data = input–>y1;
    in_image.img_cols = hres;
    in_image.img_rows = vres/2;

    out_image.img_data   = output;
    out_image.img_cols   = hres;
    out_image.img_rows   = vres;

    /*--------------------------------------------------------------------*/
    /* Two types of scratch pad memory are used, external and internal */
    /* memory. To pass this information to the application the scratch */
    /* pad structure is set with the addresses of external and internal*/
    /* memory and their corresponding sizes.                           */
    /*--------------------------------------------------------------------*/

    scratch_pad.ext_data = ext_mem;
    scratch_pad.ext_size = sizeof(ext_mem);
    scratch_pad.int_data = int_mem;
    scratch_pad.int_size = sizeof(int_mem);

    /*--------------------------------------------------------------------*/
    /* The following parameters are needed by the low pass filter, mask*/
    /* of filter coefficients and the shift value to be used.          */
    /*--------------------------------------------------------------------*/

    lpf_params.mask   = (unsigned char *) lpf_ext;
    lpf_params.shift  = shift_ext;

    /*-------------------------------------------------------------------*/
    /* The thresholding algorithm needs the threshold value as input  */
    /*-------------------------------------------------------------------*/

    thresh_params.threshold = thresh_ext;
```

```
        /*---------------------------------------------------------------*/
        /* Call Image Processing demo here. This might break out into 4   */
        /* seperate calls to each of the algorithms once it is DAISIZED.  */
        /* The input image, output image, scratchpad, low pass parameters */
        /* and threshold values are passed as arguments                   */
        /*---------------------------------------------------------------*/

        img_proc( &in_image, &out_image, &scratch_pad, FLDS, &lpf_params,
                  &thresh_params);

    }

    /*--------------------------------------------------------------------*/
    /* Reset the capture and display hardware.                            */
    /*--------------------------------------------------------------------*/

    VDIS_config(VDIS_RESET);
    VCAP_config(VCAP_RESET);
}
```

Notice that the grey_scale function gets scheduled by the BIOSII scheduler to run, whenever there is a new frame of input data to process. This in turn calls the img_proc algorithm that performs the four different image processing algorithms. This concludes the initial testing and validation of the image processsing algorithm.

5.5 Performance Considerations

The image processing demonstration using ImageLIB was developed to illustrate the power of the combined use of several key software and hardware that TI provides to accelerate image processing applications. The principle behind these software libraries was ease of use and shorter development cycles to initiate rapid prototyping on the TMS320C6x architecture. The performance table presented below has four columns of data:

1) Raw ImageLIB performance

2) Expected Performance due to ImageLIB + Image Data Manager functions

3) Actual Measured Performance

4) Performance Percentage

The cycle counts measured for performance on silicon was the number of cycles taken to execute the entire algorithm on the whole image. The expected performance does not include the additional cycles for opening streams and the set-up code for the algorithm. The expected number of cycles is defined as the number of cycles that are required for data transfer and performing the algorithm on the whole image once the streams have all been initialized.

The performance percentage is close to the theoretical estimates when the processing time is a significant percentage of the algorithm as compared to the data transfer cycles. When this is not true, the data transfer cycles tend to predominate. For example in the case of the median_3x3 algorithm it takes nine cycles to produce one output pixel while in the case of the threshold algorithm it takes 0.56 cycles for one output pixel. In all, the image processing algorithms considered in this application, report the number of output pixels is the same as the number of input pixels. Therefore, there is an eighteen-fold difference in processing between some algorithms. To hide the short processing times of some of the algorithms, multiple output lines are processed together to alleviate performance; for example, the threshold algorithm works to produce six output lines at a time, while the convolution algorithm produces one output line at a time.

The following performance figures are for the image processing demo algorithms that used an image size of [320 x 240]. In estimating the theoretical performance in column four for these algorithms, 360 cycles of overhead were added fro each DSTR_get and DSTR_put call. This overhead for the Image Data Manager routines was obtained by actual measurement. The second column contains the ImageLIB cycle count formula documented in the source and header files in the library. Although the median function is not part of the different tasks in the image processing demo, the performance for this ImageLIB kernel was also benchmarked to illustrate the need for balancing data transfer bandwith to computational bandwidth.

Table 5–1. Comparison of Performance Obtained with Theoretical Performance

Algorithm	ImageLIB Formula	ImageLIB	Expected Cycles	Measured Cycles	Performance (Expected) %/ Measured %
Conv_3x3	(4.5 * cols + 91) * rows	367440	488480	605451	81%
Median	(9 * cols +55) rows	70440	790800	910805	86%
Sobel	(3 * cols + 34) *(rows –2)	238560	315360	461372	69%
Threshold	(0.565 *cols + 24)	49152	58752	111255	53%

Consider the threshold example:

The raw performance for just the algorithm for a 320 by 240 image:

(0.565 * 320 + 24) * 240 = 49152 cycles

If only 1 line of the output image were to be processed for every iteration of the loop, 240 DSTR_get and DSTR_put calls would be needed. The additional cycles required for this would be:

(240 * 360) = 86400 cycles.

Clearly a significant portion of the cycles is consumed in data transfer and not in the processing itself. In this case the total number of expected cycles would be:

86400 cycles (data transfer) + 49152 cycles (processing) = 135552 cycles.

The actual measured performance is significantly better than this because six lines are processed at a time, therefore the raw performance in this scenario is:

(0.565 * 320 * 6 + 24) * 40 = 44352 cycles (smaller processing time)

(360 * 240) /6 = 14400 cycles (data transfer)

Therefore the total cycles to be expected is:

44352 cycles (processing) + 14400 cycles (data transfer) = 58752 cycles

The performance data needs to be studied with the following caveats in mind:

- The image processing and wavelet demos were put together using Image Data Manager libraries in conjunction with CSL to serve as a rapid prototyping platform capable of achieving real-time performance with the shortest development time possible.
- The image processing demo does not pipeline individual algorithms as is typical in an end application, and issues separate data transfer requests for the same input data used by several algorithms.
- The Image Data Manager libraries can be optimized further to take advantage of the DMA/EDMA routines of CSL rather than the DAT routines that are currently used.
- The additional cycles required for stream initialization and control code before the processing loop were not added to the estimated cycle count.
- Even with all these caveats and limitations, an average performance of 72 % was obtained on six complete algorithms, which is an impressive level of performance for a rapid prototyping platform.

5.6 Conclusions

This chapter examined the use of ImageLIB to perform various common image processing tasks. The programmer is advised to consult the user's guide to get a complete reference of the available image processing functions under ImageLIB. A complete image processing demonstration was then created, using three kernels that are available in ImageLIB. The neessary data flow for these kernels was provided through IDM. This application required an image size, which required the modifcation of the generic application drivers, provided in Chapter 3. This chapter also showed how simple modifications to the generic application drivers can achieve the required objective. In addition, it examined putting all these constituent pieces together, and perfoming initial evaluation and testing of the algorithm. The validated algorithm will be converted to be eXpressDSP-compliant in the next chapter. The integration of this eXpressDSP-compliant algorithm into the Channel Manager framework will be discussed in Chapter 7.

Chapter 6

Integration of an Application into the Imaging Framework

We have discussed how to create a standalone algorithm using the IDK hardware in the previous chapters. While a standalone algorithm can be very useful for prototyping, applications usually require an application framework. In DSP systems, the application framework is typically a series of software modules that manage data and algorithms. Software frameworks allow a multi-tasking environment with different priorities for different tasks. The software framework allows modular addition and deletion of different algorithms in the application without reworking the entire system. A software framework, the Channel Manager (CM) is provided by Texas Instruments (TI) that allows a multi-tasking environment. In this chapter, we will integrate the standalone example from the prior chapters into the Channel Manager (CM) software framework provided by TI.

Topic **Page**

6.1 Overview

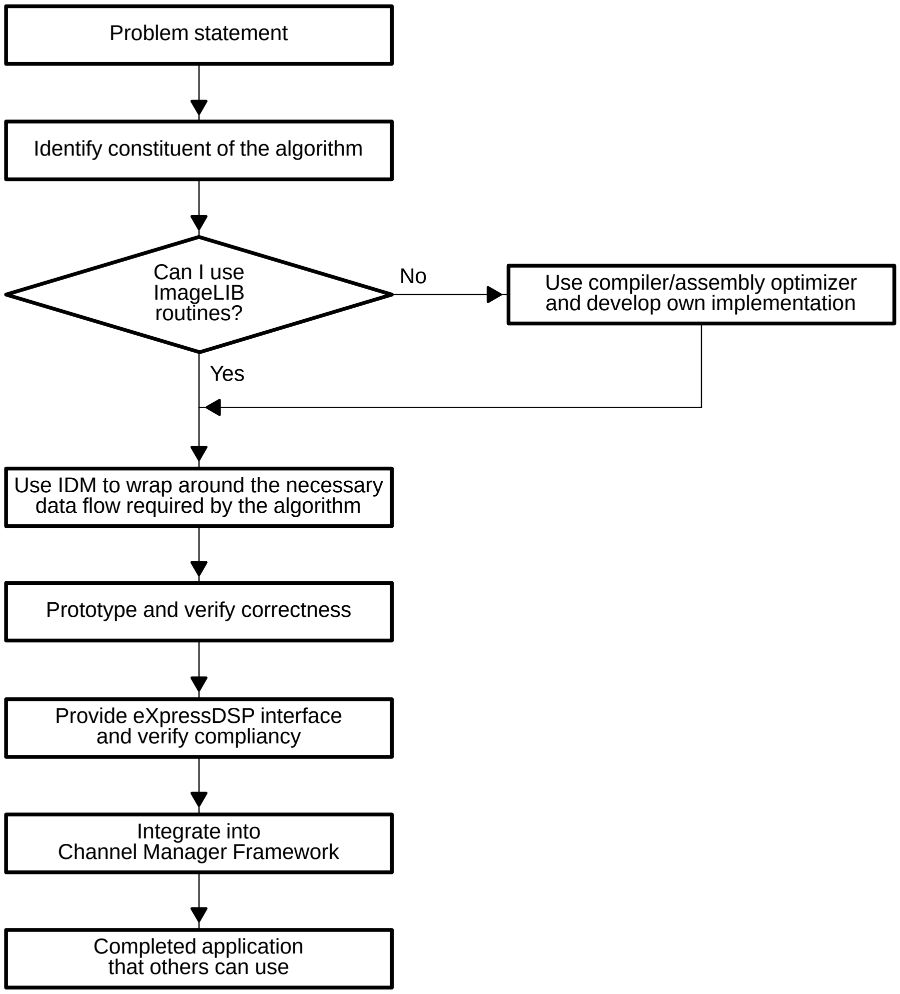
The recommended algorithm flow for the IDK is shown in Figure 6–1. The prevoius chapters have developed the procedure to prototype and verify correctness of the algorithm. This chapter will discuss the remaining steps:

- ❑ Providing eXpressDSP Interface and Verify Compliancy
- ❑ Integrate into Channel Manager Framework

The steps for completing these last two tasks are:

1) Verify eXpressDSP compliance of rules and guidelines
2) Create eXpressDSP Interfaces
3) Create main function for multi-tasking environment
4) Invoke CM function calls to implement algorithm

Figure 6–1. Recommended Algorithm Development Flow for IDK

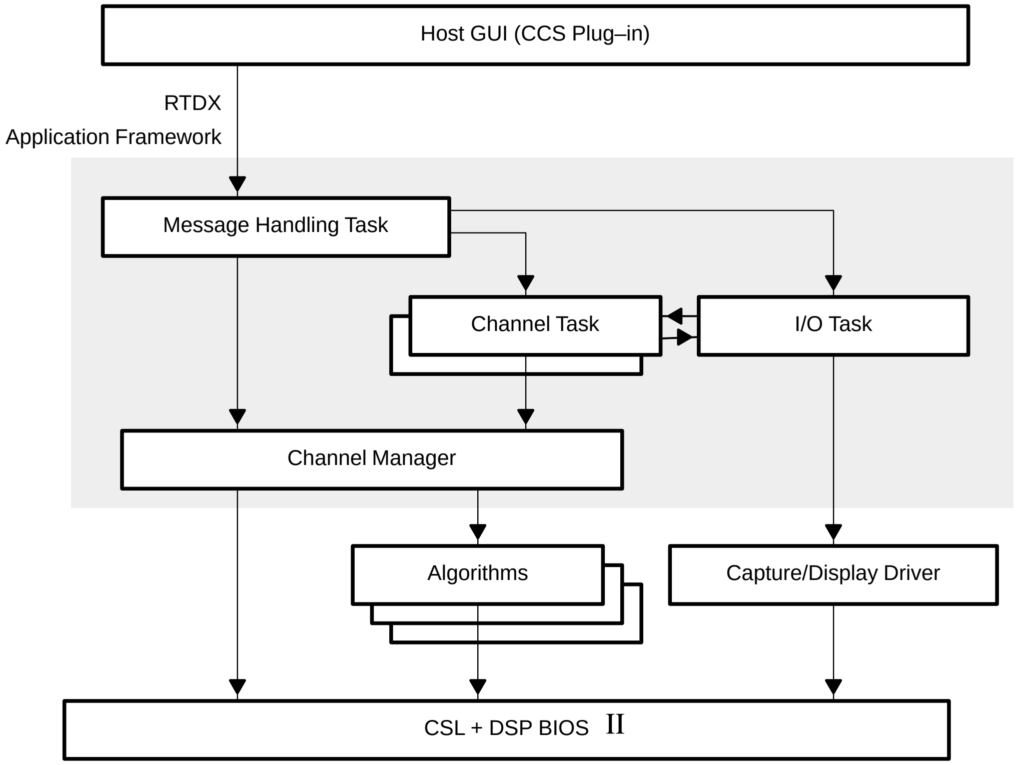

Problem statement
Identify constituent of the algorithm
Can I use ImageLIB routines?
No
Use compiler/assembly optimizer and develop own implementation
Yes
Use IDM to wrap around the necessary data flow required by the algorithm
Prototype and verify correctness
Provide eXpressDSP interface and verify compliancy
Integrate into Channel Manager Framework
Completed application that others can use

6.2 Channel Manager (CM) Overview

Before we integrate an algorithm into the software framework, it is useful to review the CM algorithm framework. Specific detail of the CM Application framework is available in the C6000 Imaging Development Kit Application Framework Application. The CM algorithm consists of three types of tasks as shown in Figure 6–2:

- ❑ I/O tasks for synchronization between algorithms and data movement for algorithms.
- ❑ Channel tasks provide an instance of each channel object for algorithm execution
- ❑ Message-handling tasks allow communication between the Host GUI of CCS to the target DSP.

Figure 6–2. IDK Channel Manager Framework



These tasks provide abstraction to access to the algorithms, capture and display drivers and the hardware of the DSP through the use of the Chip Support Library (CSL) and the DSP/BIOS kernal. The CM framework allows platform agnostic application development. To migrate between devices, the underlying pieces of the CSL and device specific drivers must be altered, but not the framework. In the example code provided, I/O tasks and Channel tasks are implemented for the framework. Additional information about message-handling tasks is available in the C6000 Imaging TDK Application Framework application note although they are not implemented in this example for simplicity of the algorithm development. The CM framework uses semaphores and the tasks described to schedule the application including the algorithm execution and data capture and display. The main application file will be changed to accomadate the methodology shift from a roundrobin approach used in the standalone example to the priority based scheme used by the CM algorithm framework.

To use the Channel Manager software framework, three requirements must be met. According to the C6000 Imaging TDK Application Framework application note, the requirements to integrate an algorithm into the channel manager are:

1) The algorithm works on a C6711 DSK (standalone example)

2) The algorithm is eXpressDSP-compliant (implements the IALG interface and observe all rules required by the eXpress DSP Algorithm Standard)

3) The algorithm provides the Channel Manager with a function pointer that points to its processing function of the form, void* XXXApply(IALG_Handle handle, void* in, void* out)

The starting point for algorithm integration is that the algorithm operates as a standalone system as described in the earlier chapters of the IDK Programmer’s Guide. Fulfilling the second two requirements will be described in detail using the Image Processing demonstration example used in previous chapters of the IDK Programmer’s Guide.

6.3 Verify eXpressDSP Compliance of Rules and Guidelines

The algorithm must be eXpressDSP-compliant to be used in the channel manager software framework. The software programming guidelines for eXpressDSP compliance is covered in great detail in other documents including:

- *TMS320 DSP Algorithm Standard API Reference* (Literature number SPRU360A)
- *TMS320 DSP Algorithm Standard* (Literature number SPRA581A)
- *TMS320 DSP Algorithm Developer's Guide* (Literature number SPRU424)
- *TMS320 DSP Algorithm Standard Rules and Guidelines* (Literature number SPRU352)
- *Using the TMS320 DSP Algorithm Standard in a Static DSP System* (Literature number SPRU577A)
- *Using the TMS320 DSP Algorithm Standard in a Dynamic DSP System* (Literature number SPRU580A)
- *Making DSP Algorithms Compliant with the TMS320 DSP Algorithm Standard* (Literature number SPRA579A)

For the purpose of this document, it is assumed that the algorithm is compliant with the eXpress DSP Algorithm Standard Rules and Guidelines. The eXpressDSP interfaces can be generated using the eXpressDSP template generator tool provided in Code Composer Studio™ (CCS). This tool generates the different files required for providing the eXpressDSP wrapper functions. The standalone image processing algorithms used in the previous chapters is compliant with the algorithm standard. Once compliance of the algorithm is confirmed, the interfaces can be created using the eXpressDSP template generator tool in CCS.

6.4 Create eXpressDSP Interfaces

The steps of creating the eXpressDSP interfaces are shown using the standalone convolution 3x3 algorithm used in the image processing example. The eXpressDSP interfaces are created using a eXpressDSP Template Tool provided in TI's integrated development environment, Code Composer Studio (CCS). The tool is used to generate the outline of code required for the interfaces. The files are updated with algorithm specific code. A step by step guide is detailed in the *TMS320 DSP Algorithm Developer's Guide* (literature number SPRU424). Refer to the application note for additional detail. This example is not meant to replace eXpress DSP Algorithm Standard documents, but is designed to build an example application as a guide. The image processing example used here is built using TI C6000 Code Composer Studio v2.

6.4.1 eXpressDSP Template Tool

The first step in creating the eXpressDSP interfaces is to use the eXpressDSP Template Tool. The tool can be accessed from the CCS v2 toolbar by selecting Tools –> Algorithm Standard –> Template Code Generator. The inputs to the tool are determined by examining the algorithm. For more information on creating the eXpressDSP interfaces, refer to *TMS320 DSP Algorithm Developer's Guide* (Literature number SPRU424).

For the example algorithm of CONV3X3 the inputs to the eXpressDSP Template Tool are:

Table 6–1. User Inputs to eXpressDSP Template Code Generator

eXpressDSP Tool Field	Example Input	Description
Algorithm Name	CONV3X3	The naming convention of eXpressDSP requires that the algorithm name be in all capital letters.
Vendor	TI	
Project locations	location newly generated files placed (usually specifc for each algorithm)	
Instance Creation Parameters	XDAS_Int32 pitch XDAS_Int8* lpf_coeffs XDAS_Int32 shift	These parameters are initialized at create time. When the algorithm is created space is reserved for these parameters.

Table 6–1. *User Inputs to eXpressDSP Template Code Generator (Continued)*

eXpressDSP Tool Field	Example Input	Description
Status Creation Parameters	XDAS_Int32 pitch XDAS_Int8* lpf_coeffs XDAS_Int32 shift	These parameters can be read and written to during execution
Algorithm Methods	XDAS_Int32 apply(XDAS_Int8* in, XDAS_Int8* out) XDAS_Bool control(ICONV3X3_Cmd cmd, ICONV3X3_Status *status)	The application level interfaces for algorithms include the apply method and a control method to allow integration into the CM framework

Check all three boxes for Generate These files:

❑ Algorithm Interface Files(I*.[ch] files)

❑ API Library Stubs (module [ch] files)

❑ Algorithm Implementation Stubs (module_vendor*.* files)

When the inputs are complete, select the Build files button.

The tool creates the following nine files in the project locations directory:

Table 6–2. *List of Files Generated by eXpressDSP Template Code Generator*

File	Description	Modified
(a) Application Framework		
conv3x3.c	implementation of API concrete functions	no
conv3x3.h	API concrete interface definitions	no
(b) Module Specific Interface		
iconv3x3.c	definition of default parameter structure settings	yes
iconv3x3.h	abstract interface definition header	yes
(c) Algorithm Specific Interface		
conv3x3_ti.c	client specific algorithm functions	yes
conv3x3_ti.h	client specific implementation header file used in application	no
conv3x3_ti_vtab.c	function v-table definitions (properities of each instance)	yes
CONV3X3.i	lists the inputs to the eXpressDSP Template tool	no
Project.mak		no

6.4.2 Modify eXpressDSP Template Tool Output Files

The second step to create the eXpressDSP interfaces is to modify the output files for the specific algorithm.

iconv3x3.c and iconv3x3.h

The module specific interface files, iconv3x3.c and iconv3x3.h files must be modified. The iconv3x3.c file is modified to set the default values for parameters for the algorithm objects. If this file is not modified the default values for all parameters will be zero. In this example, the low pass filter coefficents are modified and the shfit and pitch parameters are set as shown below.

```
 /*
 *  ======== iconv3x3.c ========
 *  This file defines the default parameter structure for iconv3x3.h
 */
#include <std.h>
#include <iconv3x3.h>
/*
 *  ======== conv3x3_PARAMS ========
 *  This constant structure defines the default parameters for conv3x3 objects
 */
char lpf_ext[3][9] = {
  { 1,   2,  1,  2, 4,  2,  1,  2,  1},
  { –1, –1, –1, –1, 8, –1, –1, –1, –1},
  { –1, –1, –1, –1, 9, –1, –1, –1, –1},
};
#define shift_ext1  4
#define shift_ext2  0
#define shift_ext3  0
const int    shift_ext[3] = {shift_ext1, shift_ext2, shift_ext3};
ICONV3X3_Params ICONV3X3_PARAMS = {
    sizeof(ICONV3X3_Params),
    640,                 /* int pitch; */
    (char *)lpf_ext[0],  /* char* lpf_coeffs; */
    shift_ext1           /* int shift; */
};
```

The iconv3x3.h file is modified include the xdas.h file that provides the xdas data types for the processor that were used as inputs to the eXpressDSP template generator tool. The updated file is shown below.

```
/*
 *  ======== iconv3x3.h ========
 *  ICONV3X3 Interface Header
 */
#ifndef ICONV3X3_
#define ICONV3X3_

#include <ialg.h>
#include <xdas.h>


/*
 *  ======== ICONV3X3_Handle ========
 *  This handle is used to reference all CONV3X3 instance objects
 */
typedef struct ICONV3X3_Obj *ICONV3X3_Handle;

/*
 *  ======== ICONV3X3_Obj ========
 *  This structure must be the first field of all CONV3X3 instance objects
 */
typedef struct ICONV3X3_Obj {
    struct ICONV3X3_Fxns *fxns;
} ICONV3X3_Obj;

/*
 *  ======== ICONV3X3_Status ========
 *  Status structure defines the parameters that can be changed or read
 *  during real-time operation of the algorithm.
 */
typedef struct ICONV3X3_Status {
    Int size;   /* must be first field of all status structures */
    XDAS_Int32 pitch;
    XDAS_Int8* lpf_coeffs;
    XDAS_Int32 shift;

} ICONV3X3_Status;
/*
 *  ======== ICONV3X3_Cmd ========
 *  The Cmd enumeration defines the control commands for the CONV3X3
 *  control method.
 */
typedef enum ICONV3X3_Cmd {
        ICONV3X3_GETSTATUS,
        ICONV3X3_SETSTATUS
} ICONV3X3_Cmd;

/*
 *  ======== ICONV3X3_Params ========
 *  This structure defines the creation parameters for all CONV3X3 objects
 */
typedef struct ICONV3X3_Params {
    Int size;   /* must be first field of all params structures */
    XDAS_Int32 pitch;
    XDAS_Int8* lpf_coeffs;
    XDAS_Int32 shift;

} ICONV3X3_Params;

/*
 *  ======== ICONV3X3_PARAMS ========
 *  Default parameter values for CONV3X3 instance objects
 */
extern ICONV3X3_Params ICONV3X3_PARAMS;

/*
 *  ======== ICONV3X3_Fxns ========
 *  This structure defines all of the operations on CONV3X3 objects
 */
typedef struct ICONV3X3_Fxns {
    IALG_Fxns   ialg;    /* ICONV3X3 extends IALG */
    XDAS_Bool   (*control)(ICONV3X3_Handle handle, ICONV3X3_Cmd cmd,
ICONV3X3_Status *status);
    XDAS_Int32    (*apply)(ICONV3X3_Handle handle,  XDAS_Int8* in,
XDAS_Int8* out);

} ICONV3X3_Fxns;

#endif  /* ICONV3X3_ */
```

conv3x3_ti_vtab.c

The file conv3x3_ti_vtab.c is renamed conv3x3_TI_ialgvt.c.

conv3x3_ti.c

The third file that must be modified is the conv3x3_ti.c file. The tool creates the outline of the implementation of algorithm functions for the eXpressDSP interfaces, but the algorithm specific information must be added.

In the conv3x3 algorithm example, the changes to the conv3x3_ti.c file as shown in the sample code are:

❑ The file needs to be renamed conv3x3_TI_ialg.c.

❑ Add algorithm specfic header files.

❑ Add the object data types required by the algorithm to the CONV3X3_TI_Obj structure.

❑ Enter the memory requirements for the algorithm in the CONV3X3_TI_alloc() function.

❑ Update CONV3X3_TI_free() function to include additional memory specified in CONV3X3_TI_alloc() function.

- ❑ Add calls to the conv3x3 algorithm initialization routine to the CONV3X3_TI_initObj() function.
- ❑ Update CONV3X3_TI_control() function for algorithm specific implementation.
- ❑ Update CONV3X3_TI_apply() function for algorithm specific implementation.

In this conv3x3 example, the following functions are not implemented:

- ❑ CONV3X3_TI_activate
- ❑ CONV3X3_TI_deactivate
- ❑ CONV3X3_TI_exit
- ❑ CONV3X3_TI_init
- ❑ CONV3X3_TI_moved

```
/*
 *  ======== Conv3x3_ti.c ========
 *  Implementation of the CONV3X3_TI.h interface; TI’s implementation
 *  of the ICONV3X3 interface.
 */
#pragma CODE_SECTION(CONV3X3_TI_alloc, ”.text:algAlloc”)
#pragma CODE_SECTION(CONV3X3_TI_free, ”.text:algFree”)
#pragma CODE_SECTION(CONV3X3_TI_initObj, ”.text:algInit”)
#pragma CODE_SECTION(CONV3X3_TI_init, ”.text:init”)
#pragma CODE_SECTION(CONV3X3_TI_exit, ”.text:exit”)

#include <std.h>
#include <conv3x3.h>
#include <iconv3x3.h>

#include <conv3x3_ti.h>

#include ”img_proc.h”
#include ”conv3x3_image.h”

/*
 *  ======== CONV3X3_TI_Obj ========
 */
typedef struct CONV3X3_TI_Obj {
    IALG_Obj    alg;          /* MUST be first field of all CONV3X3 objs
*/
    /*! TODO: add custom fields here !*/
    SCRATCH_PAD scratch_pad;
    unsigned char* mask;
    int shift;
    int pitch;
    void *base[2];
} CONV3X3_TI_Obj;

/*
 *  ======== CONV3X3_TI_activate ========
 *  Activate our object; e.g., initialize any scratch memory required
 *  by the CONV3X3_TI processing methods.
 */
Void CONV3X3_TI_activate(IALG_Handle handle)
{
    CONV3X3_TI_Obj *conv3x3 = (Void *)handle;

    /*! TODO: implement algActivate !*/
}

/*
 *  ======== CONV3X3_TI_alloc ========
 *  Return a table of memory descriptors that describe the memory needed
 *  to construct a CONV3X3_TI_Obj structure.
 */
Int CONV3X3_TI_alloc(const IALG_Params *conv3x3Params, IALG_Fxns **fxns,
IALG_MemRec memTab[])
{
    const ICONV3X3_Params *params = (Void *)conv3x3Params;

    /*! TODO: implement algAlloc !*/

    if (params == NULL) {
        params = &ICONV3X3_PARAMS;    /* set default parameters */
    }

    /* Request memory for CONV3X3 object */
    memTab[0].size = sizeof(CONV3X3_TI_Obj);
    memTab[0].alignment = 0;
    memTab[0].space = IALG_EXTERNAL;
    memTab[0].attrs = IALG_PERSIST;

    memTab[1].size = (16  * 240);
    memTab[1].alignment = 4;
    memTab[1].space = IALG_DARAM0;
    memTab[1].attrs = IALG_SCRATCH;

    return (2);
}

/*
 *  ======== CONV3X3_TI_deactivate ========
 *  Deactivate our object; e.g., save any scratch memory required
 *  by the CONV3X3_TI processing methods to persistent memory.
 */
Void CONV3X3_TI_deactivate(IALG_Handle handle)
{
    CONV3X3_TI_Obj *conv3x3 = (Void *)handle;
    /*! TODO: implement algDeactivate !*/
}
```

```
/*
 *  ======== CONV3X3_TI_exit ========
 *  Exit the CONV3X3_TI module as a whole.
 */
Void CONV3X3_TI_exit(Void)
{
    /*! TODO: implement module exit !*/
}

/*
 *  ======== CONV3X3_TI_free ========
 *  Return a table of memory pointers that should be freed.  Note
 *  that this should include *all* memory requested in the
 *  CONV3X3_TI_alloc operation above.
 */
Int CONV3X3_TI_free(IALG_Handle handle, IALG_MemRec memTab[])
{
    Int n;
    CONV3X3_TI_Obj *conv3x3 = (Void *)handle;

    /*! TODO: implement algFree !*/

    n = CONV3X3_TI_alloc(NULL, NULL, memTab);
    memTab[0].base = conv3x3->base[0];
    memTab[1].base = conv3x3->base[1];

    return (n);
}

/*
 *  ======== CONV3X3_TI_init ========
 *  Initialize the CONV3X3_TI module as a whole.
 */
Void CONV3X3_TI_init(Void)
{
    /*! TODO: implement module init !*/
}

/*  ======== CONV3X3_TI_initObj ========
 *  Initialize the memory allocated for our instance.
 */
Int CONV3X3_TI_initObj(IALG_Handle handle,
              const IALG_MemRec memTab[], IALG_Handle p, const IALG_Params
*conv3x3Params)
{
    CONV3X3_TI_Obj *conv3x3 = (Void *)handle;
    const ICONV3X3_Params *params = (Void *)conv3x3Params;

    /*! TODO: implement algInit !*/
    if (params == NULL) {
      params = &ICONV3X3_PARAMS;        /* set default parameters */
    }
    conv3x3->scratch_pad.ext_size = (320*246);
    conv3x3->scratch_pad.int_data = memTab[1].base;
    conv3x3->scratch_pad.int_size = (16*240);
    conv3x3->pitch = params->pitch;
    conv3x3->mask = params->lpf_coeffs;
    conv3x3->shift = params->shift;
    conv3x3->base[0] = memTab[0].base;
    conv3x3->base[1] = memTab[1].base;
    return (IALG_EOK);
}

/*  ======== CONV3X3_TI_moved ========
 *  Fix up any pointers to data that has been moved by the client.
 */
Void CONV3X3_TI_moved(IALG_Handle handle,
              const IALG_MemRec memTab[], IALG_Handle p, const IALG_Params
*conv3x3Params)
{
    CONV3X3_TI_Obj *conv3x3 = (Void *)handle;
    const ICONV3X3_Params *params = (Void *)conv3x3Params;

    /*! TODO: implement algMoved !*/
    conv3x3->scratch_pad.int_data = memTab[1].base;
}

/*
 *  ======== CONV3X3_TI_control ========
 *  TI’s implementation of the control operation.
 */
XDAS_Bool   CONV3X3_TI_control(ICONV3X3_Handle handle, ICONV3X3_Cmd cmd,
ICONV3X3_Status *status)
{
    CONV3X3_TI_Obj *conv3x3 = (Void *)handle;

        /*! TODO: implement control !*/
        if(cmd == ICONV3X3_GETSTATUS) {
          status->pitch = conv3x3->pitch;
          status->lpf_coeffs = conv3x3->mask;
          status->shift = conv3x3->shift;
    }
    else if(cmd == ICONV3X3_SETSTATUS) {
      conv3x3->pitch = status->pitch;
          conv3x3->mask = status->lpf_coeffs;
          conv3x3->shift = status->shift;
    }

    return ((XDAS_Bool  )0);
}

/*
 *  ======== CONV3X3_TI_apply ========
 *  TI’s implementation of the apply operation.
 */
XDAS_Int32    CONV3X3_TI_apply(ICONV3X3_Handle handle,  XDAS_Int8* in,
XDAS_Int8* out)
{
    CONV3X3_TI_Obj *conv3x3 = (Void *)handle;
```

```
        /*! TODO: implement apply !*/
    IMAGE out_image;
    out_image.img_rows = 480;
    out_image.img_cols = 640;

        conv3x3–>scratch_pad.ext_data = in;
        /*! TODO: implement apply !*/
    Conv3x3_image(&conv3x3–>scratch_pad, &out_image, out, conv3x3–>pitch,
                  conv3x3–>mask, conv3x3–>shift);

    return ((XDAS_Int32   )0);
}
```

This process is repeated for the individual algorithms including sobel edge detection, threshold, pre-scale, grayscale. For the purpose of the document the convolution 3x3 is highlighted, but the remaining algorithms are available in the example with the eXpressDSP interfaces.

6.5 Create Main Function for Multi-Tasking Environment

Once the eXpressDSP interfaces of the algorithm are complete, the main function is created to use the channel manager software framework for the invocation of the algorithms. The main function makes use of semaphores and multiple tasks for scheduling of the different algorithms. The algorithms are invoked through the CM algorithm framework. The CM algorithm frameworks uses the eXpressDSP interfaces created earlier to access the algorithms created. The project will include the CM files and eXpressDSP interface source and header files, in addition to the original algorithm files used in the standalone system.

The structure of the main file is different because of a change in the definition of a channel when using the CM framework. The standalone example created a DSP/BIOS task that calls the image processing algorithm. This image processing algorithm calls the four different image processing algorithms (convolution, threshold, sobel edge detection, and pass-through) with 640x480 NTSC images. For the purpose of the standalone example, a channel is defined as a single input frame regardless of how many output frames are produced. The channel manager software framework uses a different definition of a channel. A channel is defined as any input to output frame path regardless of how many different algorithms access the input frame. This change in definition is made to allow more modular design of the software framework. In this example, four channels will be defined and used to accommodate the four different output images in the image processing demonstration.

The mainapp.c file is used to demonstrate the changes required when integrating an algorithm into the CM software framework. The structure of the mainapp.c file is that each algorithm is a different channel that must be processed.

Steps to Add a Channel of an Algorithm Using the CM Framework

In project .cdb file:

1) Create sem object for new channel.
2) Create sts object for channel.
3) Create tsk object for channel (mapped to function apppropriate function and priority (use other channels as examples).

In main file:

1) Add approciate header files.
2) Declare BIOS objects for additional channels.
3) Define handle for new channel object.

4) Define display offset for channel video output.

5) Define handles for channel object.

```
HANDLE hconv3x3
```

6) CM_Init() called once in the main file (if already present, do not add again).

```
CM_Init();
```

7) CM_Control called for initializing the internal and external heap pointers.

```
CM_Control(CM_SET_INTERNAL_HEAP, InternalHeap);
CM_Control(CM_SET_EXTERNAL_HEAP, ExternalHeap);
```

8) Register the channel with the CM framework.

   CM_Reg_Alg invoked to register each algorithm with the channel manager using the handles declared earlier.

```
hconv3x3 = CM_RegAlg(”Convolution 3 x 3”,
&Conv3x3_TI_IConv3x3,
(void (*)())Conv3x3_TI_IConv3x3.apply, NULL, 1, 1);
```

9) Open channel and configure to exectue in the CM framework.

   CM_Open declared to initialize each channel.

```
hCha[0] = CM_Open(””, NULL, NULL);
```

10) Set algorithms in the CM framework.

   CM_SetAlgs called to set the algorithms in each channel.

```
CM_SetAlgs(hCha[0], 1 , &hconv3x3);
```

11) Post channel semaphore to blockcapture and display buffers.

12) Execute channel object within the CM framework.

   CM_Exec invoked to execute each algorithm in the specified channel object.

```
CM_Exec(hCha[0], im, out, NULL, SIG_DEF);
```

The include files are similar to the standalone example, however the channel manager header file, cm.h is added. In addition, the header files for each of the algorithms are specified. The output quadrants are specified differently from the standalone example. Five channels are created in this example since the pre-scale algorithm, although not displayed directly is also called as a channel. By using semaphores, the input frame is not overwritten until each algorithm has processed the current frame. Additionally, an output semaphore is created to wait for all algorithms to complete for the display buffer. The quadrant of the monitor used for the output of each channel, sobel edge detection, convolution 3x3, threshold, and pass-through is determined by an individual output offset.

Objects created in the DSP/BIOS cdg file are declared as external global vairables. In this example, statistics are created in the DSP/BIOS cdb file to gather execution timings for the different algorithms.

```
/*--------------------------------------------------------------------------------*/
/* DSP/BIOS includes header files                                                */
/*--------------------------------------------------------------------------------*/
#include <std.h>
#include <log.h>
#include <swi.h>
#include <sem.h>
#include <clk.h>
#include <sts.h>
#include <tsk.h>
/*--------------------------------------------------------------------------------*/
/* CSL includes                                                                  */
/*--------------------------------------------------------------------------------*/
#include <csl.h>
#include <irq.h>
#include <dat.h>
#include <cache.h>
/* channel manager includes */
#include <cm.h>
/*--------------------------------------------------------------------------------*/
/* Algorithm header file includes                                                */
/*--------------------------------------------------------------------------------*/
#include ”grayscale_ti.h”
#include ”sobel_ti.h”
#include ”conv3x3_ti.h”
#include ”thold_ti.h”
#include ”gscopy_ti.h”
/*--------------------------------------------------------------------------------*/
/* Capture/Display Hardware Application includes                                 */
/*--------------------------------------------------------------------------------*/
#include ”vcap.h”
```

```
#include ”vdis.h”
/*--------------------------------------------------------------------------------*/
/* Function declarations for routines                                            */
/*--------------------------------------------------------------------------------*/
#define HEIGHT                       240
#define WIDTH                        320
#define CHA_CT                       4
#define MSG_FRAME_RATE               0x02
#define MSG_CHANNEL_PAUSE            0x03
#define MSG_CHANNEL_RESUME           0x04
#define MSG_CHANGE_FILTER_TYPE       0x06
#define MSG_CHANGE_THRESHOLD         0x07

#define MAX_RATE                     30
/*------------------------------------------------------------------------------*/
/* Data buffer definition                                                      */
/*------------------------------------------------------------------------------*/
#pragma DATA_ALIGN(ybuff,128);
unsigned char ybuff[(HEIGHT+6)*WIDTH];
/*------------------------------------------------------------------------------*/
/* Channel related variable definitions                                        */
/*------------------------------------------------------------------------------*/
HANDLE hCha[CHA_CT+1];
FRM_OBJ inputFrm;
FRM_OBJ *in[1];
FRM_OBJ outputFrm, *out[1];
FRM_OBJ imFrm[1], *im[1];
/*------------------------------------------------------------------------------*/
/* Output offset for each channel (upper left, upper right, lower left,       */
/* lower right.                                                                */
/*------------------------------------------------------------------------------*/
int output_offset[CHA_CT] =
{
    0,
    320,
    (640*240),
    (640*240)+320,
};
int output_size[CHA_CT][2] =
{
    {320,240},
    {320,240},
    {320,240},
    {320,240},
};
/*----------------------------------------------------------------------------*/
/* Channel task related globals variables.                                    */
/*----------------------------------------------------------------------------*/
STS_Obj* STS_ExeTime[CHA_CT];
TSK_Handle hTask[CHA_CT];
int TaskPri[CHA_CT];
SEM_Obj *SEM_Ch[CHA_CT];
int MaxRate = 30;
extern far volatile int CapRate;
/*----------------------------------------------------------------------------*/
/* Externally defined objects for example (defined in BIOS .cdb file.         */
/*----------------------------------------------------------------------------*/
extern STS_Obj STS_ExeTimeCha1, STS_ExeTimeCha2, STS_ExeTimeCha3;
extern STS_Obj STS_ExeTimeCha4;
extern int InternalHeap;
extern int ExternalHeap;
/*----------------------------------------------------------------------------*/
/* I/O Task variable definitions                                              */
/*----------------------------------------------------------------------------*/
volatile int Rate[CHA_CT];
int IncFlag[CHA_CT];
volatile void* output;
volatile void* prev_output;
volatile  VCAP_Frame *input;
SEM_Obj* semOutput;

/*----------------------------------------------------------------------------*/
/* Main function                                                              */
/*----------------------------------------------------------------------------*/
```

```
void main()
{
    int i;
    HANDLE hsobel, hconv3x3, hthold, hgrayscale, hcopy;

  /*----------------------------------------------------------------------------*/
  /* Create semaphore objects for multi–channel environment. Set initial count  */
  /* to zero for each object.                                                   */
  /*----------------------------------------------------------------------------*/
    for(i = 0; i<CHA_CT;i++)
    {
        SEM_Ch[i] = SEM_create(0,NULL);
    }
  /*----------------------------------------------------------------------------*/
  /* Initialize the I/O data buffer pointers for channels                       */
  /*----------------------------------------------------------------------------*/

    in[0] = &inputFrm;
    out[0]= &outputFrm;
    imFrm[0].Addr = ybuff;
    im[0] = &imFrm[0];
    for(i = 0; i<CHA_CT;i++)
    {
        IncFlag[i] = 0;
        TaskPause[i] = FALSE;
        Rate[i] = MaxRate;
    }
  /*----------------------------------------------------------------------------*/
  /* Create output semaphore object for display and capture buffer              */
  /*----------------------------------------------------------------------------*/
    semOutput = SEM_create(0,NULL);

    /*--------------------------------------------------------------------------*/
    /*  Initialize CSL and open the next available DMA channel for data transfer*/
    /*--------------------------------------------------------------------------*/
```

```
CSL_Init();
DAT_Open(DAT_CHAANY, DAT_PRI_LOW, DAT_OPEN_2D);

/*---------------------------------------------------------------------------*/
/*  Initialize Channel Manager application framework and setup internal and */
/*  external heap pointers.                                                */
/*---------------------------------------------------------------------------*/

CM_Init();

CM_Control(CM_SET_INTERNAL_HEAP, InternalHeap);
CM_Control(CM_SET_EXTERNAL_HEAP, ExternalHeap);
/*---------------------------------------------------------------------------*/
/*  Initialize pointers of DSP/BIOS statistics objects for channels         */
/*---------------------------------------------------------------------------*/

STS_ExeTime[0] = &STS_ExeTimeCha1;
STS_ExeTime[1] = &STS_ExeTimeCha2;
STS_ExeTime[2] = &STS_ExeTimeCha3;
STS_ExeTime[3] = &STS_ExeTimeCha4;
/*---------------------------------------------------------------------------*/
/*  Register algorithms with the channel manager application framework      */
/*---------------------------------------------------------------------------*/

hsobel = CM_RegAlg(”Sobel”, &SOBEL_TI_ISOBEL,
    (void (*)())SOBEL_TI_ISOBEL.apply, NULL, 1, 1);

hconv3x3 = CM_RegAlg(”Convolution 3 x 3”, &CONV3X3_TI_ICONV3X3,
    (void (*)())CONV3X3_TI_ICONV3X3.apply, NULL, 1, 1);

hthold = CM_RegAlg(”Threshold”, &THOLD_TI_ITHOLD,
    (void (*)())THOLD_TI_ITHOLD.apply, NULL, 1, 1);

hgrayscale = CM_RegAlg(”Pre–scale”, &GRAYSCALE_TI_IGRAYSCALE,
    (void (*)())GRAYSCALE_TI_IGRAYSCALE.apply, NULL, 1, 1);

hcopy = CM_RegAlg(”pass through”, &GSCOPY_TI_IGSCOPY,
    (void (*)())GSCOPY_TI_IGSCOPY.apply, NULL, 1, 1);

/*---------------------------------------------------------------------------*/
/*  Open channels for the CM application framework and set algorithms for   */
```

```
    /*  each channel.                                                     */
    /*  Channel 0 : 3x3 convolution                                       */
    /*  Channel 1 : sobel edge detection                                  */
    /*  Channel 2 : threshold                                             */
    /*  Channel 3 : image copy for pass–thru                              */
    /*  Channel 4 : image pre–scale                                       */
    /*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/

    for(i = 0; i < CHA_CT+1; i++)
    {
        hCha[i] = CM_Open(””, NULL, NULL);
     }

    CM_SetAlgs(hCha[0], 1 , &hconv3x3);
    CM_SetAlgs(hCha[1], 1 , &hsobel);
    CM_SetAlgs(hCha[2], 1 , &hthold);
    CM_SetAlgs(hCha[3], 1 , &hcopy);
    CM_SetAlgs(hCha[4], 1 , &hgrayscale);
    /*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
    /* Configure capture hardware for NTSC square pixel and siplay hardware   */
    /* for gray scale images.                                             */
    /*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/

    VCAP_config(VCAP_SQP);
    VDIS_config(VDIS_640X480_GS);
    CapRate = MaxRate;

}
/*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
/* I/O task used to capture I/O buffers for each channel & post semapohore    */
/* objects.                                                               */
/*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
void Task_IO()
{
    int i;
    int posted = 0;
    int id = (int)INV;

    /*------------------------------------------------------------------*/
    /* The current set of input and output pointers are obtained by    */
    /* calls to the getFrame and toggleBuffs routine. By using the     */
    /* SYS_FOREVER flag the function blocks until a new frame arrives  */
    /* By setting the output side argument to 0, the next available    */
    /* buffer is returned independent of the display event.            */
    /* ”input” and ”output” are pointers to input frame and output     */
    /* data. The output buffer pointer is obtained for the first time  */
    /* use of the previous output.                                     */
    /*------------------------------------------------------------------*/

    output = VDIS_toggleBuffs(SYS_FOREVER);
    while(1)
    {
        if(id != (int)INV)
        {
            DAT_Wait(id);
            id = (int)INV;
        }

        prev_output = output;
        input  = VCAP_getFrame(SYS_FOREVER);
        output = VDIS_toggleBuffs(0);
        in[0]–>Addr = input–>y1;

      /*-------------------------------------------------------------------------*/
      /* Execute specific channel using CM application framework.               */
      /*-------------------------------------------------------------------------*/

        CM_Exec(hCha[CHA_CT], in, im, NULL, SIG_DEF);

      /*-------------------------------------------------------------------------*/
      /* Loop through all channels and post the corresponding semaphore, the    */
      /* channel is waiting for. Each channel must release the output buffer    */
      /* to continue to allow for synchronization.                              */
      /*-------------------------------------------------------------------------*/
```

```
        for(i = 0; i< CHA_CT; i++)
        {
            if(IncFlag[i] >= MaxRate)
            {
                IncFlag[i] –= MaxRate;
                SEM_ipost(SEM_Ch[i]);
                posted ++;
            }
            IncFlag[i] += Rate[i];
        }

        while(posted)
        {
            SEM_pend(semOutput, SYS_FOREVER);
            posted––;
        }
    }
}
/*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
/* Channel Task used for each channel. The tasks are differentiated by    */
/* the ChaNo.                                                             */
/*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
void Task_Ch(int ChaNo)
{
    double t0,t1;

    /*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
    /* Determine task handle and priority prior to execution.                 */
    /* Wait for the channel semaphore before continuing. Statistic objects    */
    /* are used to measure execution time of each channel algorithm. Output   */
    /* is positioned according to the channel offset value prior to algorithm */
    /* execution. The semaphore object is posted once algorithm complete.     */
    /*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/

    hTask[ChaNo] = TSK_self();
    TaskPri[ChaNo] = TSK_getpri(hTask[ChaNo]);


    while (1)
    {
       SEM_pend(SEM_Ch[ChaNo], SYS_FOREVER);
        SEM_reset(SEM_Ch[ChaNo], 0);
        t0 = ((double)CLK_getl-
time()*(double)CLK_getprd())/(double)CLK_countspms();
        out[0]–>Addr = ((char *)output)+output_offset[ChaNo];


        CM_Exec(hCha[ChaNo], im, out, NULL, SIG_DEF);


        t1 = ((double)CLK_getl-
time()*(double)CLK_getprd())/(double)CLK_countspms();
        STS_add(STS_ExeTime[ChaNo], (t1–t0));


        SEM_ipost(semOutput);
    }
}
/*--------------------------------------------------------------------------*/
/* End of mainapp.c file.                                                  */
/*--------------------------------------------------------------------------*/
```

6.6 Conclusions

Although standalone algorithms are useful for benchmarking and feasibilty studies, most applications require a framework to manage the algorithms and the DSP hardware. This document takes a standalone image processing example and integrates it into the generic Channel Manager Algorithm framework that supports the eXpress DSP Algorithm Standard.

Chapter 7

Conclusions

This chapter summarizes the concepts discussed in the programmer's guide. The chapters that preceded this chapter discussed the underlying software and hardware components that were used to form the IDK. The examples included as part of the programmer's guide were intended to ease the learning curve that users typically face. The IDK makes use of several key software pieces such as CSL, IDM and the Channel Manager Framework to be able to implement multichannel implementation of several algorithms running concurrently in a multi-tasking environment. In addition the ability to develop algorithms, whose run time parameters can be dynamically changed is particularly impressive. A thorough knowledge of these different components should aid programmers in developing new imaging applications using the power of TI C6000 DSP on the Imaging Developer's Kit.

Topic **Page**

7.1 Review of Programmer’s Guide

Chapter 1: **Introduction**

This chapter introduced users to the various underlying software and hardware aspects of the IDK. An insight into what programmers should learn from the different documents was also provided. The programmer’s guide itself is not intended to supplement any of the IDK documentation. It is rather intended to decrease the learning curve associated with using the IDK.

Chapter 2: **Image Data Manager**

The Image Data Manager (IDM) is a software component of the IDK that provides the users the abstraction of double buffered DMAs to perform the data transfers in the background, by seamlessly bringing data from external to internal memory without stalling the CPU. The IDM provides to the user a set of APIs that make use of the DAT calls in CSL to effectively bring the data required for an algorithm. The “double buffered” and “sliding window” mechanisms are supported by IDM. Useful features such as stream rewind allow users to implement different kinds of data flows in an effective manner. IDM is used in the examples included in the programmer’s guide to implement the data flow required for every algorithm.

Chapter 3: **Development of Application Divers as Generic Templates for Image Processing**

This chapter uses IDM to develop a set of application drivers for grayscale and color based processing. These drivers allow the user to specify regions in internal and external memory between which data transfer is to take place. Although the actual data on the Imaging daughter card, resides in separate fields, the application drivers developed in this chapter allow users to process the image either in progressive order or in field order. Further when data is being processed in field order, the user can choose the number of lines to be fetched every time and thus balance the data bandwidth to the processing bandwidth. The application drivers for color allow users to investigate several possible trade-offs between implementing data transfer through the EDMA versus using the CPU. The resulting performance on the color and grayscale application drivers are summarized.

Chapter 4: **Application Development and Prototyping Using Generic Templates**

This chapter demonstrates how the application drivers developed in the previous chapter can be used either as is, or with a few modifications to provide the data required for developing new applications. This is demonstrated by way of a median filtering example for grayscale processing and a color rotation demonstration for color based processing.

Chapter 5: **Image Processing Using ImageLIB**

This chapter demonstrates the power of ImageLIB a collection of highly optimized image processing functions for the TMS320C6000 DSP. This chapter uses three image processing functions from ImageLIB to put together the image processing demonstration that forms part of the IDK. The image processing functions allow users to process a variable amount of the image for every invocation, thus allowing the user to trade off the processing bandwidth, data transfer bandwidth to interruptability requirements for the system. This chapter is intended to demonstrate the recommended application development flow from start to finish on a complete application of reasonable complexity.

Chapter 6: **Integration of an Application into the Imaging Framework**

The availability of an imaging framework for users to leverage is one of the most attractive software features of the IDK. The Imaging Framework has the ability to run either multiple instances of an algorithm or multiple instances of different algorithms. This allows for a multichannel implementation of an algorithm. This chapter demonstrates to users how the application developed in the previous chapter, can be converted to be eXpressDSP-compliant. It also demonstrates how the eXpressDSP-compliant algorithm can then use the Channel Manager (Imaging Framework) APIs to register itself, and to be invoked dynamically. In addition the ability to vary the parameters of an algorithm at run-time is another attractive feature of the framework. Development of applications that are eXpressDSP-compliant guarantee the inter-operability of various standards developed by different vendors. It is extremely important that new applications developed using the IDK be developed in an eXpressDSP-compliant framework.