TEXAS INSTRUMENTS

Application Report
SPRA904A – October 2003

A Multichannel Motion Detection System Using eXpressDSP RF5 NVDK Adaptation

Adit Sahasrabudhe *Technical Staff DSP Applications – Waltham, MA*

ABSTRACT

Reference Framework 5 (RF5) for eXpressDSP Software provides a fast ramp to DSP application development using DSP/BIOS and the eXpressDSP Algorithm Standard (XDAIS). The RF5 adaptation for the Network and Video Developer's Kit (NVDK) provides starterware with which to develop network and video specific applications.

This application note uses this adaptation to develop a motion detection system. Motion detection is done by comparing the current video frame to a specified reference frame. Using a pixel-by-pixel comparison, unequal portions of the current frame are colored. This application note guides you through the steps needed to transform the original generic architecture into the desired system.

The target audience is anyone who wishes to develop a video-based multichannel application using RF5. This application note covers all necessary steps—from adding channels and cells to implementing control messages.

Contents

1 Overview **2**
2 System Requirements **3**
2.1 Hardware Requirements 3
2.2 Software Requirements 3
3 Suggested Directory Structure **4**
4 Software Architecture and Data Flow **5**
4.1 Original Process Thread Architecture 6
4.2 Motion Detection System Process Thread Architecture 7
5 System Implementation **8**
5.1 Adding the DIFF Algorithm and Integrating Cells 8
5.2 Processing CIF Frames 11
5.3 Creating a Processing Channel 13
5.4 Adding Cells to a Processing Channel 16
5.5 Resetting the Reference Frame and Changing DIFF Colors 19
5.5.1 Host 20
5.5.2 thrControl 21
5.5.3 thrProcess 22
6 Conclusion **23**
7 References **23**

Figures

Figure 1. Directory Structure....4
Figure 2. Diagram Symbols....5
Figure 3. Application Data Flow....5
Figure 4. Original Process Thread Structure....6
Figure 5. Motion Detection System Process Thread....7
Figure 6. Camera Captures a CIF YUV Stream....11
Figure 7. De-multiplexing into Even and Odd Y, U, and V Buffers....11
Figure 8. Prescale from 4:2:2 to 4:2:0....12
Figure 9. CONV_YUY2toIYUV....12
Figure 10. Channel 4....15
Figure 11. Control Message Flow....19
Figure 12. SetReference Menu Option via GEL....20
Figure 13. Text Box to Change Color via GEL....21

1 Overview

This application note describes how the Reference Frameworks 5 (RF 5) Network and Video Developer's Kit (NVDK) adaptation can be modified to create a simple multichannel motion detection system. We define a motion detection system as one that compares current video frames to a desired reference frame. If a certain number of pixels are different, that signifies that "motion" has taken place (the image has changed). The pixels that have changed are colored.

RF5 is designed for multichannel and multi-algorithm applications. For a detailed description of RF5, please see *Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High-Density System* (SPRA795). In addition, knowledge of the NVDK Adaptation for RF5 upon which this motion detection system is based is suggested. The document, *A Video, Audio, Networking System on the C64xx NVDK Using eXpressDSP RF5* (SPRA844), details the NVDK adaptation.

This motion detection system uses many features of RF5. The highlights are as follows.

- **DIFF Algorithm and Cell Integration.** RF5 lets you easily integrate XDAIS algorithms. We do not detail the actual development of the algorithm; rather, we discuss the incorporation of an existing algorithm. For information on XDAIS, please see *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352E). In RF5, existing XDAIS algorithms are integrated by wrapping them with a "cell" and placing them in the application code. This document shows how to integrate the DIFF algorithm in this manner. The DIFF (Difference) algorithm is where the actual motion detection processing is done.

- **Processing a CIF Frame.** The output of the camera into the NVDK is CIF format. The current NVDK Adaptation pre-scales the data down to QCIF. For this application, we want to keep the CIF processing frame size. The CONV_YUY2toIYUV function in Ateme's Graphics Library allows us to do this quickly and effectively.

- **Creating a Processing Channel.** One of the most important features of RF5 is its multichannel capability. This adaptation is implemented as a 4-channel system. The channels have 1, 2, 2, and 3 cells, respectively.

- **Adding Cells to a Processing Channel.** Given this channel structure, we need to be able to arrange cells within the channels appropriately. For this we use three different cells:
  - **YUV2RGB.** Takes YUV data and converts it to RGB format for output to a VGA display.
  - **ROTATE.** Rotates a (U,V) pair on the color spectrum.
  - **DIFF.** Runs the difference algorithm, taking the current frame and comparing it to a reference frame. If any pixel is different, it assigns a user-defined color to that pixel.
- **Resetting the Reference Frame and Changing DIFF Colors.** Using the control thread, we can pass messages to the process thread via the DSP/BIOS Mailbox (MBX) object. The control messages are used to reset the reference frame and to change the color of the unequal pixels in the DIFF cell.

2 System Requirements

2.1 Hardware Requirements

The main piece of hardware needed for this design is the NVDK board from Ateme. With a TI C6416 DSP, the board is specially designed for network and video applications. It can be purchased from http://www.dspestore.com.

A VGA monitor is needed to test the system and see the output. The monitor attaches directly to the NVDK, so no additional wires or adapters are needed.

An emulator is used to connect CCStudio to the NVDK. In addition to simply loading the program to the DSP, it also provides debugging capabilities over JTAG. There are various emulators available, and they can be ordered from http://www.dspestore.com.

2.2 Software Requirements

Code Composer Studio (CCStudio) is an Integrated Development Environment designed for TI Digital Signal Processors. With features like Real Time Analysis, Real Time Data Exchange (RTDX), and DSP/BIOS, CCStudio helps shorten development time. For this motion detection application, CCStudio 2.2 is required as a minimum.

Along with the actual NVDK, Ateme also provides development software. The version needed, 1.3, can be downloaded from http://www.ateme.com.

3 Suggested Directory Structure

Figure 1 shows the suggested directory structure for this application.

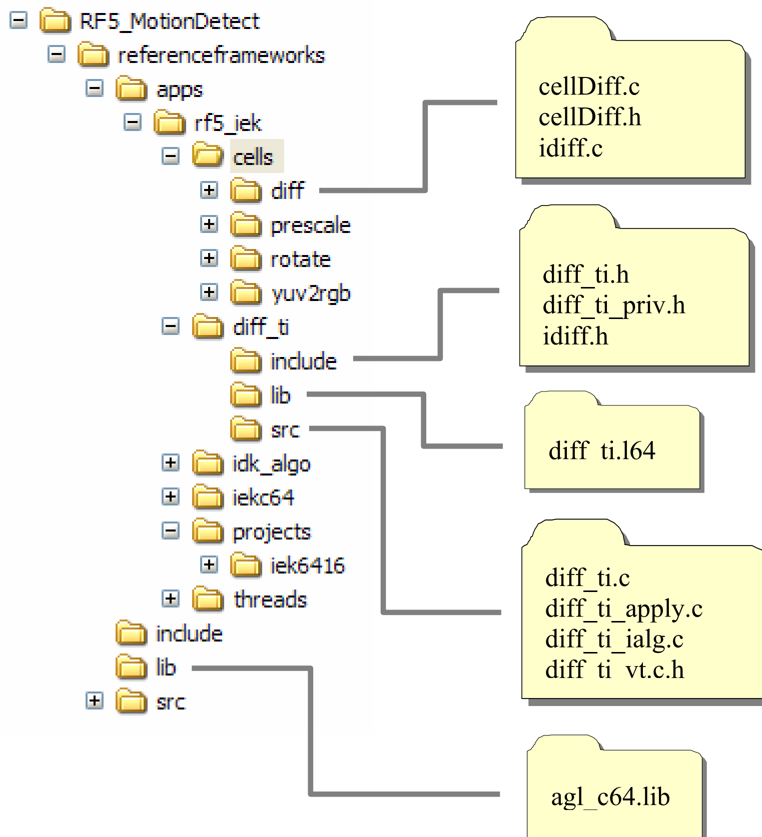

Figure 1. Directory Structure

A note about a few of the directories:

- **\referenceframeworks\apps\rf5_iek\cells\diff.** Contains the cell wrapper code for the DIFF algorithm. Notice that it is located under \apps so that different applications can house their own set of cells.
- **\referenceframeworks\apps\rf5_iek\diff_ti.** Contains all source code for the DIFF algorithm. The \lib subfolder contains the library (diff_ti.l64) that is linked into the application project.
- **\referenceframeworks\lib.** Contains all libraries used by RF5. It also houses the Ateme Graphics Library (agl_c64.lib).

4 Software Architecture and Data Flow

We begin by describing the overall structure of the application, in terms of software architecture and data flow. The symbol key to all data flow diagrams in this application note is shown in Figure 2.

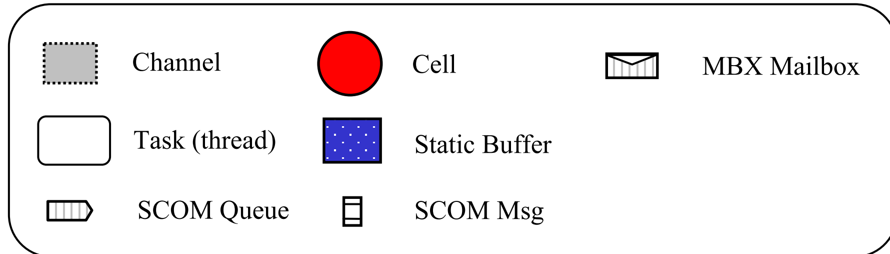

Figure 2. Diagram Symbols

The general application data flow is shown in Figure 3.

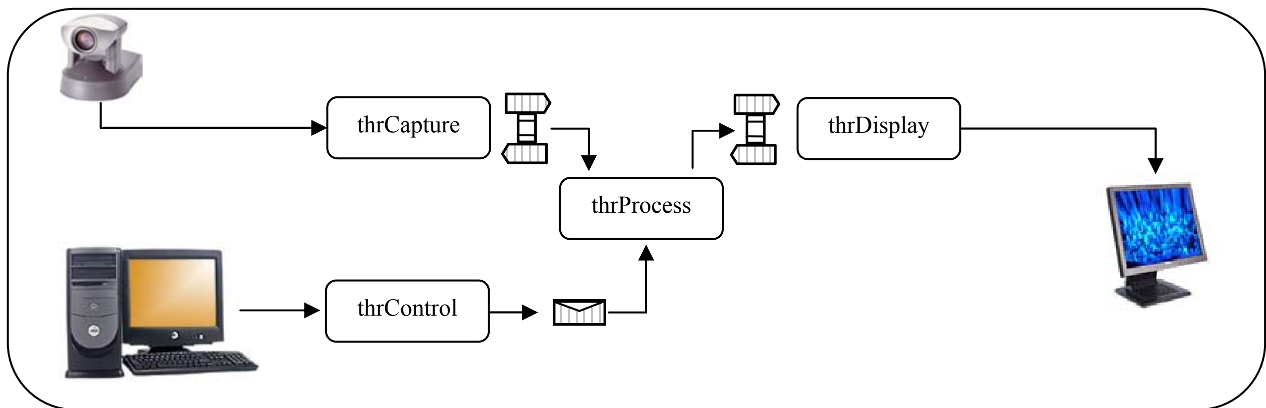

Figure 3. Application Data Flow

4.1 Original Process Thread Architecture

The overall structure in Figure 3 remains in our motion detection system. The changes occur inside the process thread. Let us examine that portion more closely. Originally, the NVDK adaptation for RF 5 has a process thread like that displayed in Figure 4.

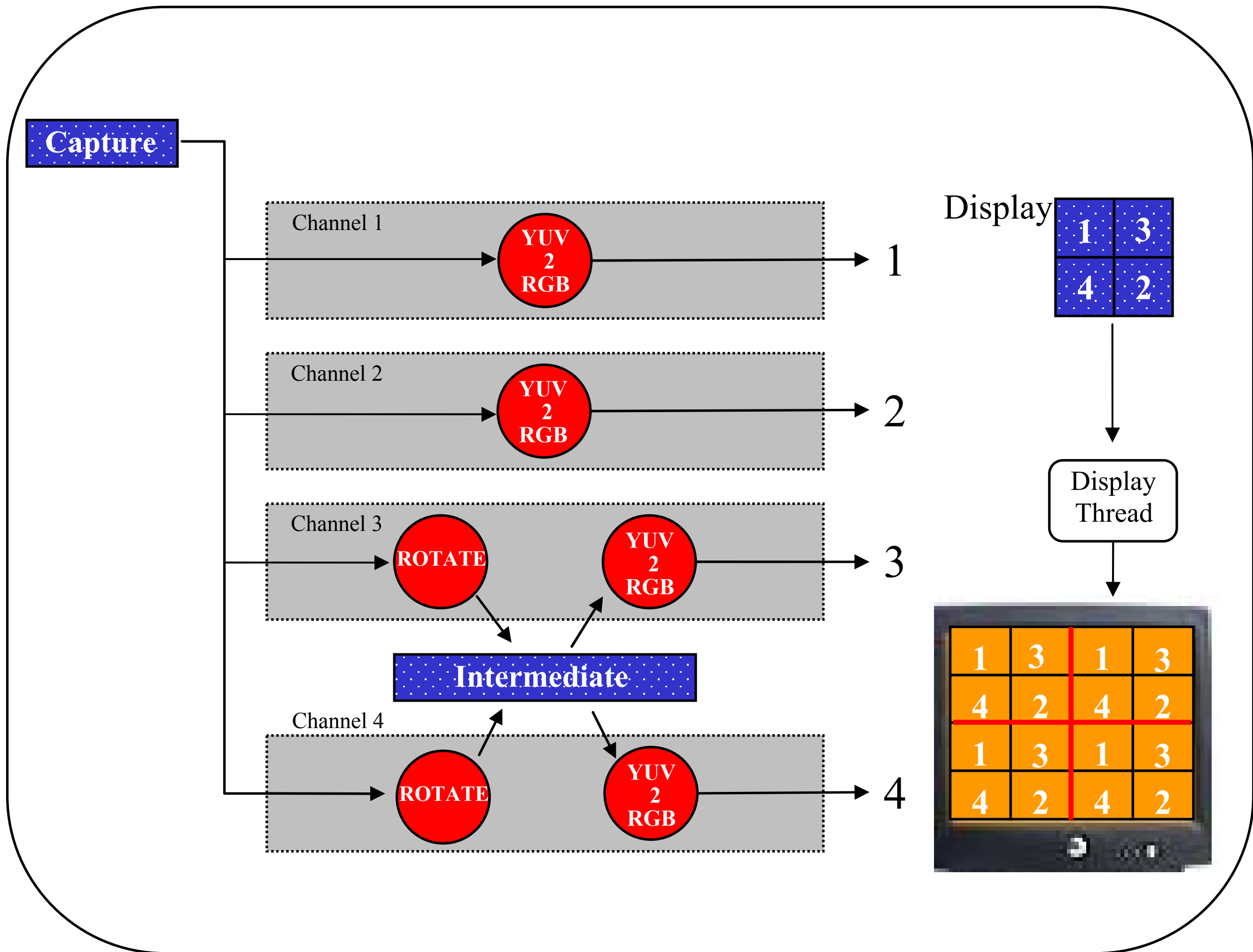

Figure 4. Original Process Thread Structure

Channels 1 and 2 in Figure 4 simply convert the video format from YUV to RGB; therefore, we denote that type of channel to be *pass-through*.

Channel 3 performs a color rotation. It also converts the video format from YUV to RGB.

Channel 4 performs a color rotation along with a color reversal (negation). The color is reversed automatically because the ROTATE algorithm instance is configured with a parameter that indicates the reversal should be performed. This channel also converts the video format from YUV to RGB.

The camera captures an image that is CIF (4:2:2, 352x240). Each frame is received as one buffer with the Y, U, and V values interlaced. The capture thread pre-scales each frame to QCIF (4:2:0, 176x120), while also separating the Y, U, and V values into unique buffers. With this downscaling we see that the processing frame size for this system is QCIF.

As is shown in the display buffer, we only fill the top left quadrant with each channel. The display thread simply replicates that portion of the buffer into the other quadrants.

It should be noted that in addition to supporting variable frame sizes, RF5 also easily implements PAL and NTSC systems.

4.2 Motion Detection System Process Thread Architecture

Now we examine the structure of the process thread in our motion detection system.

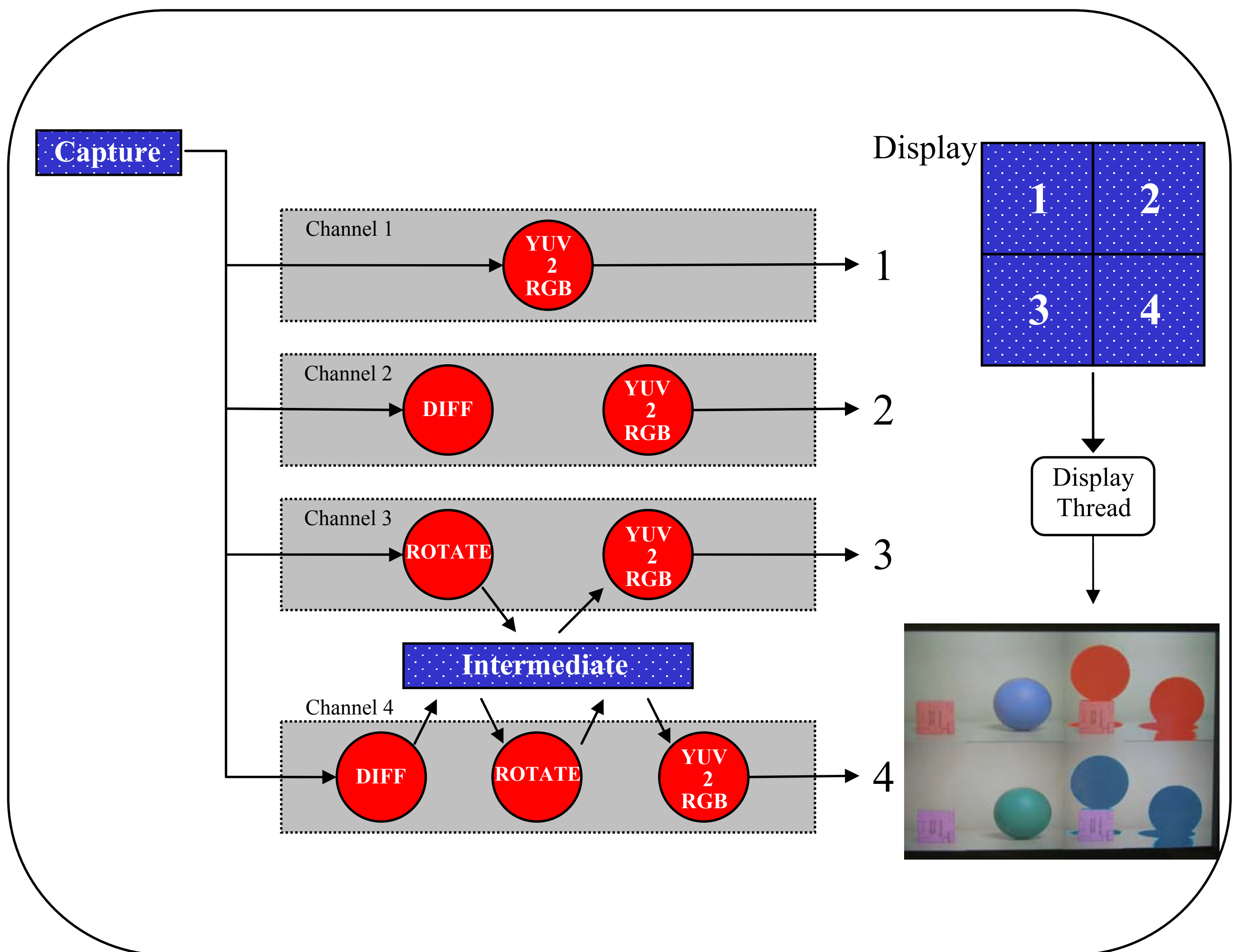

Figure 5. Motion Detection System Process Thread

Channel 1 in the top-left quadrant is a live feed.

Channel 2 is a *difference* channel, where the DIFF algorithm is what performs the motion detection. We can see this in the top-right quadrant of Figure 5. Since both the original and current locations of the ball have changed, those areas and their shadows are colored red.

Channel 3 in the bottom-left quadrant is a *rotate* channel. The colors of the live feed have been rotated to make the box and the ball change colors.

Channel 4 is a combination of 2 and 3, having a DIFF followed by a ROTATE. Not only is the box a different color, but the colored movements that are red in Channel 2 have also turned blue.

The processing frame size has changed. Instead of processing QCIF frames, we are processing CIF frames. This shows in the output since each quadrant of the monitor displays a unique channel.

5 System Implementation

5.1 Adding the DIFF Algorithm and Integrating Cells

DIFF is an XDAIS-compliant difference algorithm that needs to be incorporated into our application. We can describe the operation of the algorithm very easily:

1. Compare all pixels in a certain block and count the number of differences
2. If there are more than DIFFTHRESHOLD (user-defined) differences, go to 3. Else, stop.
3. Find all different pixels and change the color value.

The source code for the algorithm is located in referenceframeworks\apps\rf5_iek\diff_ti. This folder contains all source code, header files, and libraries associated with the DIFF algorithm, so that it can be used in your own system.

The /lib folder contains the library diff_ti.l64. We need to create a cell wrapper around this algorithm to incorporate it into RF5. Two things need to be done:

- **Include diff_ti.l64.** Within the application's linker command (.cmd) file, the following lines are added.

```
/*
 *  Algorithm DIFF: bind the generic DIFF symbol to TI's implementation
 *  of the algorithm, and include the appropriate library
 */
-l diff_ti.l64
_DIFF_IDIFF = _DIFF_TI_IDIFF;
```

- **Add appropriate source and header files.**
  - **cellDiff.c.** This is the main source file for running the DIFF algorithm. It contains the vector table of ICELL_Fxns being used by the algorithm as well as any function prototypes needed.

```
#include <std.h>
#include <csl_dat.h>
#include <algrf.h>
#include <icell.h>
#include <utl.h>

#include "cellDiff.h"
#include "appResources.h"
#include "appThreads.h"

static void runDIFF( IDIFF_Handle handle, Short **inData,
                       Uint32 **outData, DIFF_Env * env );

// v-table for this cell
ICELL_Fxns DIFF_CELLFXNS = {
    NULL,                   // cellClose
    NULL,                   // cellControl
    DIFF_cellExecute,       // cellExecute
    NULL                    // cellOpen
};
```

DIFF_cellExecute is the main entry point for the application into the algorithm.

```
/*
 *  ======== DIFF_cellExecute ========
 */
Bool DIFF_cellExecute( ICELL_Handle handle, Arg arg )
{
    IDIFF_Handle diffHandle = (IDIFF_Handle)handle->algHandle;

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

    runDIFF(diffHandle,
         (Short **)handle->inputIcc[0]->buffer,
         (Uint32 **)handle->outputIcc[0]->buffer,
         (DIFF_Env *)handle->cellEnv );

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

    return(TRUE);
}
```

The runDIFF function sets up appropriate intermediate buffer addresses and the DIFF environment, DIFF_Env, which is defined in the header file cellDiff.h. It is within runDIFF that the apply function is called. This is where the actual work of the DIFF algorithm is done. It is entirely possible to have an algorithm in which there is no need for an intermediate function. In that case, one would just replace the call to runDIFF with the algorithm's apply function.

– **cellDiff.h.** In addition to the function prototype for any of the ICELL_Fxns used in the algorithm, this file also contains the definition of the cell environment.

```
/*
 *  ======== DIFF_cellExecute ========
 *  Function used to execute the algorithm corresponding to the cell
 */
Bool DIFF_cellExecute( ICELL_Handle handle, Arg arg );

/*
 * ======== DIFF cell environment ========
 */
typedef struct DIFF_Env {
    Char *intYBuf;            //internal Y buffer for processing
    Char *intCrBuf;           //internal Cr buffer for processing
    Char *intCbBuf;           //internal Cb buffer for processing
    Char *prevY;              //Y reference frame buffer
    Char *prevCr;             //Cr reference frame buffer
    Char *prevCb;             //Cb reference frame buffer
    Uns   yBufSize;           //Number Y pixels in single processing block
    Uns   crBufSize;          //Number Cr pixels in single processing block
    Uns   cbBufSize;          //Number Cb pixels in single processing block
    Uns   numBlocks;          //Number of processing blocks
    Bool  SetReference;       //If true, need to reset reference frame
    Int   yValue;             //Y value for unequal pixels
    Int   crValue;            //Cr value for unequal pixels
    Int   cbValue;            //Cb value for unequal pixels
} DIFF_Env;
```

The DIFF_Env structure holds information about the algorithm's environment, e.g. intermediate buffer addresses, processing block size, whether or not the current frame is the reference frame, and what the color of the unequal pixels should be.

– **idiff.h.** The following structures and objects are defined in idiff.h.

The IDIFF_Obj:

```
/*
 *  ======== IDIFF_Obj ========
 *  This structure must be the first field of all DIFF instance objects
 */
typedef struct IDIFF_Obj {
    struct IDIFF_Fxns *fxns;
} IDIFF_Obj;
```

IDIFF_Params structure:

```
/*
 *  ======== IDIFF_Params ========
 *  This structure defines the creation parameters for all DIFF objects
 */
typedef struct IDIFF_Params {
    Int  size;          /* must be first field of all params structures */
} IDIFF_Params;
```

The IDIFF_Fxns structure defining operations performed on DIFF objects:

```
/*
 *  ======== IDIFF_Fxns ========
 *  This structure defines all of the operations on DIFF objects
 */
typedef struct IDIFF_Fxns {
    IALG_Fxns   ialg;    /* IDIFF extends IALG */
    Void (*apply)(IDIFF_Handle handle, unsigned char y[],
        unsigned char cr[], unsigned char cb[],
        unsigned char prevY[],unsigned char prevCr[],
        unsigned char prevCb[], Int lumaSize, Int chromaSize,
        Int yValue, Int crValue, Int cbValue, Int procWidth);
} IDIFF_Fxns;
```

5.2 Processing CIF Frames

In the original NVDK adaptation, the capture thread performs two functions. First, it separates the Y, U, and V values from the interlaced buffer into six unique buffers for the even and odd indices of Y, U, and V as shown in Figure 7. The reason for splitting into even and odd buffers is for the second function thrCapture performs. It pre-scales those buffers from CIF to QCIF format.

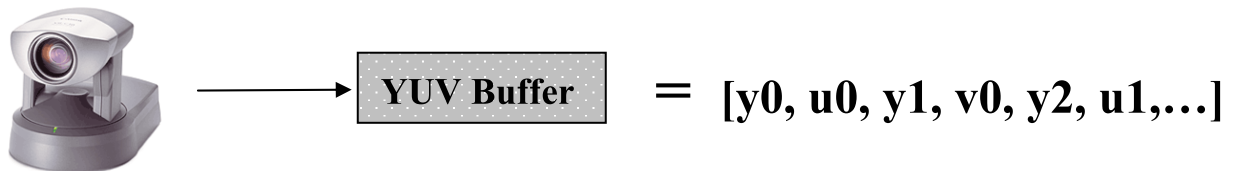

Figure 6. Camera Captures a CIF YUV Stream

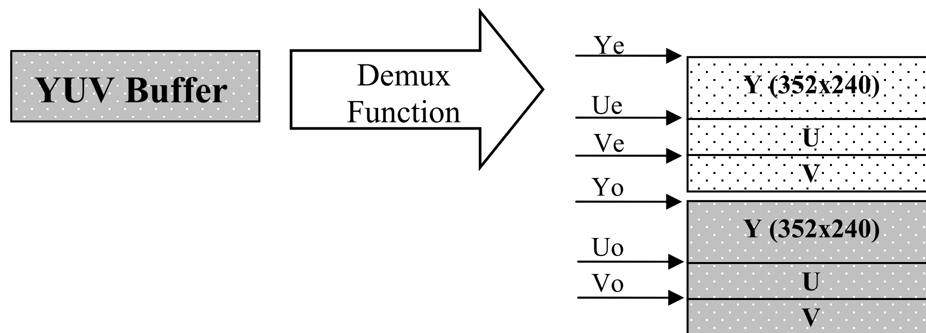

Figure 7. De-multiplexing into Even and Odd Y, U, and V Buffers

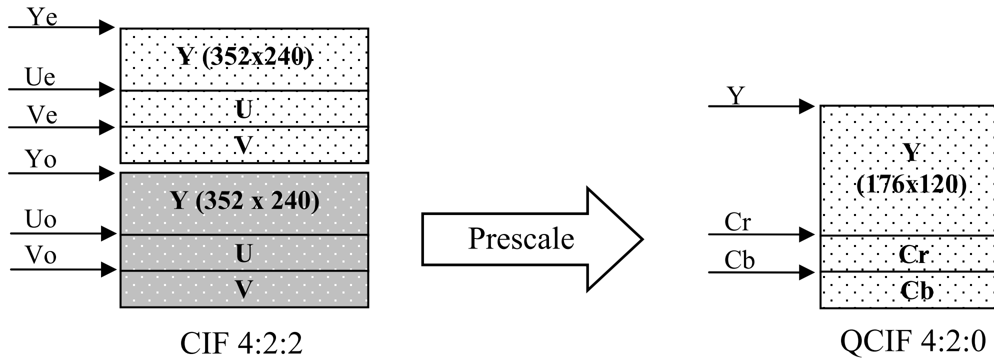

Figure 8. Prescale from 4:2:2 to 4:2:0

For our motion detection system, our cells require us to have the Y, U, and V values separated into their own buffers but not further divided into even and odd. Therefore, we need to be able to de-multiplex the input YUV Buffer without the even and odd separation. Further, we need to remove the pre-scale algorithm, since we want to keep the processing frames in CIF format.

One of the functions provided in Ateme's Graphics Library is CONV_YUY2toIYUV. This function does the appropriate extraction from the interlaced buffer into unique Y, U, and V buffers as in Figure 9.

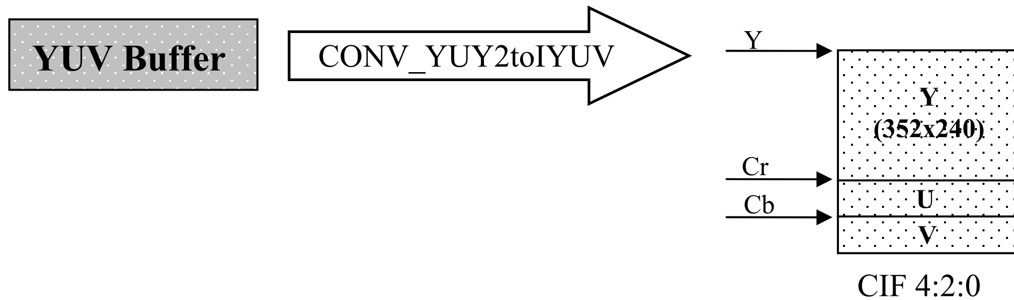

Figure 9. CONV_YUY2toIYUV

In terms of what code needs to be changed, we simply replace the Demux function with CONV_YUY2toIYUV and bypass the pre-scale algorithm in thrCapture.c. We add the following code to function getInputFrame:

```
static Void getInputFrame(Void)
{
...
    CONV_YUY2toIYUV((const Uint16 *) framePtr, CAPF_WIDTH, CAPF_HEIGHT,
                    (Uint16 *) thrCaptureBufferY,
                    (Uint16 *) thrCaptureBufferU,
                    (Uint16 *) thrCaptureBufferV);

    // Copy address of Y, U, and V into output buffers
    thrCaptureBufYUV[0] = thrCaptureBufferY;
    thrCaptureBufYUV[1] = thrCaptureBufferU;
    thrCaptureBufYUV[2] = thrCaptureBufferV;
}
```

The pre-scale algorithm is bypassed by just not making a call to CHAN_execute in the main function of the capture thread, thrCaptureRun.

One thing to notice are the frame dimensions CAPF_WIDTH and CAPF_HEIGHT passed to CONV_YUY2toIYUV. These represent the height and width of the captured frame (coming from the camera) and are defined in the header file appThreads.h. In that same file, there are two other definitions that need to be altered. PROCF_WIDTH and PROCF_HEIGHT represent the height and width of the processing frame.

```
// Processed frame dimensions across processing channels
//#define PROCF_HEIGHT            (CAPF_HEIGHT >> 1)          //Old Values
//#define PROCF_WIDTH             (CAPF_WIDTH >> 1)

#define PROCF_HEIGHT            (CAPF_HEIGHT)                 //New Values
#define PROCF_WIDTH             (CAPF_WIDTH)
```

NOTE: Changing the processing size of the frames implies that larger chunks of data are to be passed into and out of the cells. Many cell wrappers use the DAT_copy function for data transfers. However, that function has a 64k word transfer limit. So when processing CIF frames, be sure to change all DAT_copy statements to DAT_copy2d. To do this it is necessary to be certain that the call to DAT_open in appThreadInit has the DAT_OPEN_2D flag.

We have now taken care of integrating an XDAIS algorithm into a cell wrapper and making some modifications to process a CIF frame instead of QCIF. Let us now examine how we add the type of channels we want in Figure 5.

5.3 Creating a Processing Channel

This section describes how to create and use the desired channel structure.

1. **Add the appropriate enumerations to thrProcess.h.** This step allows us to create any channel structure we like. One set of enumerations defines the type and number of each channel, while another set describes the cells each channel contains.

   The following set of enumerations describes how many of each type of channel we are implementing. Since all channels in Figure 5 are unique, there is only one of each. Hence,

NUMPASSCHANS, NUMDIFFCHANS, NUMROTATECHANS, and NUMCOMBOCHANS all have values of 1.

```
// Pass-thru Channels
enum { CHPASS0   = 0,
       NUMPASSCHANS
};

// Difference Channels
enum { CHDIFF0   = 0,
       NUMDIFFCHANS
};

// Rotate Channels
enum { CHROTATE  = 0,
       NUMROTATECHANS
};

// Combination Channels
enum { CHCOMBO  = 0,
       NUMCOMBOCHANS
};
```

The following set of enumerations describes the cells each channel uses. Channels 1, 2, 3, and 4 have 1, 2, 2, and 3 cells, respectively. Accordingly, CHPASSNUMCELLS, CHDIFFNUMCELLS, CHROTATENUMCELLS, and CHCOMBONUMCELLS have the values 1, 2, 2, and 3, respectively.

```
// Cells for Pass-thru
enum { CHPASSCELLYUV  = 0,
       CHPASSNUMCELLS
     };

// Cells for Difference
enum { CHDIFFCELLDIFF = 0,
       CHDIFFCELLYUV,
       CHDIFFNUMCELLS
     };

// Cells for Rotate
enum { CHROTATECELLROTATE = 0,
       CHROTATECELLYUV,
       CHROTATENUMCELLS
     };

// Cells for Combo
enum { CHCOMBOCELLDIFF = 0,
       CHCOMBOCELLROTATE,
       CHCOMBOCELLYUV,
       CHCOMBONUMCELLS
     };
```

For the next few steps, Channel 4 is used as an example. Since it is a superset of the other channels, an understanding of its working suffices. Recall the structure of channel 4.

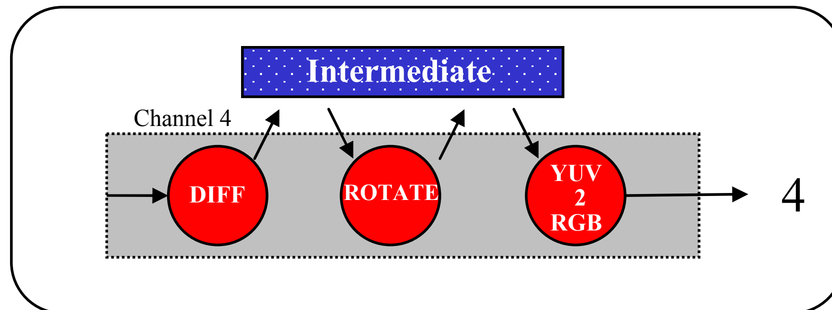

Figure 10. Channel 4

2. **Modify the ThrProcess structure in thrProcess.h.** The ThrProcess structure defines the overall structure and state of the system. Among other variables, it contains the CHAN and ICELL objects for each channel. The Int comboPlacement defines to which quadrant in the display buffer we are putting this channel.

```
/* Definition of the structure describing the state of the thread. */
typedef struct ThrProcess {
    CHAN_Obj      comboChans[NUMCOMBOCHANS];
...
    ICELL_Obj     comboCells[NUMCOMBOCHANS * CHCOMBONUMCELLS];
...
    Int           comboPlacement[NUMCOMBOCHANS];
...
}
```

3. **Open each channel.** The following code opens each instance of this channel type. We run a loop for the number of channels that have the same DIFF→ROTATE→YUV2RGB structure. In our system the loop is only run once. The code is located in the file thrProcess.c in the function setParamsAndStartChannels.

```
//Open Combo chans
for (chanNum = 0; chanNum < NUMCOMBOCHANS; chanNum++) {
    // Set the algorithm's parameters
    thrProcess.comboCells[chanNum * CHCOMBONUMCELLS].algParams =
        (IALG_Params *)&diffParams;
    thrProcess.comboCells[
        chanNum * CHCOMBONUMCELLS + CHCOMBOCELLROTATE].algParams =
        (IALG_Params *)&rotateParams;
    thrProcess.comboCells[
        chanNum * CHCOMBONUMCELLS + CHCOMBOCELLYUV].algParams =
        (IALG_Params *)&yuv2rgbParams;

    // Open the channel
    UTL_logDebug1("Process: Combo Channel Number: %d", chanNum);
    rc = CHAN_open( &thrProcess.comboChans[chanNum],
        &thrProcess.comboCells[chanNum * CHCOMBONUMCELLS],
        CHCOMBONUMCELLS, NULL );
    UTL_assert( rc == TRUE );
}
```

The channels are opened as part of an initialization routine; therefore, this previous code is run only once.

4. **Execute the channel.** We now get into the channel execution portion of the code. This is contained in the main function of the process thread, called thrProcessRun, and is executed in an infinite loop:

```
//Execute all combo channels
for (chanNum = 0; chanNum < NUMCOMBOCHANS; chanNum++) {

    chan = &(thrProcess.comboChans[chanNum]);

    // assign the buffers to the ICC objects
    // Redundant with the SCOM queue is single-buffered. Needed when the
    // queue is multiple-buffered.
    ICC_setBuf(chan->cellSet[CHCOMBOCELLDIFF].inputIcc[0],
        scombufCap->bufYCRCB, 0);

    ICC_setBuf(chan->cellSet[CHCOMBOCELLYUV].outputIcc[0],
        scombufDisp->bufRGB + thrProcess.comboPlacement[chanNum], 0);

    UTL_stsStart( stsExeTimeChCombo );
    rc = CHAN_execute( &thrProcess.comboChans[ chanNum ], NULL );
    UTL_assert( rc == TRUE );
    UTL_stsStop( stsExeTimeChCombo );
}
```

Having added everything necessary to create a custom channel, we next look at a few of the extra things that are needed to add the cells that we want within a channel.

5.4 Adding Cells to a Processing Channel

Cells give us a convenient and consistent way to wrap an algorithm and insert it into RF5. The Channel Manager (CHAN) then uses the wrapper as an interface for efficient execution. In Section 5.1 we described how to create the cell wrapper itself for an algorithm. Now, we describe how to integrate that cell into a processing channel.

First, each cell's environment variable needs to be declared in thrProcess.h:

```
/* Definition of the structure describing the state of the thread. */
typedef struct ThrProcess {
...
    DIFF_Env    diffEnv[NUMDIFFCHANS + NUMCOMBOCHANS];
    ROTATE_Env  rotateEnv[NUMROTATECHANS + NUMCOMBOCHANS];
...
}
```

The environment structure is used to store parameter, status, or any information about the algorithm that is needed on a persistent frame-to-frame basis. These structures need to be initialized, and we do that inside the function thrProcessInit:

```
//Set up the diff cell’s env.
for(i = 0; i < (NUMDIFFCHANS + NUMCOMBOCHANS); i++)
{
    thrProcess.diffEnv[i].intYBuf   = intYBuf;
    thrProcess.diffEnv[i].intCrBuf  = intCrBuf;
    thrProcess.diffEnv[i].intCbBuf  = intCbBuf;
    thrProcess.diffEnv[i].prevY     = prevY;
    thrProcess.diffEnv[i].prevCr    = prevCr;
    thrProcess.diffEnv[i].prevCb    = prevCb;
    thrProcess.diffEnv[i].yBufSize  = DIFF_Y_BUFSIZE;
    thrProcess.diffEnv[i].crBufSize = DIFF_CR_BUFSIZE;
    thrProcess.diffEnv[i].cbBufSize = DIFF_CB_BUFSIZE;
    thrProcess.diffEnv[i].numBlocks = DIFF_NUMBLOCKS;

//Set up the rotate cell's env
for (i = 0; i < (NUMROTATECHANS + NUMCOMBOCHANS); i++)
{
    thrProcess.rotateEnv[i].intYBuf = intYBuf;
    thrProcess.rotateEnv[i].intCrBuf = intCrBuf;
    thrProcess.rotateEnv[i].intCbBuf = intCbBuf;
    thrProcess.rotateEnv[i].yBufSize = ROTATE_Y_BUFSIZE;
    thrProcess.rotateEnv[i].crBufSize = ROTATE_CR_BUFSIZE;
    thrProcess.rotateEnv[i].cbBufSize = ROTATE_CB_BUFSIZE;
    thrProcess.rotateEnv[i].numBlocks = ROTATE_NUMBLOCKS;
}
```

The offsets in this code are to account for the DIFF_Env and ROTATE_Env structures for the cells in channels 2 and 3, respectively.

Again using channel 4 as an example, we register each cell within a channel. This is also done in setParamsAndStartChannels within thrProcess.c:

```
// ------------------------------------
// Combo Channels (DIFF and then ROTATE)
// ------------------------------------

for (chanNum = 0; chanNum < NUMCOMBOCHANS; chanNum++) {
    /*
     *  cell 0 - DIFF: create an input and output linear ICC.
     *  The address to the input ICC will be set in the thrProcessRun()
     *  function via ICC_setBuf().
     */
...
    inputIcc = (ICC_Handle)ICC_linearCreate( NULL, 0);
    outputIcc = (ICC_Handle)ICC_linearCreate( bufYCRCB,
                        sizeof(bufYCRCB));

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

    /* cell 1 - ROTATE: create an input and output linear ICC */
...
    inputIcc = (ICC_Handle)ICC_linearCreate(
        bufYCRCB, sizeof(bufYCRCB));
    outputIcc = (ICC_Handle)ICC_linearCreate(
        bufYCRCB, sizeof(bufYCRCB));

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

    /* cell 2 - YUV2RGB: create an input and output linear ICC */
...
    inputIcc = (ICC_Handle)ICC_linearCreate(
        bufYCRCB, sizeof(bufYCRCB));
    outputIcc = (ICC_Handle)ICC_linearCreate(bufRGBundoubled +
        thrProcess.comboPlacement[chanNum],
        PROCF_SIZE_IN_PIXELS);

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

5.5 Resetting the Reference Frame and Changing DIFF Colors

The control message scheme in this architecture is rather straightforward. Figure 11 shows a diagram of the control message flow.

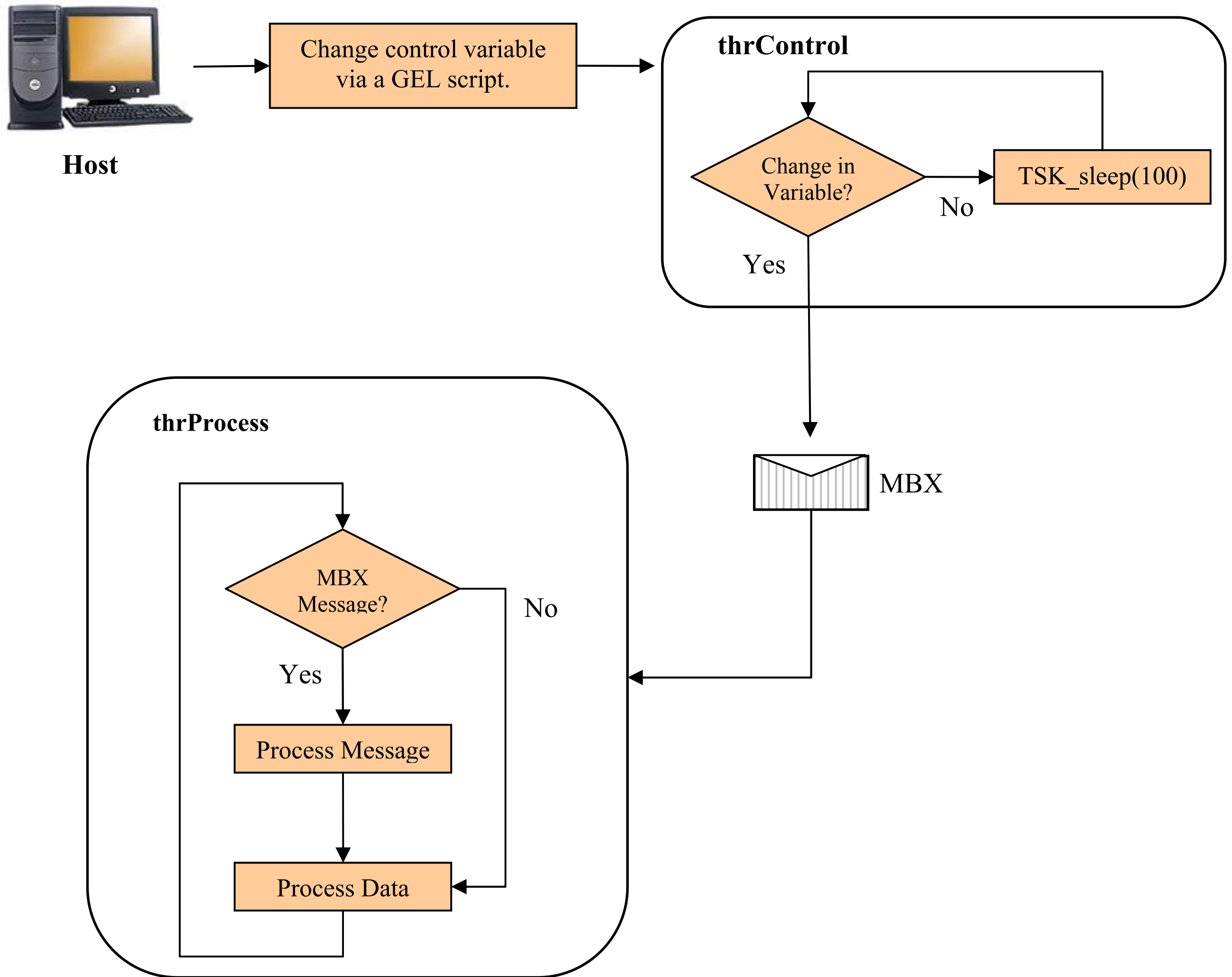

Figure 11. Control Message Flow

The following sections highlight portions of the control message flow shown in Figure 11.

5.5.1 Host

Through a GEL script, we make a change to a global variable. (The script file is located at \referenceframeworks\apps\rf5_iek\projects\iek6416\appControl.gel.) The global variable is defined in thrControl.c as follows:

```
//Structure for external control variables
typedef struct ExternalControl {
    Int setReference;       //0 or 1, to set reference frame
    Int yValue;             //New color values. All three,
    Int crValue;            //Y, Cr, and Cb are passed together
    Int cbValue;            //in one message
} ExternalControl;
```

As this structure shows, we have implemented two different types of control messages. First, we reset the reference frame for the DIFF algorithm's frame comparisons. To do this we change the value of SetReference to 1.

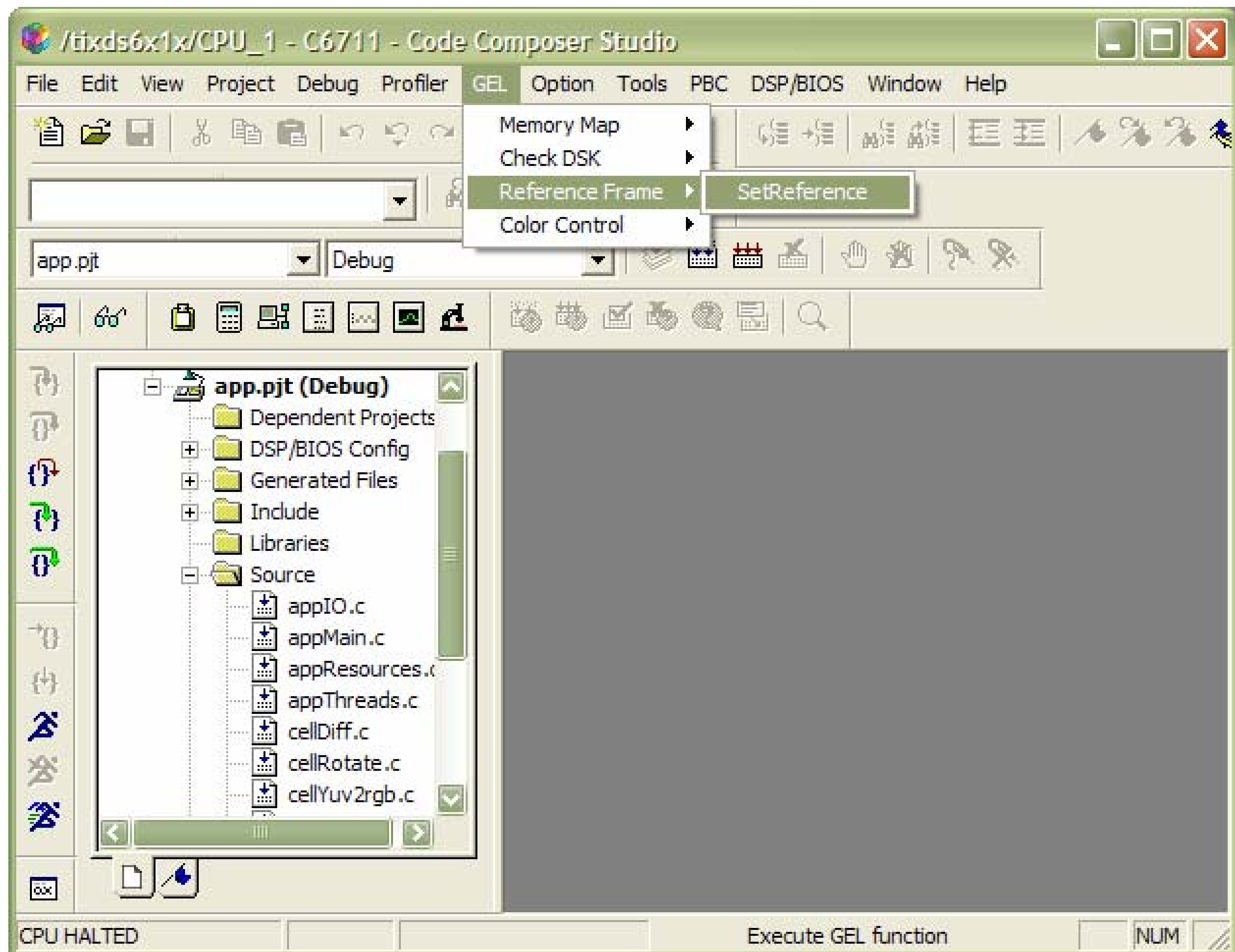

Figure 12. SetReference Menu Option via GEL

Our second control message is to change the color we apply to the unequal pixels. This is done with the text box shown in Figure 13.

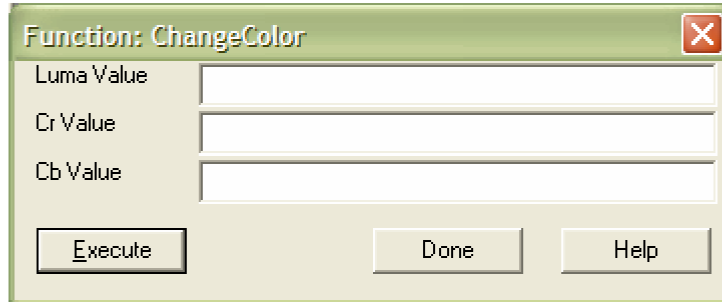

Figure 13. Text Box to Change Color via GEL

5.5.2 thrControl

Moving to the next portion of Figure 11, we look at the thrControl thread. In this thread we use an infinite loop to continuously monitor the values of the ExternalControl structure. If a value changes, we post a message to a DSP/BIOS mailbox object.

```
// Main loop
while (TRUE) {
    //See if user requested new reference frame
    if (externalControl.SetReference)
    {
        txMsg1.cmd = MSGNEWREFERENCE;
        txMsg1.arg1 = 0;
        txMsg1.arg2 = 0;
        MBX_post(&mbxProcess, &txMsg1, 0);

        //Reset SetReference to FALSE
        externalControl.SetReference = 0;
    }
    //See if user requested new color
    //We encode (Cr,Cb) pair into one value so that
    //we do not have to post 3 messages
    if ((externalControl.yValue != externalControlPrev.yValue) ||
        (externalControl.crValue != externalControlPrev.crValue) ||
        (externalControl.cbValue != externalControlPrev.cbValue))
    {
        txMsg2.cmd = MSGNEWCOLOR;
        txMsg2.arg1 = externalControl.yValue;
        txMsg2.arg2 = (externalControl.crValue << 8) + externalControl.cbValue;
        MBX_post(&mbxProcess, &txMsg2, 0);
    }

    externalControlPrev = externalControl;

    // suspend self for 100 ticks, and then poll again
    TSK_sleep( 100 );
}
```

NOTE: The values MSGNEWREFERENCE and MSGNEWCOLOR are defined in thrProcess.h.

If no message needs to be sent, then the thread sleeps for 100 ticks and starts over.

5.5.3 thrProcess

The process thread is continuously processing frames of data. Each time we are about to handle another frame, we check the DSP/BIOS mailbox object to see if there is a message from the Control thread. If a message is waiting, we process it accordingly and continue with handling the next frame.

```
while (TRUE)
{
    checkMsg();
    ...
    //Update Reference Frame if necessary
    if (thrProcess.diffEnv->SetReference == TRUE)
    {
        prevYId = DAT_copy2d(DAT_2D1D, (Void *) scombufCap->bufYCRCB[0], (Void *)
                prevY, PROCF_WIDTH, PROCF_HEIGHT, PROCF_WIDTH);
        prevCrId = DAT_copy2d(DAT_2D1D, (Void *) scombufCap->bufYCRCB[1], (Void *)
                prevCr, PROCF_WIDTH>>1, PROCF_HEIGHT>>1, PROCF_WIDTH>>1);
        prevCbId = DAT_copy2d(DAT_2D1D, (Void *) scombufCap->bufYCRCB[2], (Void *)
                prevCb, PROCF_WIDTH>>1, PROCF_HEIGHT>>1, PROCF_WIDTH>>1);

        thrProcess.diffEnv->SetReference = FALSE;
    }
    DAT_wait(prevYId);
    DAT_wait(prevCrId);
    DAT_wait(prevCbId);

    //Process Data
    ...
}

static Void checkMsg()
{
    CtrlMsg rxMsg;
    Int index;

    // check message in "mbxProc"
    while( MBX_pend( &mbxProcess, &rxMsg, 0) ) {
        switch (rxMsg.cmd) {
            case MSGNEWREFERENCE: //setting a new reference frame
                thrProcess.diffEnv->SetReference = TRUE;
                break;

            case MSGNEWCOLOR:     //change color of unequal pixels
                for (index = 0; index < (NUMDIFFCHANS + NUMCOMBOCHANS); index++)
                {
                    thrProcess.diffEnv[index].yValue = rxMsg.arg1;
                    thrProcess.diffEnv[index].crValue = rxMsg.arg2 >> 8;
                    thrProcess.diffEnv[index].cbValue = rxMsg.arg2 % 256;
                }
                break;
            default:
                break;
        }
    }
}
```

We can see that we use the DIFF_Env structure to pass the message into a cell parameter. The DIFF cell uses the values modified by checkMsg and applies them to the algorithm.

After the call to checkMsg in the while loop, we reset the reference frame if needed. By sending a “Set Reference” message during control thread initialization, we make sure that the prevY, prevCr, and prevCb buffers are initialized the first time the loop is run.

With this simple message-passing format, we are able to create any kind of message we desire and pass it into any cell we desire.

6 Conclusion

This motion detection system provides a taste of the type of applications to which RF5 can be adapted. The idea of displaying multiple channels on one monitor is very applicable to security systems. In particular, the motion detection algorithm gives an idea of what an “intruder detection” system might look like.

Using the guidelines described in this document, the RF5 NVDK adaptation provides a quick and systematic way to develop video-based applications that use DSP/BIOS and XDAIS. In particular the features of RF5 that enable such seamless development are:

- **CHAN.** Objects allow us a simple way to define, instantiate, and execute a large number of unique user-defined channels.
- **ICELL.** Objects provide a way to incorporate cells into any channel and execute them in any order desired. Their “lift and place” compatibility makes them a robust design tool.
- **Processing Frame Size.** All the necessary adjustments are made in a single file. You can change from CIF to QCIF or NTSC to PAL in a matter of seconds. All buffer declarations are taken care of since changes in that file, appThreads.h, propagate throughout the code.
- **Control Messages.** These give us the opportunity to make our system more robust. Passing messages from the host into a cell allows us the flexibility of making changes to our system on the fly.

7 References

This application note assumes that the reader has some knowledge about Reference Framework 5 and its NVDK adaptation. The following application notes provide information on these topics. They are available at http://www.ti.com.

1. *Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High-Density System* (SPRA795).
2. *A Video, Audio, Networking System on the C64x NVDK Using eXpressDSP RF5* (SPRA844).
3. *An RF5 Adaptation for IDMA2-Based Algorithms: A JPEG Example* (SPRA842)

Documentation for the NVDK is available with Ateme’s development software mentioned in Section 2.2.

IMPORTANT NOTICE

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

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

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

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

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

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

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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

Copyright © 2003, Texas Instruments Incorporated