An RF6 JPEG Decoder Adaptation on the OMAP5910 Processor

Rishi Bhattacharya, Vincent Wan
Niclas Anderberg, Alan Campbell

ABSTRACT

The Reference Framework 6 (RF6) software provides a fast ramp to general purpose processor (GPP) and DSP application development on the OMAP5910 dual-core processor. This RF6 JPEG adaptation for the OMAP5910 Innovator development platform provides starterware with which one can develop GPP based video specific applications that utilize the C55x DSP core.

This application uses eXpressDSP-compliant JPEG decoder and post-processing algorithms to demonstrate DSP-based JPEG decoding under the Linux operating system on the OMAP5910 processor. The target audience is anyone who wishes to develop a video-based application using RF6.

Contents

1  Overview ................................................................. 2
2  System Requirements .................................................. 3
   2.1  Hardware Requirements ........................................... 3
   2.2  Software Requirements .......................................... 5
3  Directory Structure .................................................... 6
   3.1  GPP-Side Directory Structure (Linux Workstation) ........... 6
   3.2  DSP-Side Directory Structure (Windows Workstation) ........ 7
4  Installing, Building, and Running the RF6 JPEG Application ........ 8
   4.1  Installing the Software .......................................... 8
        4.1.1  Linux Workstation ......................................... 9
        4.1.2  Windows Workstation ....................................... 9
   4.2  Build Procedure .................................................. 10
        4.2.1  Linux Workstation ......................................... 10
        4.2.2  Windows Workstation ....................................... 11
   4.3  Running the Application in MontaVista Linux .................. 11
5  RF6 JPEG Application Technical Overview .......................... 12
   5.1  Application Behavior Overview .................................. 12
        5.1.1  DSP Processing Threads .................................... 14
        5.1.2  GPP Data Processing Threads ............................... 14
        5.1.3  GPP-DSP Communication .................................... 14
   5.2  Detailed Data Flow ............................................... 15
6  System Implementation ............................................... 16

Trademarks are the property of their respective owners.
An RF6 JPEG Decoder Adaptation on the OMAP5910 Processor

1 Overview

This application note describes a JPEG Decoder adaptation with Reference Framework 6 (RF6) on the OMAP5910 dual-core processor. In this application, a JPEG file is passed from the GPP to the DSP via DSP/BIOS Link. After the DSP decodes the JPEG image, it uses the DSP DMA to send the resultant RGB output buffer directly to the LCD’s framebuffer. The adaptation supports QCIF resolution (176 X 144) images in the grey scale, YUV 4:2:0, YUV 4:2:2, and YUV 4:4:4 color subsampling formats.

RF6 is designed for multichannel and multi-algorithm applications in a GPP-DSP environment. For a detailed description of RF6, please see Reference Frameworks for eXpressDSP Software: RF6, A DSP/BIOS Link-Based GPP-DSP System (SPRA796).

This JPEG decoder adaptation uses many features of RF6. The highlights are as follows:

- **MSGLINK.** Once the DSP has completed decoding the JPEG file and has used the DMA to send the RGB output buffer to the LCD’s framebuffer, the pre-processing thread makes use of the MSGLINK messaging module in RF6 to signal the GPP that the decoding is complete.

- **JPD Integration.** RF6 lets you easily integrate XDAIS algorithms. We will not detail the actual development of the algorithm; rather, we discuss the implementation of an existing algorithm. For more information on XDAIS, please see TMS320 DSP Algorithm Standard Rules and Guidelines (SPRU352). In RF6, existing XDAIS algorithms are integrated by wrapping them with a “cell” and placing them in the application code. This application note demonstrates how to integrate the JPD (JPEG Decoder) algorithm in this manner.

- **Cell Environments and Inter-Cell Communication.** Since the JPD and JPSP XDAIS algorithms are closely related, cell environments are used to share data between the two...
algorithms. The proper procedure for implementing the JPD and JPSP cell environments to allow for this communication is shown.

- **Creating a Processing Channel.** This adaptation is implemented as a single channel system. The channel as two cells (JPD and JPSP).

- **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 two different cells:
  - **JPD.** Decodes the JPEG file and outputs YUV data in raster scan format.
  - **JPSP.** Takes the raster scan YUV data as input and converts it to RGB16 format for output to the Innovator LCD screen.

2 System Requirements

In order to successfully install, build, and run the RF6 JPEG Decoder application, the development environment (on the Windows PC and Linux host machine) must be set up correctly. It is therefore essential to precisely define the minimum system requirements for running the RF6 JPEG application.

Details regarding each of the components in the development environment can be found in the documents referenced in *Reference Frameworks for eXpressDSP Software: RF6, A DSP/BIOS Link-Based GPP-DSP System* (SPRA796). For example, instructions for installing DSP/BIOS Link in MontaVista Linux are available in the *DSP/BIOS Link for MontaVista Linux (MVL) User’s Guide.*

2.1 Hardware Requirements

- Figure 1 shows the typical hardware setup used for the provided implementation of RF6 on the Innovator5910 OMAP platform.
This hardware setup consists of the following platforms:

- **Windows Development/Debug Host.** This machine should be a PC running Microsoft Windows NT, 2000, or XP. It is used to build, run, and debug the DSP-side content using Code Composer Studio. The machine must have a free COM port for terminal emulation.

- **Red Hat Linux Development/Debug Host.** This may be either a dedicated Linux host workstation or a Linux Virtual machine running under Windows. It should run Red Hat Linux version 7.2 or greater. VmWare (www.vmware.com) is one such package that has been successfully used with RF6. This host is where you build the GPP-side of your RF6-based applications. The Ethernet connection allows the Innovator to boot up into the Linux kernel and mount its root filesystem—for example, using the Network File System (NFS). At production time, you will likely put the target filesystem, including the kernel and necessary utilities (for example, ls and pwd) into Flash memory. This enables a standalone boot, eliminating the need for Ethernet.

- **Innovator5910 EVM ES 2.6 or greater.** This board contains both a ARM925TDMI processor and a TMS320C55x DSP. The version specified has Rev C or greater of the processor module (OMAP1510C) and Rev D or greater of the Innovator Ethernet Break-out-Board.

The hardware connections shown in Figure 1 are as follows:

- **JTAG Emulator (e.g. XDS510, XDS560, XDS510PP+).** Connects the DSP to the PC running Windows for connection to Code Composer Studio.
• **Ethernet cable.** Connects the Innovator board to the Linux file system.
• **9-pin null modem serial cable.** Connects the GPP to the PC running a terminal emulator.

### 2.2 Software Requirements

The following software should be installed on the platforms used for RF6:

**• Windows Development/Debug Host.**
- Microsoft Windows NT, 2000, or XP
- Code Composer Studio for OMAP v2.20.18 or greater
- DSP/BIOS Link v1.02
- A terminal emulator, for example Tera Term Pro 2.3 or greater. Tera Term is a freeware terminal emulation program with the ability to transfer files in binary format. It is available at [http://hp.vector.co.jp/authors/VA002416/teraterm.html](http://hp.vector.co.jp/authors/VA002416/teraterm.html).

**• Red Hat Linux Development/Debug Host.**
- Red Hat Linux version 7.2 or greater
- MontaVista Linux Professional v3.0 or greater, which includes the Linux kernel, bootloader, etc. This software officially supports Red Hat Linux v7.2.
- The latest recommended patches from the MontaVista Zone (http://support.mvista.com). These range from kernel patches to the latest TI Innovator Kit Linux Support Package (LSP).
- Network services such as telnetd, ftpd, and nfsd should be configured and available.
- DSP/BIOS Link v1.02

**• Innovator5910 EVM ES 2.6 or greater.**

For detailed instructions on how to install RF6, please refer to Section 4 of *Reference Frameworks for eXpressDSP Software: RF6, A DSP/BIOS Link-Based GPP-DSP System* (SPRA796).
3 Directory Structure

3.1 GPP-Side Directory Structure (Linux Workstation)

Figure 2 shows the GPP-side folders for RF6 and highlights some important files they contain.

Folders to notice include:
- `/apps/rf6/Linux/OMAP/JPEG`. Contains the source files and the makefile that builds both debug and release versions of the rf6_gpp_jpeg GPP-side application.
- `/bin/Linux/OMAP/dsplink`. Contains the linkcfg.txt file needed to rebuild the DSP/BIOS Link driver.
- `/bin/rf6/Linux/OMAP`. Contains the rf6_gpp_jpeg GPP-side application executable, in both debug and release versions.
- `/test_images`. Contains JPEG test images that can be used with the RF6 JPEG Application.
3.2 DSP-Side Directory Structure (Windows Workstation)

Figure 3 shows the DSP-side folders for RF6 and highlights some important files they contain.

![Diagram of Windows Folder Hierarchy]

**Figure 3. Windows Folder Hierarchy**
Folders to notice include:

- **apps\rf6.** The root folder for the RF6 JPEG DSP application.
  - **appConfig.** Contains DSP/BIOS TextConf scripts that are generic to all platforms. These scripts are imported by the board-specific appcfg.tcf file.
  - **cells.** Contains JPD and JPSP cell implementation code.
  - **projects.** Contains platform-specific files for the RF6 JPEG Decoder DSP application. This includes Innovator board specific configuration files, project files and linker command files.
  - **threads.** Contains hardware-independent source files for the threads.
- **include.** Contains a number of public header files used by the RF6 JPEG Decoder application.
- **lib.** Contains the JPD and JPSP algorithm libraries and the OMAP5910 CSL library. Note that all the libraries are in the 55x large data model (.l55l) format.

## 4 Installing, Building, and Running the RF6 JPEG Application

This section describes how to install, build, and run the RF6 JPEG Decoder application, which decodes a JPEG image and displays it on the Innovator LCD screen.

It is assumed that you have created a setup that fulfills all the prerequisites described in section 2, *System Requirements*. These instructions are specific to the following setup:

- Innovator 5910 platform
- MontaVista Linux Professional 3.0 or greater on the Linux host machine
- Code Composer Studio 2.20.18 or greater on the target debug host

It is also assumed that you are familiar with the documents listed in section 3.3, *Recommended Reading*, of *Reference Frameworks for eXpressDSP Software: RF6, A DSP/BIOS Link-Based GPP-DSP System* (SPRA796).

### 4.1 Installing the Software

Begin by downloading the RF6 JPEG Decoder code distribution files from the Linux OMAP5910 Extranet site at http://www.ti.com/linuxomap5910. The Reference Framework Level 6 code package is delivered as two separate files:

- **rf_dsp_jpeg.zip.** A zip file containing the DSP-side source code for the RF6 JPEG application and the Reference Framework modules. This should be unzipped on the Windows development machine.
- **rf_gpp_jpeg.tar.gz.** A tar file containing the GPP-side minimal source code for the RF6 JPEG application and associated modules. This should be unzipped on the Linux host machine.

Why is the package split in this fashion? The primary reason is that the home for each of these deliverables may be quite different. The DSP-side content typically sits on a PC running Windows while the GPP-side code is on an NFS mount of a Linux host. These two file systems may be entirely separate.
4.1.1 Linux Workstation

1. Make a directory with a name of your choice:
   
   ```
   [>] cd ~/montavista
   [>] mkdir -p xyzdir
   ```

2. Go into the new directory:
   
   ```
   [>] cd xyzdir
   ```

3. Untar the GPP-side code in a directory of your choice:
   
   ```
   [>] tar xvzf rf_gpp_jpeg.tar.gz
   ```

4.1.2 Windows Workstation

1. If you have not already done so, install Code Composer Studio for OMAP 2.20.18 or a later version. It is recommended that you have the latest version of the Code Composer Studio software, as it may contain important features or problem fixes.

2. Unzip the DSP-side code. A suggested location for these files is the `$INSTALL_DIR\myprojects` folder. `$INSTALL_DIR` stands for the Code Composer Studio installation directory, which is `c:\ti` by default.

   Make sure to use directory names when you unzip the file. You may need to enable an option called something similar to "Use folder names" in your zip utility.

   Do *not* extract the zip file into a directory with a path that contains spaces such as `c:\Program Files`. Spaces in directory paths are not currently supported by the TI Code Generation Tools.

   **NOTE:** The top-level folder of the Reference Frameworks DSP-side distribution is called "referenceframeworks". The full path to this folder is called `RF_DIR` in this application note. If you unzip the distribution as suggested in `c:\ti\myprojects`, `RF_DIR` is `c:\ti\myprojects\referenceframeworks`.

3. If you have a version of DSP/BIOS Link newer than v1.02, you might want to point RF6 to the newer version of the DSP/BIOS Link DSP-side library (for example, `dsplink.l55l`). There are two ways to accomplish this:

   - Copy the newer DSP/BIOS Link DSP-side archives (for example, `dsplink.l55l`) to the Reference Frameworks folders. For example, you might copy the archives from `c:\dsplink\dsp\lib\debug\` to `RF_DIR\lib\debug`.

   - Change the linker command file (`link.cmd`) to point to the new archive. For example:

     ```
     -l c:\myNewDSPLink\dsp\lib\debug\dsplink.l55l
     ```

   The RF6 JPEG decoder applications includes the DSP-side archives of DSP/BIOS Link to allow you to build the application out of the box. There is currently no well-established way to define symbolic links in Code Composer Studio.

4. Additional steps that are necessary for debugging RF6 JPEG application are outlined in section 4.5 of SPRA796 and section 7 of the *Playwave Application using DSP/BIOS Link*. Note that this setup is not required for running the application.
4.2 Build Procedure

This section describes how to build both the DSP-side and GPP-side RF6 base applications.

4.2.1 Linux Workstation

After you untar the Linux package, you can build the GPP-side of the RF6 JPEG application.

1. Using the linkcfg.txt file found in the /bin/Linux/OMAP/dsplink folder, rebuild the DSP/BIOS Link GPP side kernel module and library. Instructions for doing this are in the DSP/BIOS Link for MontaVista Linux (MVL) User’s Guide (LNK_022_USR). Changes to the linkcfg.txt file include the following:
   - Increasing the maximum buffer size from 16KB to 48 KB. This allows for large images to be sent in a single data transfer.
   - Mapping the LCD’s framebuffer memory location to DSP external memory space via the DSP MMU. In the Montavista Linux Pro 3.0 distribution, the LCD’s framebuffer is placed in the internal SRAM on the OMAP5910 processor. This mapping will have to be modified if you have changed the default setting.

2. Open Rules.make using your favorite text editor. The Rules.make file at the top level of the folder hierarchy is a central place to specify paths. It is used instead of scripts to avoid different shell (ksh, csh, bash, ...) nuances.

3. Modify the DSPLINK_DIR path to point to your DSP/BIOS Link installation. The local makefile for the RF6 JPEG GPP application defines $(RF_ROOT_DIR) to point to the top-level rf_gpp directory (where Rules.make is located). For example:

   DSPLINK_DIR := $(RF_ROOT_DIR)/dsplink

4. Verify that the path to the cross compiler tools is correct. This ensures that the compiler, linker and archiver can be found. For example:

   CROSS_COMPILER_DIR := /opt/hardhat/devkit/arm/920t_le/bin

5. In apps/rf6/Linux/OMAP/jpeg, build the GPP application with the following command:

   $ make

   The GPP application should build without errors. Both debug and release versions should be generated as a result.

   If the build fails, it may be because the dsplink library was not built or the Rules.make DSPLINK_DIR variable was not set to point to the Link GPP code. Remember that the RF6 JPEG adaptation does not ship a prebuilt dsplink library and kernel module on the GPP-side to ease potential versioning issues.

6. Copy the resulting executable, rf6_gpp_jpeg, from /bin/rf6/Linux/OMAP/JPEG/Debug to the NFS-mounted target file system. Use a directory of your choice. (You will put the DSP-side application in the same directory.) For example type the following command at a single command prompt:

   $ cp ~/montavista/xyzdir/rf_gpp_jpeg/bin/rf6/Linux/OMAP/jpeg/Debug/rf6_gpp_jpeg ~/montavista/filesys/home/mynname/rf6jpegdir/
4.2.2 Windows Workstation

After you unzip the Windows package, you can build the DSP-side of the RF6 application.

1. Use the timake command to rebuild the DSP-side application from the Windows command prompt:

   ```bash
   [%] cd c:\ti\myprojects\referenceframeworks\apps\rf6\projects\innovator1510
   [%] timake app.pjt DEBUG
   ```

   **NOTE:** You must first run `<$(INSTALL_DIR)\dosrun.bat` so that the timake command and the TI Code Generation Tools are found.

2. Copy the resulting DSP-side application (app.out) from apps\rf6\projects\target\debug to the target’s file system, in the same directory where the GPP-side application is located. (For example, to ~/montavista/filesys/home/myname/rf6jpegdir.)

4.3 Running the Application in MontaVista Linux

This section assumes you have already set up the MontaVista Linux operating system on the target platform. On the NFS-mounted target file system, follow this procedure, which assumes you copied both the GPP-side and DSP-side RF6 JPEG applications to /home/myname/rf6jpegdir:

1. Copy the following files to the /home/myname/rf6jpegdir directory using Linux:
   - The dsplink load/unload script from your DSP/BIOS Link installation.
   - The dsplinkk.o Linux kernel module you built in step 8 (choose either the debug or release version).

   For example:

   ```bash
   [%] cd /home/myname/rf6jpegdir
   [%] cp ~/montavista/dsplink/etc/target/scripts/Linux/dsplink .
   [%] cp ~/montavista/dsplink/gpp/export/BIN/Linux/OMAP/RELEASE/dsplinkk.o .
   ```

2. Reboot the Innovator and then start the Linux kernel via the bootloader.

3. At the prompt of the Innovator file system, enter the following commands:

   - Go to the directory containing the application:
     ```bash
     [%] cd /home/myname/rf6jpegdir
     ```

   - Ensure that the appropriate file permissions have been set to load the various modules and execute RF6.
     ```bash
     [%] chmod -R u+rwx *
     ```

   - Load the DSP/BIOS Link kernel module. (Do not forget the trailing dot, which is an argument specifying the directory location of the DSP/BIOS Link kernel module.)
     ```bash
     [%] ./dsplink load .
     ```

   - Disable the power-down feature of the Innovator LCD by using the following command:
     ```bash
     [%] setterm –blank 0 > /dev/tty0
     ```

   - Run the rf6_gppjpeg user-mode process, which loads the DSP executable app.out and starts data streaming through DSP/BIOS Link.
     ```bash
     [%] ./rf6_gppjpeg app.out <image_name.jpg>
The last parameter is the name of the jpeg image you wish to display. For your convenience, a number of test images are provided in the test_images directory in the gpp side release. Please copy these over to the directory containing the application.

4. Figure 4 shows the directory contents of the /home/mynname/rf6jpegdir and the output of the application once it is executed:

Once the application has finished, you will see the decoded image on the Innovator LCD screen.

5 RF6 JPEG Application Technical Overview

This section gives an overview of the RF6 JPEG Application. Included are an overview of the application behavior and the module hierarchy.

5.1 Application Behavior Overview

In the RF6 JPEG application, the GPP sends a QCIF resolution (176X144) JPEG image to the DSP, which then decodes the image, and displays the image by sending the decoded output buffer directly to the LCD’s framebuffer. The adaptation supports QCIF resolution (176X144) images with YUV 4:2:0, YUV 4:2:2 and YUV 4:4:4 and grayscale color subsampling formats.

Figure 5 shows the overall processing flow in the RF6 JPEG application.
JPEG decompression uses Huffman decoding, which is inherently variable length. There is no concept of packet or slice in the standard, and consequently a fixed input buffer size is not possible.

In order to accommodate this variable input data request from the JPEG decoder algorithm, the application sends the entire encoded JPEG image file to the DSP via DSP/BIOS Link. Since this image file is quite large, it is kept in an external memory buffer on the DSP. This allows for the placement of critical code and data sections of the DSP application in the 55x DSP’s internal memory, which speeds up the application’s execution.

A pre-processing task is responsible for feeding the data requests from the JPEG decoder in the process task. This task copies the amount of data requested by the algorithm from the external memory buffer mentioned above into a smaller input buffer located in the DSP’s internal memory. Again, this is done to speed up application execution.

For processing task 0, the application takes the input data supplied by the pre-processing task, and applies the JPD (JPEG decoder) and JPSP (JPEG post-processing) algorithms to it. The JPD algorithm decodes the QCIF resolution JPEG image 11 MCUs (minimum coded units) at a time (equivalent to 4 lines of an image that uses YUV 4:4:4 color subsampling and 16 lines of an image that uses YUV 4:2:0) and outputs YUV data in raster scan format. The JPSP algorithm then takes this data and converts it to the RGB16 format, and stores the resultant output buffer in a buffer located in external memory. The process task then sends a message back to the pre-processing task (via SCOM) with the amount of new data it requires.
After the entire image has been decoded in this manner, processing task 0 uses the DSP DMA on the OMAP5910 processor to send the entire output frame directly to the LCD's framebuffer, which is located in the internal SRAM. This is done to avoid the additional overhead that would be caused by sending the output buffer back to the GPP and using the OS' framebuffer driver to display the image. Finally, the status task sends a message (via MSGLINK) to the GPP to indicate that the decoding is complete and the image has been displayed on the LCD.

The subsections that follow describe portions of the RF6 application in further detail.

5.1.1 DSP Processing Threads

The DSP side of the RF6 application is divided into the following threads. This list reflects roughly the sequence in which threads are performed. (The status threads and processing threads may be interspersed.)

- **thrPreProcess.** The pre-processing task for thrProcess0. This task runs the thrPreProcessRun() function. In this application, this task receives the entire JPEG image file from the GPP (using SIO APIs) and stores it into an external memory location. After this is accomplished, this task is responsible for responding to data requests from the JPD algorithm in the thrProcess0 task.

- **thrProcess0.** This task executes the cells defined for each channel. It runs the thrProcess0Run() function. In this application, this task contains a single channel which contains cells for the JPD and JPSP algorithms.

- **thrStatus.** The status thread on the DSP. This task runs the thrStatusRun() function, which informs the GPP via the MSGLINK module when the image has been decoded and displayed on the LCD.

5.1.2 GPP Data Processing Threads

The GPP side of the RF6 application is divided into the following thread:

- **rf6_gpp_jpeg.** The RF6 JPEG GPP process. This is a Linux process, which is responsible for:
  - Calling the DSP/BIOS Link PROC module APIs to bootload the DSP with the app.out executable
  - Reading the jpeg file (stored on the host filesystem) and sending it to the DSP via DSP/BIOS Link CHNLs.
  - Waiting for the DSP to send a message via MSGLINK to signal the completion of the decoding and display of the image on the LCD.

5.1.3 GPP-DSP Communication

The RF6 JPEG application implements data communication between the GPP and DSP. In addition, a status channel passes messages from the DSP to the GPP. You can modify this behavior as needed when creating your own application.

The encoded JPEG data is passed from the GPP to the DSP via DSP/BIOS Link CHNLs. Status messages are passed from the DSP to GPP via the MSGLINK module.
5.2 Detailed Data Flow

A combined view of the data flow in the RF6 JPEG application looks as shown in Figure 7.
6 System Implementation

6.1 GPP Side Implementation

The GPP side implementation consists of a single linux process, rf6_jpeg_gpp, which handles bootloading the DSP and sending the JPEG image to the DSP. After this has been completed, it waits for a message from the DSP to signal that the image has been decoded and displayed on the Innovator LCD. The following sections will describe these parts of the RF6 JPEG GPP side application.

6.1.1 DSP/BIOS Link Initialization

We start out by looking at the section of the GPP process that handles the initialization of the DSP, loading the app.out executable and starting the execution (i.e. releasing the DSP from reset). This sequence is common to all RF6 and DSP/BIOS Link based applications, and is documented in detail in the DSP/BIOS Link for MontaVista Linux (MVL) User’s Guide.

```c
status = PROC_Setup();
if ( !DSP_SUCCEEDED(status) ) {
    terminateAppl( "Error: Failed PROC_Setup()", APPLSTATE0 );
}
status = PROC_Attach(ID_PROCESSOR, NULL);
if ( !DSP_SUCCEEDED(status) ) {
    terminateAppl( "Error: Failed PROC_Attach()", APPLSTATE1 );
}
status = PROC_Load(ID_PROCESSOR, argv[1], 1, loopback);
if ( !DSP_SUCCEEDED(status) ) {
    terminateAppl( "Error: Failed to load executable to DSP", APPLSTATE2 );
}
status = PROC_Start (ID_PROCESSOR);
if ( !DSP_SUCCEEDED(status) ) {
    terminateAppl( "Error: Failed to start the DSP", APPLSTATE2 );
}
...
```

Create and initialize the proc object.

Attach to the DSP process.

Load the DSP executable

Start execution on DSP.
6.1.2 Transferring JPEG Data to the DSP

Following the DSP initialization, we create an output data channel for sending data to the DSP and allocate the corresponding output buffer.

```c
ChannelAttrs chnlAttrOutput;
DSP_STATUS status;

chnlAttrOutput.mode      = ChannelMode_Output;
chnlAttrOutput.endianism = Endianism_Default;
chnlAttrOutput.size      = ChannelDataSize_16bits;

status = CHNL_Create( ID_PROCESSOR, CHNL_ID_OUTPUT, &chnlAttrOutput );

if ( !DSP_SUCCEEDED( status ) ) {
    terminateAppl( "Error: Failed CHNL_Create() of output channel", APPLSTATE4 );
}

status = CHNL_AllocateBuffer( ID_PROCESSOR, CHNL_ID_OUTPUT,
    (char **) outBufArray, BUFFERSIZE, NUMBUFS);

if ( !DSP_SUCCEEDED( status ) ) {
    terminateAppl( "Error: Failed CHNL_AllocateBuffer() on output", APPLSTATE5 );
}
```

We then read data from the JPEG file in the FillBuffer function and send the data to the DSP. Note that BUFFERSIZE is set to 48KB, which is the maximum DSP/BIOS Link buffer size we specified in `linkcfg.txt`. This allows us to send images that have file sizes up to 48KB in a single data transfer.

```c
for ( i = 0; i < NUMBUFS; i++ ) {
    ioReqOutput.buffer = (char *) outBufArray[i];
    ioReqOutput.size   = BUFFERSIZE;
    FillBuffer();
    status = CHNL_Issue ( ID_PROCESSOR, CHNL_ID_OUTPUT, &ioReqOutput );
    if ( !DSP_SUCCEEDED( status ) ) {
        terminateAppl( "Error: Failed to prime DSP stream", APPLSTATE6 );
    }
}
```

An RF6 JPEG Decoder Adaptation on the OMAP5910 Processor
Note that another alternative would be to increase the maximum buffer size supported by DSP/BIOS Link to the desired maximum input image file size, so that only one data transfer would have to take place.

### 6.1.3 Using MSGLINK to Signal Completion

Since the process by which the DSP decodes the image and displays it on the Innovator LCD is completely transparent to the GPP, the GPP will need to know when the image has been displayed (so that it may detach from the DSP and exit the application). For this reason, the MSGLINK messaging component of RF6 is used to receive a message from the DSP once the image is displayed. Note that the specified timeout is infinite — therefore the GPP will wait until the DSP sends a message back before exiting the application.

```c
... 
typedef struct ImageMsg {
    unsigned short cmd;
} ImageMsg;

ImageMsg msg;
...

MSGLINK_init();

if ( MSGLINK_create(MSGLINK_INPUT, sizeof(ImageMsg), WAIT_FOREVER) != MSGLINK_SUCCESS ) {
    terminateAppl( "Error: Failed to create message channel", APPLSTATE3 );
}
...

MSGLINK_recv( &msg );
...
```

**Initialize and create messaging channel to the DSP.**

**Receive message from DSP.**
6.2 DSP Side Implementation

6.2.1 Pre-Processing Thread Architecture

The pre-processing thread on the DSP receives the entire JPEG image file from the GPP (using SIO APIs) and stores it into an external memory location. After this is accomplished, this task is responsible for feeding data requests from the JPD algorithm in the thrProcess0 task. Figure 8 shows the architecture of the pre-processing thread.

![Pre-Process Thread Architecture Diagram]

Note that instead of sending the entire JPEG file over to the DSP and having the pre-processing thread “feed” the requested data to the JPD algorithm, it would also be possible to have the DSP send messages back to the GPP via MSGLINK with the amount of data requested by the algorithm, and have the GPP read the JPEG file and send over the requested data through DSP/BIOS Link.

While this alternative architecture would save external memory consumption, it would also increase the latency between calls to the decoder, and consequently slow the decoding process down considerably. It is for this reason that the architecture shown in Figure 8 was chosen.
6.2.2 Adding the JPD Algorithm and Integrating Cells

In this section we will describe the proper procedure for incorporating the JPD algorithm into our application. JPD is a TMS320C55x baseline-DCT compliant JPEG decoder algorithm. The features of the algorithm are as follows:

- support for Arbitrary Huffman tables
- supports YUV 4:4:4, YUV 4:2:2, YUV 4:2:0 and gray scale color format
- arbitrary image sizes (not supported in this implementation)
- fully compliant with the XDAIS specification

The /lib folder in the DSP side directory structure contains the library jpd_tif.l55l. We need to create a cell wrapper around this algorithm to incorporate it into RF6. Two things need to be done:

1. **Include jpd_tif.l55l.** Within the application’s linker command file (link.cmd), the following lines are added:

   ```
   * Algorithm JPD: bind the generic JPD symbol to TI’s implementation
   * of the algorithm, and include the appropriate library;
   */
   -l jpd_tif.l55l /* JPEG Decoder algorithm */
   _JPD_IJPD = _JPD_TIF_IJPD;
   ```

2. **Add appropriate source and header files.** Within the application’s linker command file (link.cmd), the following lines are added:
   - **cellJpd.c.** This is the main source file for running the JPD 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 <algrf.h>
   #include <jpd_tif.h>
   #include <yuvtodisplay.h>

   #include "ijpsp.h"
   #include "jpsp_tii.h"
   #include "rgbTables.h"
   #include "celljpd.h"
   #include "appThreads.h"

   ICELL_Fxns JPD_CELLFXNS = {
       NULL,             // jpdClose
       NULL,             // jpdControl
       JPD_cellExecute,  // jpdExecute
       JPD_cellOpen      // jpdOpen
   };
   ```
JPD_cellExecute is the main entry point for the application into the algorithm:

```c
/*
 * ======== JPD_cellExecute ========
 */
Bool JPD_cellExecute( ICELL_Handle handle, Arg arg )
{
    int status;
    long streamin_len, streamin_pos;

    IJPDFxns *jpdFxns = (IJPDFxns *)handle->algFxns;
    IJPDHandle jpdHandle = (IJPDHandle)handle->algHandle;
    MYINTYUVBUF * my_buf_ptr = (MYINTYUVBUF *)handle->outputIcc[0]->buffer;

    // Set interfacedec to point to buffers from ICC
    interfacedec->out.Y_ptr = (long)my_buf_ptr->ybuff;
    interfacedec->out.U_ptr = (long)my_buf_ptr->ubuff;
    interfacedec->out.V_ptr = (long)my_buf_ptr->vbuff;

    status = jpdFxns->JPGDecode(jpdHandle, my_jpd->interfacedec);

    *(my_jpd->MCUdecoded) += *(my_jpd->no_of_mcus);
    *(my_jpd->lastMCUdecoded) = *(my_jpd->CurrentMCU) +
    *(my_jpd->no_of_mcus);
    *(my_jpd->unMCUsleftInRow) = *(my_jpd->unMCUsleftInRow) -
    *(my_jpd->no_of_mcus);
    ...
```

This function sets up appropriate intermediate buffer addresses. The decoder algorithm is also called within JPD_cellExecute.

- `cellJpd.h`. In addition to the function prototype for any of the ICELL_Fxns used in the algorithm, this file contains the definition of the cell environment.

```c
Bool JPD_cellExecute( ICELL_Handle handle, Arg arg );
Bool JPD_cellOpen( ICELL_Handle handle );
```

```c
typedef struct JPD_Env {
    IJPDFxns * jpdFxns;
    IJPDHHandle jpdHandle;
    MYINTYUVBUF * my_buf_ptr;
    ...
    struct JPD_Env {
        IJPDFxns * jpdFxns;
        IJPDHHandle jpdHandle;
        MYINTYUVBUF * my_buf_ptr;
        ...
```
The JPD_Env structure holds information about the algorithms environment, e.g. the number of MCUs decoded, the number of words the algorithm will request in the next call, a flag to indicate completion of decoding, etc. We will discuss the cell environment in more detail in the following sections.

- **ijpd.h.** The following structures and objects are defined in ijpd.h:
  
  The IJPD_Obj:
  ```c
  typedef struct IJPD_Obj {
    struct IJPD_Fxns *fxns;
  } IJPD_Obj;
  ```
  
  The IJPD_Params structure:
  ```c
  typedef struct IJPD_Params {
    Int size;          /* must be first field of all params structures */
  } IJPD_Params;
  ```
  
  The IJPD_Fxns structure defining operations performed on JPD objects:
  ```c
  typedef struct IJPD_Fxns {
    IALG_Fxns ialg;    /* IJPD extends IALG */
    int (*JPGReset)(IJPD_Handle handle);
    int (*JPGDecode)(IJPD_Handle handle, IJPD_InterfaceStruct *cfg);
    int (*getStatus)(IJPD_Handle handle, IJPD_InterfaceStruct *cfg, IJPD_ImgParams *ret);
  } IJPD_Fxns;
  ```

  The incorporation of the JPSP algorithm is done in a similar fashion to the procedure shown above.

### 6.2.3 Creating the Processing Channel

This section describes how to create and use the desired channel structure as shown in Figure 7.

1. **Add the appropriate enumerations to thrProcess.h.** This step allows us to create any channel structure we like.

   The following enumeration describes the cells that our channel uses. Channel 0 has two cells (JPD and JPSP), and consequently the value of THRPROCESS0_NUMCELLS is 2.

   ```c
   // Enumeration of channel cells (so we use names instead of numbers for index)
   enum {
     THRPROCESS0_CELLJPD = 0,
     THRPROCESS0_CELLJPSP,
     THRPROCESS0_NUMCELLS
   };
   ```

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 the channel. The MYINTYUVBUF structure contains pointers to the intermediate buffers between the two cells.
3. **Open the channel.** The following code opens an instance of this channel type. We run a loop for the number of channels that have the same JPD-JPSP structure. Since there is only a single channel (i.e. THRPROCESS0_NUMCHANNELS = 1), the loop is only run once. The following code is located in the file thrProcess.c in the function setParamsAndStartChannels.

```c
for (chanNum = 0; chanNum < THRPROCESS0_NUMCHANNELS; chanNum++) {
    thrProcess0.cellList[(chanNum * THRPROCESS0_NUMCELLS) + THRPROCESS0_CELLJPD].algParams = (IALG_Params *)&jpdParams;
    thrProcess0.cellList[(chanNum * THRPROCESS0_NUMCELLS) + THRPROCESS0_CELLJPSP].algParams = (IALG_Params *)&jpspParams;

    UTL_logDebug1("Channel Number: %d", chanNum);

    // open the channel: this causes the algorithms to be created
    rc = CHAN_open(&thrProcess0.chanList[chanNum],
                   &thrProcess0.cellList[(chanNum * THRPROCESS0_NUMCELLS),
                                      THRPROCESS0_NUMCELLS,
                                      NULL);
    UTL_assert(rc == TRUE);
}
```

Set parameters for JPD and JPSP.

Open channel 0.
Execute the channel. We now look at the channel execution part of the application. This is contained in the main function of the process thread, called thrProcessRun, and is executed until the finished_decoding flag is set to 1 by the JPSP algorithm.

```c
// process the data
for( chanNum = 0; chanNum < THRPROCESS0_NUMCHANNELS; chanNum++ ) {
    CHAN_Handle chanHandle = &thrProcess0.chanList[ chanNum ];

    // Set the input ICC buffer
    ICC_setBuf(chanHandle->cellSet[THRPROCESS0_CELLJPD].inputIcc[0],
               scomMsgPreProcess->bufChannel, FRAMELEN );
    UTL_stsStart(stsTime1);  // start the stopwatch
    rc = CHAN_execute( chanHandle, NULL ); Execute channel 0.
    UTL_assert( rc == TRUE );
    UTL_stsStop(stsTime1);  // elapsed time goes to this STS
    scomMsgPreProcess->finished_decoding =
        (*(thrProcess0.jpdEnv.finished_decoding));
    scomMsgPreProcess->size =
        (*(thrProcess0.jpdEnv.unDataConsumed));
    scomMsgPreProcess->currentaddr =
        (long) (thrProcess0.jpdEnv.interfacedec->in.currentaddr);
}
```

Now that we have added everything necessary to create a custom channel, we next look at a few things that are needed to add the JPD and JPSP cells into the channel.

### 6.2.4 Adding Cells to the Processing Channel

Cells give us a convenient and consistent way to wrap an algorithm and insert it into RF6. The Channel Manager (CHAN) then uses the wrapper as an interface for efficient execution. In section 6.2.2 we described how to create the cell wrapper itself for an algorithm. Now, we describe how to integrate that cell into our processing channel (channel 0).

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

```c
// Definition of the structure describing the state of the thread.
typedef struct ThrProcess0 {

    JPD_Env jpdEnv;
    JPSP_Env jpspEnv;

    ... ThrProcess0;
}
```

The environment structure is used to store parameter, status, or any information about the algorithm that is needed on a persistent basis. These structures need to be initialized, and we do that inside the function thrProcessInit:
Note that many of the members of both cell environment structures are pointers to the same instance of a variable or structure. This is done to facilitate inter-cell communication between the JPD and JPSP cells, which is necessary because of the coupling that exists between the JPD and JPSP algorithms.

7 Conclusion

Reference Framework Level 6 (RF6) is intended to enable designers to create applications that make use of a General Purpose Processor (GPP) in addition to the DSP. Communication between the two processors can include both data streaming and control/status channels.

This RF6 based JPEG adaptation provides a taste of the type of applications to which RF6 can be adapted. Due to the presence of the iDCT hardware accelerator on the 55x DSP, the JPEG decode algorithm runs more efficiently on the DSP than on the GPP. Also, sending the decoded output data directly to the framebuffer from the DSP improves the performance of the overall system.

Using the guidelines described in this document, this adaptation provides a quick and systematic way to develop DSP based video-based applications that use DSP/BIOS, DSP/BIOS Link and XDAIS in a GPP+DSP environment.
8 References

An acquaintance with DSP/BIOS, XDAIS, and DSP/BIOS Link is needed for writing DSP applications at the RF6 level.

Complete documentation on DSP/BIOS can be found in *TMS320 DSP/BIOS User’s Guide* (SPRU423) and *TMS320C5000 DSP/BIOS API Reference Guide* (SPRU404). Documentation for XDAIS is provided in *TMS320 DSP Algorithm Standard API Reference* (SPRU360). Source code and documentation for DSP/BIOS Link can be found on the DSP/BIOS Link web page: http://www.ti.com/bioslink.

1. *Reference Frameworks for eXpressDSP Software: RF6, A DSP/BIOS Link-Based GPP-DSP System* (SPRA796)
2. *A Multichannel Motion Detection System Using eXpressDSP RF5 NVDK Adaptation* (SPRA904)

Web Resources

- TI DSPVillage: http://www.dspvillage.com
- Linux OMAP5910 Extranet site: http://www.ti.com/linuxomap5910
- MontaVista Zone: http://support.mvista.com
- DSP/BIOS Link web page: http://www.ti.com/bioslink
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:

<table>
<thead>
<tr>
<th>Products</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amplifiers</td>
<td>Audio</td>
</tr>
<tr>
<td>Data Converters</td>
<td>Automotive</td>
</tr>
<tr>
<td>DSP</td>
<td>Broadband</td>
</tr>
<tr>
<td>Interface</td>
<td>Digital Control</td>
</tr>
<tr>
<td>Logic</td>
<td>Military</td>
</tr>
<tr>
<td>Power Mgmt</td>
<td>Optical Networking</td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>Security</td>
</tr>
<tr>
<td></td>
<td>Telephony</td>
</tr>
<tr>
<td></td>
<td>Video &amp; Imaging</td>
</tr>
<tr>
<td></td>
<td>Wireless</td>
</tr>
</tbody>
</table>

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

Copyright © 2003, Texas Instruments Incorporated