Adapting the SPRA904 Motion Detection Application Report to the DM642 EVM

Adit Sahasrabudhe
Technical Staff DSP Applications – Waltham, MA

ABSTRACT
This application report describes the modifications that needed to be made to the multichannel motion detection system described in application report, A Multichannel Motion Detection System Using eXpressDSP RF5 NVDK Adaptation (literature number SPRA904) to port it to the DM642 EVM system. Project collateral discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/SPRA950.

Contents

1 NVDK DM642 EVM: Overall Considerations ......................................................... 1
  1.1 Video Ports ........................................................................................................... 1
  1.2 256KB Internal SRAM ......................................................................................... 1
2 Modifications for Video Ports ................................................................................ 2
  2.1 FVID Device Driver ............................................................................................. 2
  2.2 Changing to YCrCb Output .................................................................................. 4
3 Modifications for Less Internal SRAM ................................................................. 6
  3.1 Using ISRAM Effectively .................................................................................... 6
    3.1.1 Small Cache Issue ......................................................................................... 6
    3.1.2 Moving Sections to SDRAM ........................................................................ 7

1 NVDK DM642 EVM: Overall Considerations

1.1 Video Ports
One of the distinguishing features of the DM642 are the video ports. The video ports allow the DSP to directly interface with video encoders/decoders without using an FPGA, and more importantly, without taking bandwidth away from the EMIF. With this in mind, we need to be able to correctly program the video ports to capture an image of CIF size (352x240) and display a full D1 resolution image (720x480). To do this, we use the device driver provided in the Driver Development Kit (DDK) version 1.1. Also, the video ports make it very easy to output BT.656 format data, so we change the output format to composite YCrCb, instead of VGA RGB.

1.2 256KB Internal SRAM
A very important consideration when changing from the NVDK (C6416 DSP) to the DM642 EVM is the change in the amount of available internal memory. The 6416 has 1MB of internal memory. This allowed us to be very liberal with its usage, putting almost all needed buffers and BIOS objects there. With the DM642, we now have 256KB of internal memory, so we need to be aware of what buffers and objects we place internally. This way we can maximize the utility of internal memory.

Trademarks are the property of their respective owners.
2 Modifications for Video Ports

2.1 FVID Device Driver

Before adding code to interface the video ports with the video encoder/decoders, we must remove remnants of the video driver from the NVDK. Ateme (the maker of the NVDK) provides a set of libraries and API calls for configuring the EMIF to capture video from the FPGA on the NVDK. Therefore, the Ateme Graphics Library (agl_c64.lib and iekc64_d.lib) included in the linker command file, needs to be removed. In addition, any #include that references Ateme Graphics Libraries (agl.h and iekc64.h) must be removed from thrCapture.c, thrDisplay.c, and thrProcess.c. For driver-specific calls, the following needs to be removed:

- thrCapture.c:
  - The functions openInput() and getInputFrame()
  - The functions openDisplay() and putOutputFrame()

Drivers for the video ports have been developed and are available as a part of the DDK. The driver API for the video port is named FVID. The following calls need to be added:

NOTE: The structures EVMDM642_vCapParamsChan_EMBEDDED and EVMDM642_vDisParamsChan are available in source files that need to be added to your project. These files are available in the folder \CCS_INSTALL\boards\evmdm642\examples\video\driver\settings. Choose the source file corresponding to the format you are using. The example provided uses the files evmdm642_vcapparams360x240.c and evmdm642_vdisparamsNTSC.c.

- thrCapture.c:
  - In the function thrCaptureInit()

```c
EVMDM642_vCapParamsChan.segid = EXTERNALHEAP;
EVMDM642_vCapParamsSAA7115.hI2C = EVMDM642_I2C_hI2C;

capChan = FVID_create("/VP0CAPTURE/A/0", IOM_INPUT, &status, (Ptr)&EVMDM642_vCapParamsChan, NULL);

FVID_control(capChan, VPORT_CMD_EDC_BASE+EDC_CONFIG, (Ptr)&EVMDM642_vCapParamsSAA7115);
```

There is no longer a need to perform a scaling of the input image to CIF because the SAA7115 Decoder performs that functionality. The parameters required to do this are shown in the source file mentioned in the NOTE above. A piece of this code is shown next:
... SAA7115_ConfParams EVMDM642_vCapParamsSAA7115 = {
    SAA7115_MODE_NTSC720,
    SAA7115_MODE_USER,
    SAA7115_APMT_COMPOSITE,
    TRUE,
    TRUE,
    INV,
    LINE_SZ,
    NUM_LINES*2,
    TRUE,
};

- In the function \textit{thrCaptureStartup()}

\begin{verbatim}
FVID_control(capChan, VPORT_CMD_START, NULL);
\end{verbatim}

- In the function \textit{thrCaptureRun()}

\begin{verbatim}
... //convert 422 to 420
inBuf[Y] = capFrameBuf->frame.iFrm.y1;
inBuf[CR] = capFrameBuf->frame.iFrm.cr1;
inBuf[CB] = capFrameBuf->frame.iFrm.cb1;

outBuf[Y] = scombufCap->bufYCRCB[Y];
outBuf[CR] = scombufCap->bufYCRCB[CR];
outBuf[CB] = scombufCap->bufYCRCB[CB];

yuv422to420(inBuf, outBuf, PROCF_WIDTH, CAPF_HEIGHT, CAPF_WIDTH);

... FVID_exchange(capChan, &capFrameBuf); //Passes full buffer, receives empty
\end{verbatim}

- \textit{thrDisplay.c:}

- In the function \textit{thrDisplayInit()}

\begin{verbatim}
... EVMDM642_vDisParamsChan.segId = EXTERNALHEAP;
EVMDM642_vDisParamsSAA7105.hI2C = EVMDM642_I2C_hI2C;
disChan = FVID_create("/VP2DISPLAY", IOM_OUTPUT,
&status, (Ptr)&EVMDM642_vDisParamsChan, NULL);
FVID_control(disChan, VPORT_CMD_EDC_BASE+EDC_CONFIG,
(Ptr)&EVMDM642_vDisParamsSAA7105);
... 
\end{verbatim}

- In the function \textit{thrDisplayStartup()}

\begin{verbatim}
FVID_control(disChan, VPORT_CMD_START, NULL);
\end{verbatim}
2.2 Changing to YCrCb Output

In the original application, the output display type was RGB. This was changed to YCrCb since it is the more common type used in applications and because the video ports can easily interface in that format. To transition from RGB to YCrCb, we first removed the YUVtoRGB cell in each channel. When this cell is removed, the pass-through channel now becomes an empty channel with no cells, and any cell registration or channel open calls that are made for channel 1 need to be removed. Then, the functions yuv422to420() and yuv420to422() were added to the Capture and Display threads, respectively. These functions are supplied with source code contained within the Capture and Display thread source files.

To execute the channel, simply pass the buffer from Capture to Display without any modification:

```c
//Execute all passthrough channels
for (chanNum = 0; chanNum < NUMPASSCHANS; chanNum++) {

    //Simply place the buffers from Cap to Dis
    DAT_copy2d(DAT_1D2D, scombufCap->bufYCRCB[Y], scombufDisp->bufYCRCB[Y],
               PROCF_WIDTH, PROCF_HEIGHT, OFF_WIDTH);
    DAT_copy2d(DAT_1D2D, scombufCap->bufYCRCB[CR], scombufDisp->bufYCRCB[CR],
               PROCF_WIDTH >> 1, PROCF_HEIGHT >> 1, OFF_WIDTH >> 1);
    prevCbId = DAT_copy2d(DAT_1D2D, scombufCap->bufYCRCB[Cb], scombufDisp->bufYCRCB[Cb],
                           PROCF_WIDTH >> 1, PROCF_HEIGHT >> 1, OFF_WIDTH >> 1);

    DAT_wait(prevCbId);
}
```

A few other buffer passing schemes need to change. Originally, only the buffers being passed from cell to cell had Y, Cr, and Cb components. The last buffer passed from the process thread to the display thread was a single RGB buffer. Now, even that last buffer requires Y, Cr, and Cb components. In addition, this buffer needs to have offsets for each buffer, depending on where we want to place the channel. Assuming the same channel placement scheme as in SPRA904, the buffer offsets are added to `appThreads.h` and are as follows:
typedef struct placementBuff {
    Char *y;
    Char *cr;
    Char *cb;
} placementBuff;

enum PIXELCOMPONENTS {
    Y = 0,
    CR,
    CB,
    TOTALCOMPONENTS //total types
};

These offsets are passed into each channel before their execution via the same call to ICC_setBuf(). The only difference is that now we use a dummy structure to hold the appropriate addresses. This structure is located in appThreads.h:

typedef struct placementBuff {
    Char *y;
    Char *cr;
    } placementBuff;

// Assign correct offset values to buffers
thrProcess.OutputBuff.y = scombufDisp->bufYCRCB[Y] + Q2_Y_OFFSET;
thrProcess.OutputBuff.cb = scombufDisp->bufYCRCB[CB] + Q2_CB_OFFSET;

enum PIXELCOMPONENTS{
    Y = 0,
    CR,
    CB,
    TOTALCOMPONENTS //total types
};

Then, before the call to ICC_setBuf() we set the correct values. For example, for the DIFF channel:

We need a provision for providing each cell with its correct line pitch. The line pitch is the number of bytes from the start of one line in a transfer to the start of the next line. For cell to cell communication, the line pitch is simply the width of the processing frame, but for the last cell in a channel, the line pitch is the width of the output frame.

Using each cells environment variable, we can pass in the correct value for line pitch. Before each channel is executed, we assign the value in the following manner. Again, the DIFF channel is used as an example:
Adapting the SPRA904 Motion Detection Application Report to the DM642 EVM

Then, within the cell itself, we use a set of intermediate buffers for the output data:

```c
// Point output data to intermediate buffers
yOutData = yOutBuff;
crOutData = crOutBuff;
cbOutData = cbOutBuff;
```

These buffers are filled in a one dimensional fashion, not worrying about line pitch (same as in SPRA904). Once all of the processing is finished, we do a `DAT_copy2d` to send the data to the correct output buffer using the line pitch:

```c
// Send data to output buffer
DAT_copy2d(DAT_1D2D, yOutBuff, outData[0], PROCF_WIDTH, PROCF_HEIGHT, linePitch);
DAT_copy2d(DAT_1D2D, crOutBuff, outData[1], PROCF_WIDTH >> 1, PROCF_HEIGHT >> 1, linePitch >> 1);
finalOutId = DAT_copy2d(DAT_1D2D, cbOutBuff, outData[2], PROCF_WIDTH >> 1, PROCF_HEIGHT >> 1, linePitch >> 1);
DAT_wait(finalOutId);
```

This takes care of appropriate buffer passing for YCrCb display.

In displaying YCrCb data, we have a choice of using s-video or composite output. This selection can be done easily using the display parameters source file mentioned in Section 2.1. At the bottom of the file, we make a modification to the EVMDM642_vDisParamsSAA7105 structure:

```c
SAA7105_ConfParams EVMDM642_vDisParamsSAA7105 = {
    SAA7105_AFMT_SVIDEO, //use for s-video output
    // SAA7105_AFMT_COMPOSITE, //use for composite output
    SAA7105_MODE_NTSC720,
    SAA7105_IFMT_YCBCR422_INTERLACED,
    TRUE,
    FALSE,
    INV
};
```
3 Modifications for Less Internal SRAM

3.1 Using ISRAM Effectively

With the more limited amount of space in Internal SRAM, we want to make sure we maximize its utility. Therefore, we allocate that space for a few important tasks.

First, we make 128KB of ISRAM cache. This allows us to utilize more of a caching architecture to handle data manipulation. In the rest of the 128KB, we want to allocate some space for buffers we designate and other space for an Internal Heap used by some RF5 objects. The buffers we want to be stored internally are the intermediate processing buffers. This ensures that the processor intensive algorithms are run out of fast internal memory. The ONLY buffers we keep internal then are intYBuf, intCrBuf, and intCbBuf which are declared in thrProcess.c. The rest of the ISRAM can be allocated to heap space.

3.1.1 Small Cache Issue

There is a small cache coherency problem when updating the reference frame via GEL. The following code calls to the CACHE API resolve this issue:

```c
// Update Reference Frame if necessary
if (thrProcess.diffEnv->SetReference == TRUE) {
    CACHE_wbInvL2(scombufCap->bufYCRCB[Y], CAPF_SIZE_IN_PIXELS, CACHE_WAIT);
    CACHE_wbInvL2(scombufCap->bufYCRCB[CR], CAPF_SIZE_IN_PIXELS>>2, CACHE_WAIT);
    CACHE_wbInvL2(scombufCap->bufYCRCB[CB], CAPF_SIZE_IN_PIXELS>>2, CACHE_WAIT);

    prevYId = DAT_copy2d(DAT_2D1D, (Void *) scombufCap->bufYCRCB[Y], (Void *) prevY,
                         PROCF_WIDTH, PROCF_HEIGHT, PROCF_WIDTH);
    prevCrId = DAT_copy2d(DAT_2D1D, (Void *) scombufCap->bufYCRCB[CR], (Void *) prevCr,
                          PROCF_WIDTH, PROCF_HEIGHT, PROCF_WIDTH>>1);
    prevCbId = DAT_copy2d(DAT_2D1D, (Void *) scombufCap->bufYCRCB[CB], (Void *) prevCb,
                          PROCF_WIDTH, PROCF_HEIGHT, PROCF_WIDTH>>1);

    CACHE_invL2(prevY, CAPF_SIZE_IN_PIXELS, CACHE_WAIT);
    CACHE_invL2(prevCr, CAPF_SIZE_IN_PIXELS>>2, CACHE_WAIT);
    CACHE_invL2(prevCb, CAPF_SIZE_IN_PIXELS>>2, CACHE_WAIT);

    thrProcess.diffEnv->SetReference = FALSE;
}
```

3.1.2 Moving Sections to SDRAM

Previously, almost all of the BIOS sections in the CDB file are placed into ISRAM. However, we now need to place ALL of those objects into SDRAM. Note, the size of the SDRAM needs to be changed for the DM642 EVM since we have 32MB instead of 256MB. Going through ALL of the properties of the various tabs in the CDB file ensures that the sections have been placed properly.

All compiler sections should also be placed into SDRAM. With the utilization of cache, running code from external memory is acceptable, while leaving space internally for our processing buffers.
IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2019, Texas Instruments Incorporated