De-Interlacing and YUV 4:2:2 to 4:2:0 Conversion on TMS320DM6446 Using the Resizer

Zhengting He, Senthil Natarajan

ABSTRACT

Video signals captured directly from charge-coupled device (CCD) cameras naturally have interlaced effects and are in a 4:2:2 interleaved format. They typically need to be converted to a 4:2:0 planar format before being encoded because most video compression standards accept input only in 4:2:0 format. It is better to reduce or remove the interlaced artifacts in the original video signal when feeding it to a progressive encoder because these artifacts degrade visual quality and increase video encoder loading.

This application report describes how to do simple de-interlacing and a YUV 4:2:2 to 4:2:0 color format conversion on the TMS320DM6446 using the resizer hardware through the use of two examples. The first one performs the de-interlacing operation on input video frames in 4:2:2 format at National Television System Committee (NTSC) standard definition resolution and generates de-interlaced output at the same format and resolution. The second one combines the de-interlacing operation with a 4:2:2 to 4:2:0 conversion to generate the 4:2:0 format output at a 4CIF resolution.

The purpose of this application report is to show how to use the existing hardware (the resizer on TMS320DM6446) to fully remove interlaced artifacts in the captured video signal and offload the 4:2:2 to 4:2:0 conversion task from the DSP. Because the resizer was not designed specifically for de-interlacing purposes, the de-interlacer being implemented uses a simple, non-ideal algorithm. The provided examples have their usages in applications when a perfect, yet expensive, de-interlacer is not required. This is especially true when DSP millions instructions per second (MIPS) need to be saved by offloading the 4:2:2 to 4:2:0 conversion and de-interlacing operations to some other hardware. In fact, such a method for achieving de-interlacing can at times provide quality that is at par or even better than some high-quality, high-complexity de-interlacing algorithms after video compression is taken into account.

Note: The source code released with this application report assumes that Codec Engine 1.x is being deployed. Memory_getPhysicalAddress() has become an unsupported Codec Engine API for CE 2.0 and higher. If you are using CE 2.x and higher, replace it with the more powerful, and supported Memory_getBufferPhysicalAddress() API.

This application report contains project code that can be downloaded from this link. http://www-s.ti.com/sc/techlit/sprc374.gz

Contents

1 Introduction 2
2 De-Interlacing and YUV 4:2:2 to 4:2:0 Conversion Algorithm 6
3 Implementation Details 8
4 Example User's Guide 16
5 References 17

List of Figures

1 YUYV Interleaved 4:2:2 Data 3
2 UYVY Interleaved 4:2:2 Data 3
3 YUV Planar 4:2:0 Data 4
4 Interlaced Effects 4
5 Resizer in TMS320DM6446 5
6 De-Interlacing 9
7 De-Interlacing + 4:2:2 to 4:2:0 Conversion: 1st Call for Y 12
8 De-Interlacing + 4:2:2 to 4:2:0 Conversion: 2nd Call for U 13

List of Tables

1 Package Content 16

1 Introduction

Video signal captured directly from CCD cameras has interlaced effects and is in a 4:2:2 interleaved format. It typically needs to be converted to a 4:2:0 planar format before being encoded because most video compression standards accept input only in a 4:2:0 format. Interlaced artifacts in the original video signal are better to be reduced or removed when feeding it to the encoder because those artifacts not only downgrade the video's visual quality but also increase the video encoder loading.

This application report describes how to use the resizer hardware in the TMS320DM6446 processor to do a 4:2:2 to 4:2:0 color format conversion and de-interlacing. Two examples are provided.

- The first one performs the de-interlacing operation on the input video frames in 4:2:2 format at NTSC standard definition resolution (NTSC SD) and generates de-interlaced output at the same format and resolution.
- The second one combines the de-interlacing operation with a 4:2:2 to 4:2:0 conversion to convert each 4:2:2 format, NTSC standard resolution input frame to an output frame in 4:2:0 format at 4CIF resolution (704×480).

The purpose of this application report is to show how to use an existing hardware module, the resizer in TMS320DM6446, to remove interlaced artifacts in the captured video signal and convert them to an encoder friendly format to save DSP MIPS. Because the the resizer was specifically designed for video resolution scaling purpose but not for de-interlacing, the de-interlacer being implemented here is simple from an algorithm perspective. It basically discards every odd row in the original video frame and re-interpolates it using neighboring even rows. The provided examples have their usages in the applications when a perfect yet expensive de-interlacer is not required but DSP MIPS need to be saved by offloading the 4:2:2 to 4:2:0 conversion and de-interlacing operations to some other hardware. One usage case might be applying it before a low bit rate encoder. In this case, a high-quality de-interlacer is overkill because much detailed information in the input video frame is lost after compression. For applications demanding a high-quality de-interlacer, more complicated algorithms need to be implemented by software.

In the rest of the section, some background knowledge is introduced. Section 2 describes the de-interlacing and 4:2:2 to 4:2:0 conversion algorithms. Section 3 describes the implementation details. Section 4 serves as the user's guide for the provided examples. Section 5 provides references.

1.1 Video Signal Captured From CCD Camera

The video frames captured from a CCD camera are in standard resolution. The American NTSC standard definition (NTSC SD) resolution is 720 pixels per row, 480 pixels per column, and 30 frames per second. The European PAL standard definition (PAL SD) resolution is 720 pixels per row, 576 pixels per column, and 25 frames per second. In this document, we assume the input video signal is in NTSC SD resolution.

The information for each pixel contains three components:

- Y, which is the luminance (luma) information
- Cb (U), which is the blue color information
- Cr (V), which is the red color information

Because human eyes are more sensitive to luminance but less sensitive to color information, the standards define that color components (chroma) are down-sampled by half horizontally in the captured video signal. Thus, for a NTSC SD CCD camera, each captured frame has 720×480 Y values, 360×480 U values, and 360×480 V values. Each value is 8 bits (a byte) in range [0, 255], which makes each NTSC SD frame (720+360+360) × 480 = 691200 bytes.

The Y/U/V components in the captured frame are typically interleaved. They usually are called YUV 4:2:2 interleaved format, or simply 4:2:2 format. Figure 1 and Figure 2 show two typical organizations, where $Y/U/V_{i,j}$ means the Y/U/V component on row i and column j. The examples shown in the document assume the input data are organized as Figure 2.

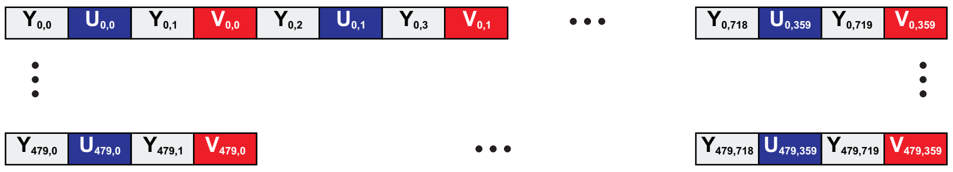
$Y_{0,0}$ $U_{0,0}$ $Y_{0,1}$ $V_{0,0}$ $Y_{0,2}$ $U_{0,1}$ $Y_{0,3}$ $V_{0,1}$
$Y_{0,718}$ $U_{0,359}$ $Y_{0,719}$ $V_{0,359}$
$Y_{479,0}$ $U_{479,0}$ $Y_{479,1}$ $V_{479,0}$
$Y_{479,718}$ $U_{479,359}$ $Y_{479,719}$ $V_{479,359}$

Figure 1. YUYV Interleaved 4:2:2 Data

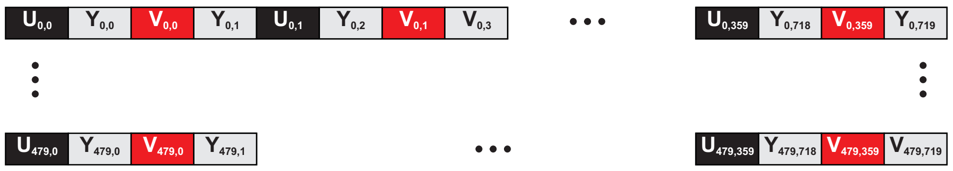
$U_{0,0}$ $Y_{0,0}$ $V_{0,0}$ $Y_{0,1}$ $U_{0,1}$ $Y_{0,2}$ $V_{0,1}$ $V_{0,3}$
$U_{0,359}$ $Y_{0,718}$ $V_{0,359}$ $Y_{0,719}$
$U_{479,0}$ $Y_{479,0}$ $V_{479,0}$ $Y_{479,1}$
$U_{479,359}$ $Y_{479,718}$ $V_{479,359}$ $V_{479,719}$

Figure 2. UYVY Interleaved 4:2:2 Data

1.2 YUV 4:2:0 Planar Data for Video Encoder

Most video compression standards require the input video frames to be in YUV 4:2:0 format. There are two distinctions between 4:2:2 interleaved data and 4:2:0 planar data.

- The chroma information is down-sampled vertically further by another half. That is, for each NTSC SD frame, U or V components each contain 360×240 bytes. Thus, for each NTSC SD frame in 4:2:0 format, the total size is 720×480 + 360×240×2 = 518400 bytes. The video compression standards require chroma to be down-sampled further so the encoders have less data to process while maintaining acceptable visual quality for human eyes.
- Efficient implementations of video compression standards further require that luma and chroma components are separated in memory because the encoding algorithms may process them in different ways.

Figure 3 shows the NTSC SD video frame in 4:2:0 planar format.

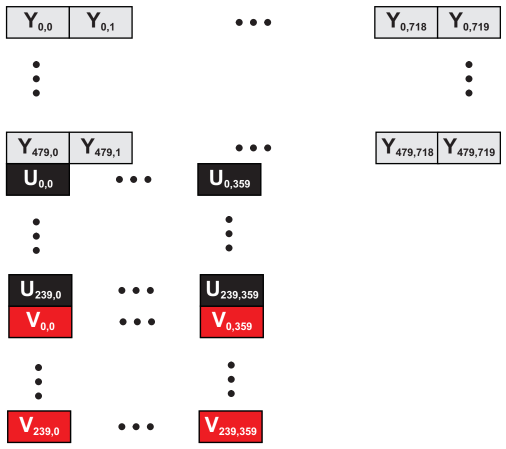

Figure 3. YUV Planar 4:2:0 Data

1.3 *Interlaced Artifacts*

When the NTSC/PAL standard was being defined, hardware and bandwidth constraints required individual frames to be captured as two separate shots. A video frame is divided in odd/even fields, and each shot captures the odd/even rows. The two fields are then merged together to form a complete frame. For NTSC standard where video frames are captured at 30f/s, the start time between two sequential shots is 16.67 ms. For PAL standard, the corresponding interval is 20 ms. Such a solution creates interlaced artifacts when there are fast motion activities in the video scenes to be captured.

Figure 4 shows how interlaced artifacts are created. Suppose in the video scene there is a rectangular object moving horizontally in the right direction. The odd field captures the rectangular at position 1. Later the even field captures the rectangular at the position that is slightly right-shifted compared to position 1. When these two fields are merged, the vertical edges of the rectangular are no longer smooth but show the saw-tooth effect. Such artifacts created by capturing a moving video object at different shots are called interlaced artifacts.

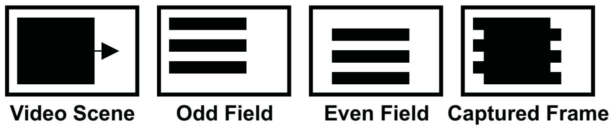

Figure 4. Interlaced Effects

Interlaced artifacts not only downgrade visual quality, but since they are represented as high-frequency noise, they prove to be particularly challenging for progressive video encoders.

- Human eyes are more sensitive to low-frequency information and much less sensitive to high-frequency information. Therefore, removing some high-frequency information in the original frame when encoding the video can maintain acceptable visual quality.
- In many cases, for each 16×16 or 8×8 block in a video frame, we can find a similar or even identical block in a neighboring frame because adjacent video frames contain correlated information. Therefore, when encoding a block in a frame, it is a good idea to find a similar or identical block in the previously coded frame, and code only the delta between them to achieve a high-compression ratio. In most video compression standards, a motion estimation (ME) module is defined for this purpose.

In the worst case, interlaced artifacts can appear in almost every 16×16 or 8×8 block in video frames capturing high amounts of motion activities. When encoding a block, such high-frequency noises can cause the progressive-based ME module difficulties or even failures in finding the similar block in the previous coded frame, which makes the delta bigger and increases the number of bits to encode it. To make it worse, many bits are wasted to encode the interlaced artifacts that are supposed to be removed. Therefore, it is a good idea to reduce or remove the interlaced artifacts in the captured frame before feeding it to a progressive video encoder.

1.4 Resizer Hardware in TMS320DM6446

There is a resizer hardware module in the video processing sub-system (VPSS) of the TMS320DM6446 processor for the purpose of resizing an uncompressed video frame. Figure 5 is the high-level block diagram of the resizer.

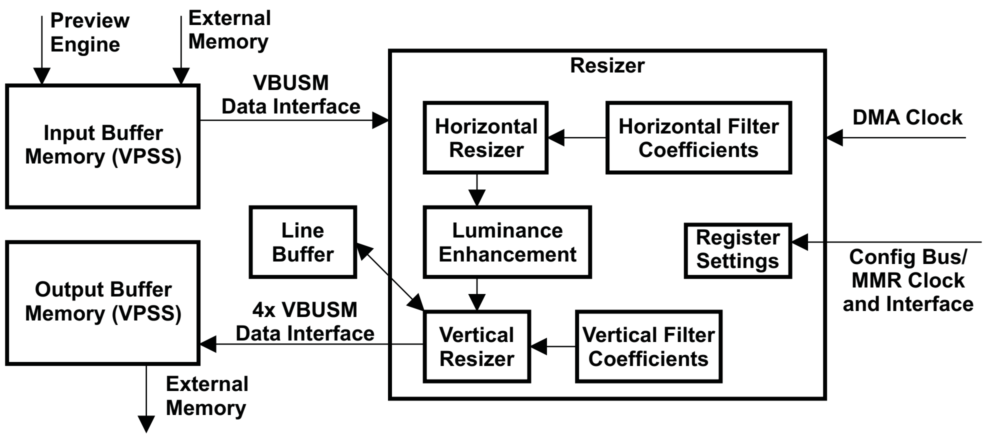

Figure 5. Resizer in TMS320DM6446

The resizer module can accept input image/video data from either the preview engine or external memory. The output of the resizer is sent to external memory. The resizer module is programmable via its registers that are accessible by a host processor (ARM core in TMS320DM6446). The following features are supported by the resizer module.

- Maximal output width of 1280 horizontal pixels
- Output width must be 32-byte aligned
- Support from 1/4× to 4× scaling in each direction (horizontal and vertical). The scaling factor is independent for each direction.
  - For 1/4x to 1/2x scaling, 4 7-tap filters are used that each filter represents phase 0, 0.25, 0.5, and 0.75, respectively. The total number of effective coefficients is 28.
  - For 1/2x to 4x scaling 8 4-tap filters are used that each filter represents phase 0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, and 0.875, respectively. The total number of effective coefficients is 32.
- All the filter coefficients are programmable.
- Programmable luminance sharping (enhancement).

The resizer performs horizontal resizing, then vertical scaling. Between these scalings, a luminance enhancement feature is optional. Because vertical adjacent pixels are not contiguous in physical memory, they are placed contiguously in a line buffer for processing to achieve higher efficiency.

The resizer needs to be configured correctly before performing image scaling. The following are important parameters for configuration purpose.

- in_hsize/out_hsize, which is the width (horizontal size) of the input/output frame. Due to hardware constraints, the scaling ratio is roughly but NOT exactly equal to out_hsize/in_hsize. That is, if in_hsize = 720 and out_hsize = 360, the actual scaling factor recognized by the resizer is not exactly equal to 360/720 = . To make exactly 1/2× scaling, in_hsize needs to be adjusted to 720 + delta. The exact value of delta is calculated by some internal mathematical equations in the resizer and is not explained in this document. However, it can be obtained using an offline utility program, which will be released with the production version of the resizer driver.
- in_vsize/out_vsize, which is the height (vertical size) of the input/output frame. Again, the scaling ratio is roughly but NOT exactly equal to out_vsize/in_vsize. The delta amount to adjust in_vsize can be calculated using the utility program provided with the resizer driver.
- in_pitch, which defines the start point of each row of the input frame. Let X represents the start address of row N, then the start address of row N+1 is X + in_pitch.
- inptyp and pix_fmt, which define the input frame format.
  - For 4:2:0 planar format, inptype = RSZ_INTYPE_PLANAR_8BIT and pix_fmt = RSZ_PIX_FMT_PLANAR.
  - For 4:2:2 interleaved format organized as Figure 1 (YUYV), inptype = RSZ_INTYPE_YCBCR422_16BIT and pix_fmt = RSZ_PIX_FMT_YUYV.
  - For 4:2:2 interleaved format organized as Figure 2 (UYVY), inptype = RSZ_INTYPE_YCBCR422_16BIT and pix_fmt = RSZ_PIX_FMT_UYVY.
- hfilter_coeffs[0:31], which define the horizontal filter coefficients.
  - For 1/4x to 1/2x scaling where 4 7-tap filters are used, hfilter_coeffs[i×8 + 0, …, i×8 + 6] (i=0,1,2,3) correspond to filter taps for phase 0.25×i. For example, hfilter_coeffs[0, 1, …, 6] are the 7 taps for phase 0 filter. hfilter_coeffts[7, 15, 23, 31] are not used.
  - For 1/4x to 1/2x scaling where 8 4-tap filters are used, hfilter_coeffs[i×4 + 0, …, i×4 + 3] (i=0, …, 7) correspond to filter taps for phase 0.125×i. For example, hfilter_coeffs[0, 1, 2, 3] are the 4 taps for phase 0 filter.
- vfilter_coeffs[0:31] which define the vertical filter coefficients. They are organized in the same way as hfilter_coeffs.

The filter coefficients are in Q8 format and in the range [-256, 256]. A filter coefficient X set for the resizer corresponds to the real number X/256, i.e., if a coefficient is set to 256 for the resizer, it is real value is 1.0.

For details on resizer module in TMS320DM6446, please refer to *TMS320DM644x DMSoC Video Processing Front End (VPFE) User's Guide* (SPRUE38) and *TMS320DM6446 Digital Media System-on-Chip* (SPRS283).

2 De-Interlacing and YUV 4:2:2 to 4:2:0 Conversion Algorithm

2.1 *Interlacing History*

The use of interlace capture and interlace display of video signals originated from the need to conserve bandwidth in the early 20th century. The idea was to capture, transmit, and display only half a frame every 60th of a second, rather than a full frame. This is achieved by capturing (and displaying) even and odd scanlines in an alternating pattern at approximately 60 fields per second. Therefore, de-interlacing involves recreating missing fields that were never captured in the first place and as a result is inherently not an exact science.

2.2 Front-End vs. Back-End De-Interlacing

The need for de-interlacing video arises for two main reasons. First, de-interlacing is required when interlaced video content is displayed on a progressive video display. De-interlacing also is required when interlaced video content is compressed using a progressive video encoder. The quality needs for the former are typically much higher than that of the later. This is based on whether de-interlacing is performed before or after video compression. Therefore, there are many different ways to perform de-interlacing, each with their own level of algorithmic complexity and computational requirements. The choice of which method to use is often determined by the needs of the application, required video quality, and available resources.

2.3 De-Interlacing Methods

A progressive frame is one that consists of all lines within the frame being captured at the same time instance. For example, a 480p30 sequence is made of 30 progressive frames captured each second. Each individual frame is made up of 480 lines of video that were all captured at the same time instance. A simple way to think of this is that each progressive frame is just a 480-lined picture captured every 1/30th of a second. In contrast, in an interlaced scheme, video frames are composed as a combination of two alternating fields that are from different time instances. For example, a 480i60 sequence consists of 60 interlaced fields captured each second. Each video frame is composed of a top field of 240 lines and a bottom field of 240 lines. The top field consists of only even rows of the frame while the bottom field consists of only odd rows of the frame. Each top and bottom pair of fields that create an interlaced frame is captured at a 1/60th of a second interval apart from each other.

In an interlaced frame, there is often a strong temporal correlation between a missing field and its neighboring co-located fields. In other words, although a bottom field at t = 33.3 ms might not have been captured, it potentially has a strong correlation with the bottom fields that were captured at t = 16.7 ms and t = 50.0 ms. In addition to co-located fields, a missing bottom field at t = 33.3 ms also has a strong spatial correlation with the captured top field at t=33.3 ms.

De-interlacing is simply the process of utilizing these two correlations to recreate fields that were never captured. The different de-interlacing algorithms that convert interlaced frames into progressive frames use different combinations of these two basic correlations. Some methods rely solely on spatial correlations while others rely on temporal correlations. More sophisticated de-interlacing algorithms use a combination of the two correlations within a single frame. The most intelligent and computationally expensive algorithms use such a combination after extensive analysis of the current and neighboring frames.

Typically for areas with minimal motion, temporal correlation is quite high. In fact, in areas with no motion at all, the correlation is 100%. However, for areas of a frame with moderate to severe motion, high temporal correlation is achieved only when motion is compensated for. Otherwise, co-located fields often become of little use in generating a missing field. The motion compensation needed to make use of temporal correlations within scenes containing motion requires algorithms that can range from 15-40% of typical DSP loadings of video compression algorithms.

In contrast, high spatial correlation between a missing field and an available opposite field has less to do with motion and more to do with the level of spatial detail within a frame. Candidate frames with high spatial detail (i.e., high-frequency content) are more susceptible to degradation when utilizing spatial correlations between fields. Conversely, frames with less spatial detail are less vulnerable to such loss of quality.

Therefore, a simple spatial correlation based method discards an entire set of fields and uses only information from the remaining field to generate the missing data. For example, in the case of 480i60, discarding all bottom field data would yield a 240p30 video. The remaining 240p30 data is then resized vertically by a factor of 2X to generate a 480p30 de-interlaced result. Although this method removes 100% of all interlacing artifacts, the disadvantage from this approach is the loss of vertical fidelity.

Such a loss of vertical fidelity can be drastic in some cases; however, the consequence is not always dire. The viability of such a route depends on several factors. If such a de-interlacing method is used to convert high-quality interlaced video into progressive video for immediate progressive display, the approach may prove to be less than satisfactory. Instead, such an approach may be adequate when used as a pre-processing step before progressive compression, This is based on the fact that lossy video compression algorithms, especially at low bitrates, typically discard high frequencies. Depending on the source content, video quality, and compression needs of the application, such a method for achieving de-interlacing can at times prove to be at par with some high-quality, high-complexity de-interlacing algorithms after compression is taken into account.

In addition to generating compressed video at reasonable quality for certain applications, such an approach is also low on the range of computational complexity for the various de-interlacing algorithms. In fact, many DSPs (including DM6446) incorporate hardware resizers that can facilitate such an approach at no expense to DSP loading. Achieving reasonable quality de-interlacing without consuming DSP cycles makes such an approach highly attractive for certain applications. This is especially true when the DSP cycles saved can be alternately used to further enhance compression quality. Using DSP cycles for software based de-interlacing instead of using an available hardware resizer typically has a marginal impact on the quality of compressed video. Allocating those same cycles to further enhance video compression algorithms can, however, often have a significant impact on the compressed video quality.

2.4 Other Alternate Approaches

An alternate approach, would be to discard a field, encode 240p30, and rescale vertically by 2X back to 480p30 on the decode side. Because only half the data is being encoded, such an approach can be used either as a means to conserve bandwidth or to freely allocate up to twice as many bits to a given frame (in an attempt to achieve much higher quality). Another alternative is to use an interlaced-based video encoder (such as MPEG-4 advanced simple profile), encode 480i60, and perform the de-interlacing on the decode side as well. Both approaches, however, perform the de-interlacing step on decompressed video content rather than on original source content. De-interlacing always yields better results when performed on original source content rather than on decompressed content. This is because the addition of compression artifacts in these methods reduces spatial and temporal correlation levels. Therefore, great care must be taken to ensure satisfactory video quality when using either of these alternate approaches.

Yet another approach involves de-interlacing an image when the source video requires a downscaling operation. Converting a 480i60 video sequence into a 352×240p30 would be one example of such a case. With this scenario, the optimal approach would be to discard an entire field and only downscale the remaining field horizontally. Doing so provides a progressive video signal without any loss of additional output vertical fidelity. The same approach can be used for other downscaling ratios.

2.5 4:2:2 to 4:2:0 Conversion Algorithm

Offloading a 4:2:2 to 4:2:0 conversion of the DSP enables DSP cycles to be utilized for computationally expensive operations, such as video encoding or video analytics. The conversion entails vertically down-sampling all chroma buffers by a factor of 2 and unpacking the video data from an interleaved to a planar format. The vertical down-sampling can be achieved by using the DM6446 resizer in its intended use. The unpacking can be achieved by using the resizer to perform horizontal down-sampling coupled with specific filter coefficients that effectively mask out unwanted data. For example, because 4:2:2 data is stored in a *U1Y1V1Y2* format, scaling this data down by a factor of 2 with appropriate coefficients yields *Y1Y2*. When applied to an entire row of 4:2:2 data, an entire row of 4:2:0 Y data is generated. A similar approach can be used to individually unpack the U and V buffers as well.

3 Implementation Details

3.1 Implementing a De-Interlacer Alone

The de-interlacing method presented in this application report is to remove every odd row in the original image and re-interpolate them using the neighboring even row pixels. The drawback of this approach is that it removes half of the information contained in the odd rows while performing de-interlacing. It is a cheap but not perfect de-interlacer implemented on hardware to offload DSP processing.

The input frame is assumed to be in a 4:2:2 interleaved format organized as Figure 2 (UYVY). The frame resolution is NTSC SD or 720×480 pixels per frame. The output is in the same format and resolution.

The basic idea to make the resizer discard every odd row and re-interpolate them is illustrated in Figure 6. We tell the resizer that the width of the input frame is 720 pixels but the pitch is twice as wide. That is, in_hsize is set to 724 pixels, which is not exactly 720, as explained in Section 1.4. The parameter in_pitch is set to (720+360×2)×2 = 2880 bytes. In this way, the resizer only performs horizontal scaling on even rows in the left and discards the odd rows in the right. Further, we tell the resizer that the width of the output frame is 720 pixels and output pitch is 1440 (720+360×2) bytes. Thus, the resizer is configured to perform horizontal scaling at ratio 1:1, and adjacent output rows will be contiguous in memory.

The input and output vertical size is set to 244 and 480 pixels respectively so that resizer performs 1:2 up-scaling vertically. Again, in_vsize = 244 but not 240, as explained in Section 1.4.

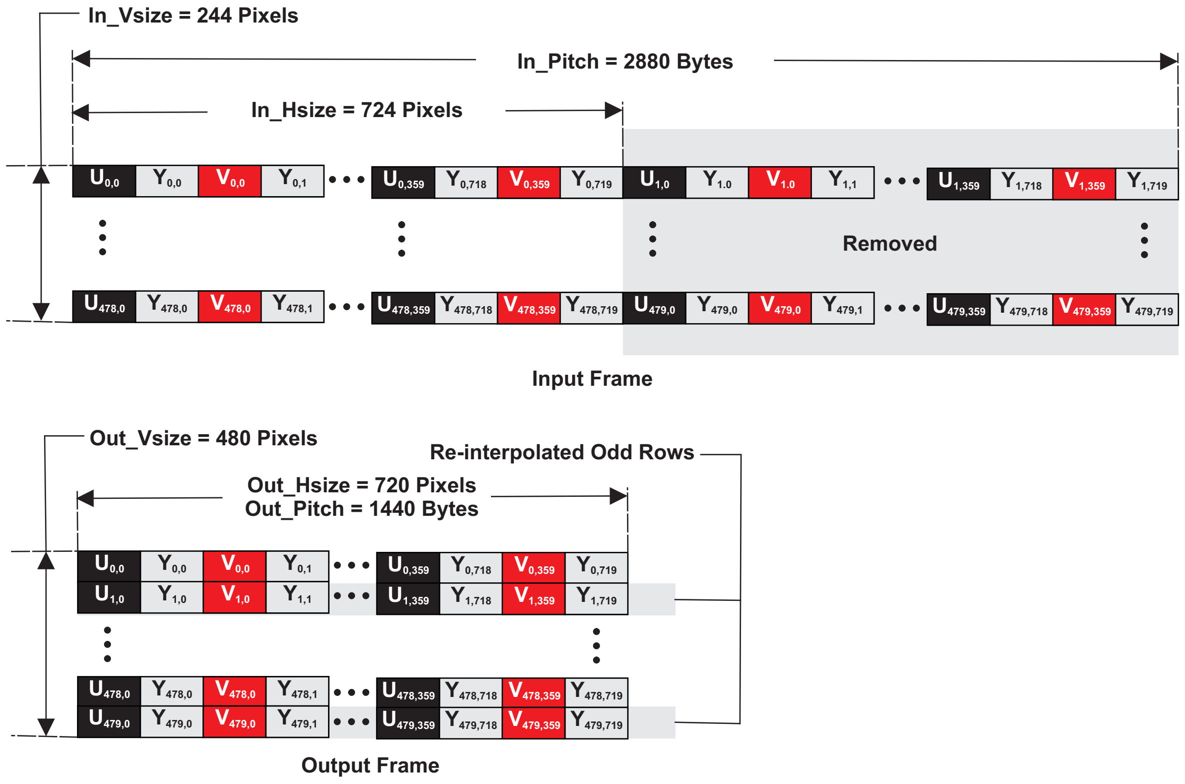
In_Vsize = 244 Pixels
In_Pitch = 2880 Bytes
In_Hsize = 724 Pixels
U0,0 Y0,0 V0,0 Y0,1 • • • U0,359 Y0,718 V0,359 Y0,719 U1,0 Y1,0 V1,0 Y1,1 • • • U1,359 Y1,718 V1,359 Y1,719
Removed
U478,0 Y478,0 V478,0 Y478,1 • • • U478,359 Y478,718 V478,359 Y478,719 U479,0 Y479,0 V479,0 Y479,1 • • • U479,359 Y479,718 V479,359 Y479,719
Input Frame
Out_Vsize = 480 Pixels
Re-interpolated Odd Rows
Out_Hsize = 720 Pixels
Out_Pitch = 1440 Bytes
U0,0 Y0,0 V0,0 Y0,1 • • • U0,359 Y0,718 V0,359 Y0,719
U1,0 Y1,0 V1,0 Y1,1 • • • U1,359 Y1,718 V1,359 Y1,719
U478,0 Y478,0 V478,0 Y478,1 • • • U478,359 Y478,718 V478,359 Y478,719
U479,0 Y479,0 V479,0 Y479,1 • • • U479,359 Y479,718 V479,359 Y479,719
Output Frame

Figure 6. De-Interlacing

Because the horizontal scaling is at ratio 1:1, only the 4-tap filter for phase 0 is used, that is, only hfiter_coeffs[0, 1, 2, 3] are used. We do not intend to perform any filtering horizontally but want to keep each value unchanged in the original frame; the values of hfilter_coeffs[0, 1, 2, 3] are set as follows.

hfilter_coeffs[0] = 256;

hfilter_coeffs[1] = hfitler_coeffs[2] = hfiter_coeffs[3] = 0;

The vertical scaling is performed at ratio 1:2, which means the phase 0 and phase 0.5 4-tap filters are used. The phase 0 filter is applied to neighboring even rows in the input frame to generate even rows in the output frame. Again, because we want to keep even row pixel values in the input frame unchanged, vfilter_coeffs[0, 1, 2, 3] should be set as follows:

vfilter_coeffs[0] = 256;

vfilter_coeffs[1] = vfitler_coeffs[2] = vfiter_coeffs[3] = 0;

The phase 0.5 filter is applied to neighboring even rows in the input frame to generate (re-interpolate) odd rows in the output frame. The coefficients correspond to phase 0.5 filter are vfilter_coeffs[16, 17, 18, 19]. When interpolating an odd row, we give more weight to the two even rows next to the odd row to be interpolated, and less weight to the other two even rows. Specifically, vfilter_coeffs[16, 17, 18, 19] are set as follows, where 160 and -32 correspond to real value 0.625 and -0.125, respectively.

vfilter_coeffs[17] = vfilter_coeffs[18] = 160;

vfilter_coeffs[16] = vfitler_coeffs[19] = -32;

The following pseudo code shows how resizer is used to accomplish this task. The comments and explanations to the pseudo code are embedded as the bold text.

```
inImageSize = 720*480*2;                 // input image size is in 422 interleaved format
outImageSize = inImageSize;              // output image is in the same size

fin = fopen ( inputFileName, "rb" );     // open file containing input frames
fout = fopen ( outputFileName, "wb" );   // open file to write output frames to
// allocate a physically contiguous memory as input buffer and get its physical address
inBuf = Memory_contigAlloc( imageSize, Memory_DEFAULTALIGNMENT);
inBufPhyAddr = Memory_getPhysicalAddress(inBuf);
resize.in_buf.index = -1; // -1 means the buffer is user supplied
resize.out_buf.index = 0;
resize.in_buf.offset = inBufPhyAddr;

// Initialize resizer device and get its file descriptor
resizerFd = initResizerDevice();

// ask resizer device to allocate memory for output buffer
req_outbufs.size = outImageSize;
req_outbufs.buf_type = RSZ_BUF_OUT;
req_outbufs.count =1;
ioctl( resizerFd, RSZ_REQBUF, &req_outbufs );

// map the output buffers to user space. The user space buffer address is outBuffer
bufd.buf_type = RSZ_BUF_OUT;
bufd.index = 0;
ioctl( resizerFd, RSZ_QUERYBUF, &bufd);
outBuffer = (char*)mmap( 0, outImageSize, PROT_READ | PROT_WRITE,
                         MAP_SHARED, resizerFd, bufd.offset);
resize.out_buf.size = outImageSize;

// set size parameters for resizer, as explained before
params.in_vsize = 244;
params.in_hsize = 724;
params.out_hsize = 720;
params.out_vsize = 480;
params.out_pitch = 1440;
params.in_pitch = 2880;

// tell resizer the input frame is in 422 interleaved format
params.inptyp = RSZ_INTYPE_YCBCR422_16BIT;
params.pix_fmt = RSZ_PIX_FMT_UYVY;

params.yenh_params.type = RSZ_YENH_DISABLE; // disable the luminance enhancement module
// set starting pixel and phase information
params.vert_starting_pixel =0;
params.horz_starting_pixel =0;
params.hstph = 0;
params.vstph = 0;

// clear horizontal filter coefficients for safety
for ( i = 0; i < 32; I++ ) {
  params.hfilt_coeffs[i] = 0;
  params.vfilt_coeffs[i] = 0;
}

// set coefficient for phase 0 horizontal filter
params.hfilter_coeffs[0] = 256;

// set coefficient for phase 0 vertical filter
params.vfilt_coeffs[0] = 256;
```

```
// set coefficients for phase 0.5 vertical filter
params.vfilt_coeffs[16] = -32;
params.vfilt_coeffs[17] = 160;
params.vfilt_coeffs[18] = 160;
params.vfilt_coeffs[19] = -32;

// configures the resizer using the parameters
ioctl( resizerFd, RSZ_S_PARAM, &params );

// set the resizer speed to fastest
rszSpeed = 0;
rszError = ioctl( resizerFd, RSZ_S_EXP, &rszSpeed );

while ( !feof (fin ) ) {

  // read a whole input frame. Otherwise, quit.
  if ( inImageSize != fread( inBuf, 1, inImageSize, fin ) )
    break;

  // ask resizer to do de-interlacing
  ioctl( resizerFd, RSZ_RESIZE, &resize );

  // write the de-interlaced frame to output file
  fwrite( resizedBuffer, 1, outImageSize, fout );
}
```

3.2 Implementing De-Interlacer With YUV 4:2:2 to 4:2:0 Conversion

This example explains how to combine the de-interlacing operation with the 4:2:2 to 4:2:0 conversion operation. For each input frame in 4:2:2 interleaved format, the resizer needs to be called 3 times to generate the de-interlaced output frame in 4:2:0 planar format, which means 3 sets of configuration parameters need to be maintained. This is because the 4:2:2 to 4:2:0 conversion needs to extract YUV values being interleaved together into 3 separate planes while the resizer hardware is not able to do that in 1 pass. Again, we assume the input frame is in NTSC SD resolution and UYVU 4:2:2 format. The output frame is in 4CIF resolution (704×480) instead of NTSC SD resolution (720×480). In the example provided, the right 16 columns in the input frame are cut due to the 32-byte output alignment limitation of the resizer. Alternately, 8 columns from the right and 8 columns from the left could have been cropped. The detailed explanation is shown below.

3.2.1 De-Interlacing and Extracting Y Components

The first call is to extract all the Y components in the input frame and perform de-interlacing on them. The idea is illustrated in Figure 7. Y components in every odd row need to be removed and re-interpolated using Y components in neighboring even rows. Again, in_pitch is set to 2880 bytes, in_vsize is set to 244, and out_vsize is set to 480 as before. Vertically, the scaling ratio is still 1:2.

Ideally we want to the width of the output Y plane to be 720 bytes, which unfortunately is not 32-byte aligned. The compromise is to make the output width 704 (out_hsize=704), which requires the right 16 columns to be removed.

An additional concern is to make sure that the interleaved U/V data does not interfere with generating the de-interlaced Y output. The solution is to tell the resizer to treat the input frame as a image in 4:2:0 plannar format (inptye = RSZ_INTYPE_PLANAR_8BIT, pix_fmt = RSZ_PIX_FMT_PLANAR). Then we tell the resizer to perform 2:1 scaling horizontally to extract every other Y component in the input frame. Thus, in_hsize is set to 1414 bytes, which is not exactly 704×2=1408, as explained in Section 1.4.

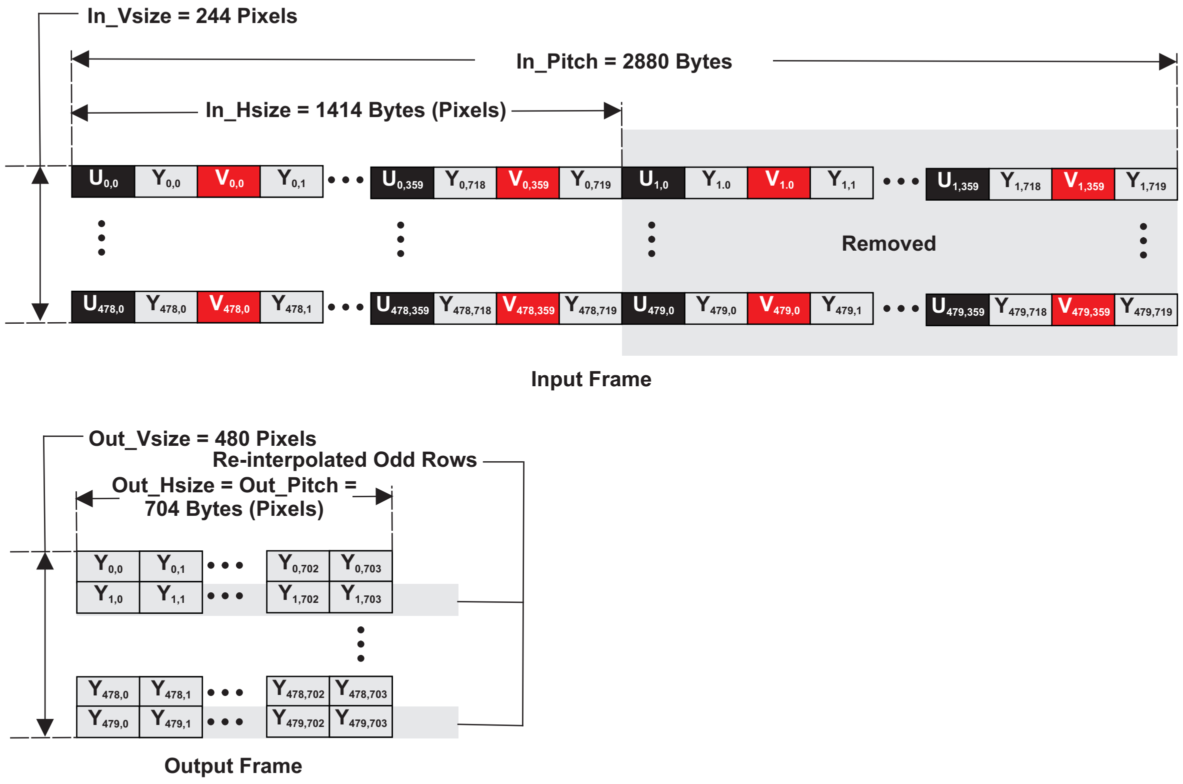

Figure 7. De-Interlacing + 4:2:2 to 4:2:0 Conversion: 1st Call for Y

Phase 0 and 0.5 4-tap filters are used for vertical scaling ratio at 1:2. The phase 0 filter is applied to Y components in neighboring even rows in the input frame to generate Y components of even rows in the output frame. vfilter_coeffs[0,1,2,3] should be set as following to keep Y values in even rows unchanged.

vfilter_coeffs[0] = 256;

vfilter_coeffs[1] = vfitler_coeffs[2] = vfiter_coeffs[3] = 0;

The phase 0.5 filter is applied to Y components in neighboring even rows in the input frame to generate (re-interpolate) Y components of odd rows in the output frame. The coefficients correspond to phase 0.5 filter are vfilter_coeffs[16, 17, 18, 19]. Again, we give more weight to the two even rows next to the odd row to be interpolated, and less weight to the other two even rows. Specifically, vfilter_coeffs[16, 17, 18, 19] are set as follows, where 160 and -32 correspond to real value 0.625 and -0.125. respectively.

vfilter_coeffs[17] = vfilter_coeffs[18] = 160;

vfilter_coeffs[16] = vfitler_coeffs[19] = -32;

Only the phase 0 4-tap filter is used for horizontal downscaling at ratio 2:1. hfilter_coeffs[0, 1, 2, 3] are set as follows to extract every Y component after U/V component.

hfilter_coeffs[1] = 256;

hfilter_coeffs[0] = vfitler_coeffs[2] = vfiter_coeffs[3] = 0;

Note: If the input frame is in YUYV (instead of UYVY) 4:2:2 interleaved format, then hfitler_coeffs[0]=256 and hfilter_coeffs[1] = hfilter_coeffs[2] = hfilter_coeffs[3] = 0.

3.2.2 Extract and Down-Sample U Components Vertically

The U components in a 4:2:2 interleaved format have been horizontally down-sampled by a ratio of 2:1 with respect to the Y component. To generate the output frame in a 4:2:0 planar format, they need to be extracted and further down-sampled vertically by a ratio of 2:1. As a result, the de-interlacing operation does not need to be applied to U components. This is because down sampling vertically by a ratio of 2:1 involves discarding all the odd rows, which automatically generates progressive U components.

The idea is illustrated in Figure 8. Because the output width for Y components has to be 704 due to the 32-byte output alignment limitation of the resizer, the output width for the U and V components should be 352. To extract every U component in the input frame, we tell the resizer to treat the input frame as a image in 4:2:0 plannar format (inptye = RSZ_INTYPE_PLANAR_8BIT, pix_fmt = RSZ_PIX_FMT_PLANAR). Then we configure it to perform horizontal down-scaling at 1:4 ratio to extract every U component. Therefore, in_pitch is set to 720×2 = 1440 bytes, in_hsize is set to 1412 bytes, and out_hsize is set to 352 bytes. Again, in_hsize is not exactly 704×2=1408, as explained in Section 1.4.

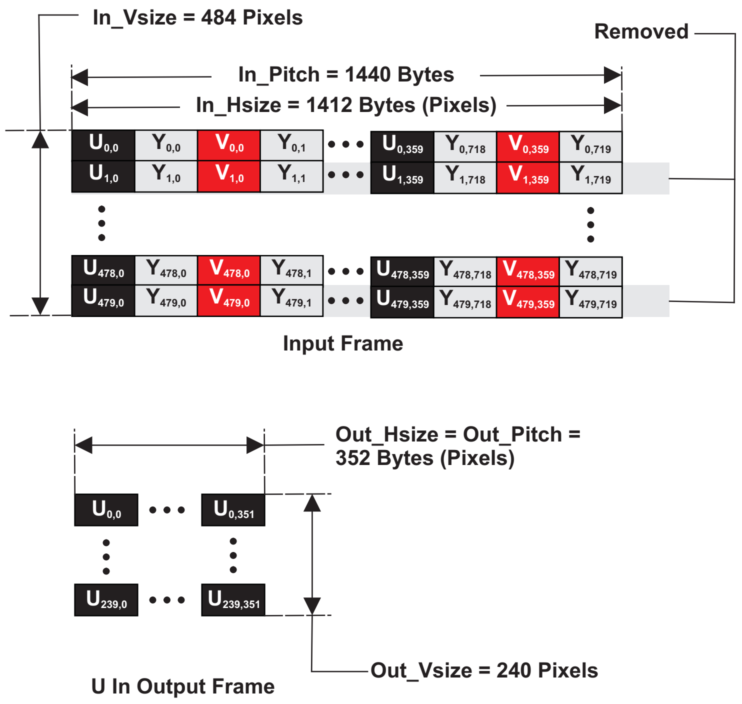
In_Vsize = 484 Pixels
Removed
In_Pitch = 1440 Bytes
In_Hsize = 1412 Bytes (Pixels)
$U_{0,0}$ $Y_{0,0}$ $V_{0,0}$ $Y_{0,1}$ $U_{0,359}$ $Y_{0,718}$ $V_{0,359}$ $Y_{0,719}$
$U_{1,0}$ $Y_{1,0}$ $V_{1,0}$ $Y_{1,1}$ $U_{1,359}$ $Y_{1,718}$ $V_{1,359}$ $Y_{1,719}$
$U_{478,0}$ $Y_{478,0}$ $V_{478,0}$ $Y_{478,1}$ $U_{478,359}$ $Y_{478,718}$ $V_{478,359}$ $Y_{478,719}$
$U_{479,0}$ $Y_{479,0}$ $V_{479,0}$ $Y_{479,1}$ $U_{479,359}$ $Y_{479,718}$ $V_{479,359}$ $Y_{479,719}$
Input Frame
Out_Hsize = Out_Pitch = 352 Bytes (Pixels)
$U_{0,0}$ $U_{0,351}$
$U_{239,0}$ $U_{239,351}$
U In Output Frame
Out_Vsize = 240 Pixels

Figure 8. De-Interlacing + 4:2:2 to 4:2:0 Conversion: 2nd Call for U

To perform horizontal downscaling at ratio 1:4, the phase 0 7-tap horizontal filter is used. They are set as follows to keep the U values unchanged.

hfilter_coeffs[0] = 256;

hfilter_coeffs[1] = vfitler_coeffs[2] = vfiter_coeffs[3] =

hfilter_coeffs[4] = vfitler_coeffs[5] = vfiter_coeffs[6] = 0;

Note: If the input frame is in YUYV (instead of UYVY) 4:2:2 interleaved format, then hfitler_coeffs[1]=256 and other coefficients are zeros.

To perform vertical downscaling at ratio 2:1, in_vsize is set to 484 and out_vsize is set to 240. in_vsize is not exactly 480, as explained in Section 1.4.

The phase 0 4-tap filter is used to perform vertical 2:1 down scaling. To keep the U values in even row unchanged and remove U values in every odd row, vfilter_coeffs[0,1,2,3] are set as follows.

vfilter_coeffs[0] = 256;

vfilter_coeffs[1] = vfitler_coeffs[2] = vfiter_coeffs[3] = 0;

3.2.3 Extract and Down-Sample V Components Vertically

The operation on V components is similar to that on U components. The only difference is that hfilter_coeffs[0, 1, 2, 3, 4, 5, 6] need to be set as follows to extract the V components in the input frame.

hfilter_coeffs[2] = 256;

hfilter_coeffs[0] = vfitler_coeffs[1] = vfiter_coeffs[3] =

hfilter_coeffs[4] = vfitler_coeffs[5] = vfiter_coeffs[6] = 0;

Note: If the input frame is in YUYV (instead of UYVY) 4:2:2 interleaved format, then hfitler_coeffs[3]=256 and other coefficients are zeros.

3.2.4 Pseudo Code

The following is the pseudo code showing how to use resizer to perform de-interlacing and 4:2:2 to 4:2:0 conversion. The explanations and comments are embedded as bold text.

```
inImageSize = 720*480*2; // input frame is NTSC SD resolution, 422 format
outImageSize = 704*480; // output size for Y plane

fin = fopen (inputFileName, "rb" ); // open file containing input video frames
fout = fopen ( envp->videoOutFile, "wb" ); // open file to write output frames to
// allocate a physically contiguous memory as input buffer and get its physical address
inBuf = Memory_contigAlloc( imageSize, Memory_DEFAULTALIGNMENT);
inBufPhyAddr = Memory_getPhysicalAddress(inBuf);
for ( i = 0; i < 3; i++ ){
  resize[i].in_buf.index = -1; // -1 means the buffer is user supplied
  resize[i].in_buf.offset = inBufPhyAddr;
}

// Initialize resizer device, get 3 file descriptors corresponding to Y, U, V operation
for ( i = 0; i < 3; i++ ) {
  resizerFd[i] = initResizerDevice();
}

// ask resizer to allocate contiguous memory as output buffer for Y, U, V plane
req_outbufs[0].size = outImageSize;
req_outbufs[1].size = outImageSize>>2; // size of U and V plane is 1/4x of Y plane
req_outbufs[2].size = outImageSize>>2;
for ( i = 0; i < 3; i++ ) {
  req_outbufs[i].buf_type = RSZ_BUF_OUT;
  req_outbufs[i].count = 1;
  ioctl( resizerFd[i], RSZ_REQBUF, &req_outbufs[i] );
  bufd[i].buf_type = RSZ_BUF_OUT;
  bufd[i].index = 0;
  ioctl( resizerFd[i], RSZ_QUERYBUF, &bufd[i] );
}

// map output buffers to user space, buffer address for component i is outBuffer[i]
outBuffer[0] = (char*)mmap( 0, outImageSize, PROT_READ | PROT_WRITE,
              MAP_SHARED,resizerFd[0], bufd[0].offset );
outBuffer[1] = (char*)mmap( 0, outImageSize>>2, PROT_READ | PROT_WRITE,
              MAP_SHARED,resizerFd[1], bufd[0].offset );
outBuffer[2] = (char*)mmap( 0, outImageSize>>2, PROT_READ | PROT_WRITE,
              MAP_SHARED,resizerFd[2], bufd[0].offset );
resize[0].out_buf.size = outImageSize;
resize[1].out_buf.size = resize[2].out_buf.size = outImageSize>>2;
for ( i = 0; i < 3; i++ ){
  resize[i].out_buf.index = 0; // index != -1 means it is allocated by resizer
}
```

```
// clear filter coefficients for safety
for ( i = 0; i < 3; i++ ) {
  for ( j = 0; j < 32; j++ ) {
  params[i].hfilter_coeffs[j] = 0;
  params[i].vfilter_coeffs[j] = 0;
  }
}

// set params which are common for Y, U, V
for ( i = 0; i < 3; i++ ) {
  // tell resizer to treat input frame as 420 planar format so that operation is 8bit based
  params[i].inptyp = RSZ_INTYPE_PLANAR_8BIT;
  params[i].pix_fmt = RSZ_PIX_FMT_PLANAR;

  params[i].yenh_params.type = RSZ_YENH_DISABLE; // disable Y enhancement module

  params[i].cbilin = 0;
  params[i].vert_starting_pixel =0;
  params[i].horz_starting_pixel =0;
  params[i].hstph = 0;
  params[i].vstph = 0;
  params[i].yenh_params.gain = 0x7f;
  params[i].yenh_params.slop = 0x7f;
  params[i].yenh_params.core = 0;
}

// set params specific for Y component
params[0].in_vsize = 244;
params[0].in_hsize = 1414;
params[0].out_hsize = 704;
params[0].out_vsize = 480;
params[0].out_pitch = 704;
params[0].in_pitch = 2880;
params[0].hfilt_coeffs[1] = 256;  // extract Y from input frame
params[0].vfilt_coeffs[1] = 256;  // keep Y values in even rows unchanged
params[0].vfilt_coeffs[16] = -32; // re-interpolate Y values in odd rows using this filter
params[0].vfilt_coeffs[17] = 160;
params[0].vfilt_coeffs[18] = 160;
params[0].vfilt_coeffs[19] = -32;

// set params specific for U, V
for ( i = 1; i < 3; i++ ) {
  params[i].in_pitch = 1440;
  params[i].out_hsize = 352;
  params[i].out_vsize = 240;
  params[i].out_pitch = 352;
  params[i].in_vsize = 484;
  params[i].in_hsize = 1412;
}
params[1].hfilt_coeffs[0] = 256; // extract U from input frame
params[2].hfilt_coeffs[2] = 256; // extract V from input frame
params[1].vfilt_coeffs[0] = 256; // keep U values in even rows unchanged
params[2].vfilt_coeffs[0] = 256; // keep V values in even rows unchanged
// configure the 3 resizer instances using the params
for ( i = 0; i < 3; i++ ) {
  ioctl ( resizerFd[i], RSZ_S_PARAM, &params[i] );
}

// set resizer speed to fastest. Since it is a device based parameter instead of a file
// descriptor based parameter, calling it once is enough
rszSpeed = 0;
ioctl( resizerFd[0], RSZ_S_EXP, &rszSpeed );

while ( !feof (fin ) ) {

  // keep reading a whole input frame. Quit if not.
  if ( inImageSize != fread( inBuf, 1, inImageSize, fin );
    break;

  // calling resizer 3 times to do de-interlacing and 422 to 420 conversion
  for ( i = 0; i < 3; i++ ) {
    ioctl( resizerFd[i],RSZ_RESIZE,&resize[i]);
  }

  // write Y, U, V plane output to file
  fwrite( resizedBuffer[0], 1, outImageSize, fout );
  fwrite( resizedBuffer[1], 1, outImageSize>>2, fout );
  fwrite( resizedBuffer[2], 1, outImageSize>>2, fout );
}
```

Note: The Memory_getPhysicalAddress() is an obsolete CE 1.xx API. For CE 2.00 and later users, call Memory_getBufferPhysicalAddress() instead of Memory_getPhysicalAddress().

4 Example User's Guide

4.1 Package Contents

sprc374.gz is the package containing the provided examples. Type the following command to unzipping it on the Linux® development host.

```
tar –xzf sprc374.gz
```

Table 1 shows the package content.

Table 1. Package Content

Name	Description
test_uyvy_720×480_2f.yuv	This is a YUV raw video file in UYVY 4:2:2 format at NTSC SD resolution. It has two frames.
zhe_szdeinterlacer	This folder contains all the source code to implement the simpler de-interlacer.
zhe_szresize	This folder includes all the source code implementing simple de-interlacing and 4:2:2 to 4:2:0 conversion.

Note: The resizer driver is not included in this package because the production version of the driver was not released at the time this document was published. To test the provided example program, please download the production version of the resizer driver when it is officially released from the TI website, www.ti.com

4.2 How to Run

The following steps show how to run the provided example executables. The prerequisite for users is that the DM6446 DVEVM demos are already installed in the DM6446 EVM board and can be executed successfully.

1. Transfer *davinci_rsz_driver.ko* to directory */opt/dvevm* on the DM6446 EVM board. */opt/dvevm* is the directory where the DM6446 DVEVM demo resides.
2. Transfer *test_uyvy_720×480_2f.yuv* provided by the example package to directory */opt/dvevm* on the DM6446 EVM board.
3. Transfer binary *zhe_szdeinterlacer* in directory *zhe_szdeinterlacer/release* of the provided package to directory */opt/dvevm* on the DM6446 EVM board.
4. Transfer binary *zhe_szresize* in directory *zhe_szresize/release* of the provided package to directory */opt/dvevm* on the DM6446 EVM board.
5. Switch to directory */opt/dvevm* on the DM6446 EVM board.
6. Type *loadmodules.sh* if it has not been done at the Linux booting time.
7. Type *insmod davinic_rsz_driver.ko* to load the resizer driver into the Linux kernel.
8. Type *zhe_szdeinterlacer -i test_uyvy_720×480_2f.yuv -o output_file_name* to execute the simple de-interlacer. The input file is specified as *test_uyvy_720×480_2f.yuv* with “-i” option. Users are allowed to feed their own input files if they are in yuvy 4:2:2 format at NTSC SD resolution. The output file name is specified with “-o”. The actual name can be any. The output frames are de-interlaced with the same format and resolution as the input file.

9. Type *zhe_szresize -i test_uyvy_720×480_2f.yuv -o output_file_name* to execute the program doing both simple de-interlacing and 4:2:2 to 4:2:0 conversion. The meaning of “-i” and “-o” are the same as the ones in *zhe_szdeinterlacer*. The output frames are de-interlaced in 4:2:0 format at 704×480 resolution.

4.3 How to Compile

Compiling the provided examples require that the DM6446 DVEVM package has been installed on the Linux development host. The following steps show how to compile the zhe_deinterlacer program.

1. Transfer directory *zhe_deinterlacer* provided with the example package to directory*"/home/user/dvevm_x_xx/demos/*" on the Linux development host. *"/home/user/dvevm_x_xx*" is the directory where the DM6446 DVEVM package is installed.
2. Switch to directory *"/home/user/dvevm_x_xx/demos/zhe_deinterlacer*"on the Linux development host and type *make*. The executable binary will be in the /release sub-directory.

The following steps show how to compile the zhe_resizer program. They are similar as compiling zhe_deinterlacer.

1. Transfer directory *zhe_resizer* provided with the example package to directory *"/home/user/dvevm_x_xx/demos/"* on the Linux development host.
2. Switch to directory *"/home/user/dvevm_x_xx/demos/zhe_resizer*" on the Linux development host and type *make*. The executable binary is in the /release sub-directory.

5 References

- *TMS320DM644x DMSoC Video Processing Front End (VPFE) User's Guide* (SPRUE38)
- *TMS320DM6446 Digital Media System-on-Chip* (SPRS283)