Mastering the Art of Memory Map Configuration for DaVinci-Based Systems

Vincent Wan, Niclas Anderberg, Davor Magdic, and Jing Cui [1] Software Development Organization

ABSTRACT

This document describes how to configure Codec Engine-based audio/video applications on the DM6446 (DaVinci) for use in a system that has less than the 256 MB of DDR2 memory that the evaluation board provides. Specifically, we present steps for shrinking memory requirements down to 64 MB, but the principles apply to any amount of DDR2. Project collateral discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/SPRAAQ6.

Contents

[1] In addition to the main authors, we also want to acknowledge others who may contribute to the DaVinci wiki page at http://wiki.davincidsp.com/index.php?title=Changing_the_DVEVM_memory_map.

1 Introduction

Developers who build audio/video applications on TI's DaVinci platforms have a rich software stack available for use with the DVEVM evaluation board. At the core of the DVSDK (Digital Video Software Development Kit), as this stack is called, is Codec Engine (CE), an application programming layer that allows ARM-side applications to execute video and other algorithms on the DSP for faster processing. The DVSDK stack also includes the MontaVista Linux OS, ARM-side Codec-Engine-using demo applications that encode and decode video and sound, and DSP-side executables running the actual video encoding and decoding algorithms. While the supplied software is a strong starting point for creating custom applications, one challenge DaVinci developers face is how to adapt all these components so they fit in a production system.

The DVEVM board has 256 megabytes of DDR2 memory. The DVSDK software stack is by default configured to use all of that memory. Since video algorithms are memory-hungry, this configuration supports most complex video processing scenarios, and it does not require you to deal with the nontrivial issue of changing the memory map. The total DVEVM memory is partitioned by default at 120 MB for Linux, 8 MB for video and other input/output buffers to be exchanged between the ARM and the DSP, and 128 MB for DSP algorithms. Of the last amount, 6 MB is set aside for code and data, and full 122 MB for the video algorithms' intermediate processing buffers organized in a heap. This setting allows several instances of video, image, and other encoders and decoders for various video formats to run at the same time.

After giving some background on the elements we are working with, this document discusses how to determine the minimum memory required and how to partition that memory. The remainder of this document consists of hands-on instructions on how to configure and rebuild the various components to fit the chosen memory map. We demonstrate the principles and execute actual steps on the Codec Engine video_copy example, whose sources every DVSDK user can access. Finally, we showcase the techniques on the example of a real-world, production system application.

1.1 Why Do Codec Engine Applications Consume 256 MB of Memory Out of the Box?

The DaVinci EVM board comes with 256 MB of external memory installed (the maximum amount currently addressable by the part). All the out-of-the-box software (DSP codecs and ARM-side applications) are spread out over all of that space for developer's comfort. This way you don't have to worry about running out of space when allocating buffers or creating memory-hungry instances of video-processing algorithms.

However, since some production platforms based on the DM6446 processor will likely have less than 256 MB of external memory available, developers must be able to shrink the memory used by applications to whatever the target platform provides.

1.2 Physical Sharing of DDR2 Memory between the ARM and the DSP

This 256 MB of physical DDR2 memory is shared between the ARM and the DSP—that is, both processors can access all of the DDR2. The ARM views this memory as virtual addresses through an MMU (Memory Management Unit), while the DSP uses the physical addresses directly. Virtual addresses are used by Linux to provide memory protection between processes, making sure a process only accesses memory to which it has access. If a Linux user process accesses an address to which it does not have access, a segmentation fault (*segfault*) occurs and the process is killed by the OS.

Since the DSP has no MMU, it can not be restricted to certain memory addresses, which means that a "rogue pointer" in DSP-side code can write not just all over the DSP DDR memory, but also over the ARM (Linux) side code and data. These issues can be very difficult to find.

Physical memory addresses for DDR2 are the same for the ARM and the DSP on the DM6446 and range from 0x80000000 to 0x90000000.

1.3 Linux Partition

Linux is different from an RTOS like DSP/BIOS in that it manages all resources in the system for the application. The application requests access to a resource and Linux grants it depending on UNIX permissions and availability. This means that all the memory you give to Linux is "owned" by Linux and is out of your direct control. The DDR Linux memory partition is segmented into pages (4 KB in size on ARM Linux), and this is the minimum unit of memory you can allocate. This means that when you call `malloc()` to reserve some memory for your application, Linux gives this memory to you as a sequence of 4 KB pages. Not only do you not have any control over where in physical memory this memory is allocated, you don't even know if they are physically contiguous (the MMU makes the memory look virtually contiguous to the process).

This is normally a great feature, but it becomes a problem when you want to share a buffer between the ARM and the DSP. This because the DSP needs *physically contiguous* memory to work with. This is the reason the CMEM kernel module was created—to provide physically contiguous buffers to be shared between the ARM and the DSP. This is also useful for buffers that are to be accessed using the DMA or the DM6446 H/W resizer.

The Linux partition is also used for various internal I/O buffers and application caching features, so the bigger this partition, the better.

1.4 CMEM: Contiguous Memory Allocator

CMEM was created to support sharing buffers between ARM Linux processes (application control) and the DSP (algorithm acceleration). CMEM takes a physical memory region you specify at module insertion time and carves it into pools of contiguous buffers according to your specifications. The buffers are not cached on the ARM side (but the Codec Engine handles the DSP side caching of these buffers).

How to use CMEM is described in sections that follow. Note that once you have inserted the *cmemk.ko* kernel module you can always execute "`cat /proc/cmem`" to get status on the buffers and pools managed by CMEM.

1.5 The DDRALGHEAP and DDR Sections

The *DDRALGHEAP* section contains the heap from which the active codecs allocate all their dynamic memory. This section can be quite large, especially if video codecs are used.

A new feature as of Codec Engine 1.20 is that you can now pass in a physically contiguous (CMEM-allocated) memory block to Codec Engine to be used as the DDRALGHEAP. See the Server_redefineHeap() API call. There are also new API calls for querying heap usage during run time: see Server_getNumMemSegs() and Server_getMemStat().

The *DDR* section contains DSP-side code and static data for all the codecs plus the system (that is, DSP/BIOS and Codec Engine). This section is called "DDR2" for CE 1.20 and later.

1.6 The DSPLINKMEM Section

The DSPLINKMEM section is used by the *DSPLink* IPC (Inter Processor Communication) software from TI. Codec Engine uses this software module for communicating between the ARM and the DSP, as well as for loading the DSP with code and controlling it.

1.7 The RESET_VECTOR Segment

The RESET_VECTOR section contains the DSP reset vector, which is the vector table that the DSP side ISTP register points to when the DSP is pulled out of reset by DSPLink. The reset vector code moves the vector table elsewhere by changing the ISTP, but this is where it is located at boot. This section needs to start at an even 1 MB and needs to be 128 bytes in size.

2 Designing the Memory Map

This section describes how to design your memory map to optimize the memory usage for your system. When this phase is completed, we will have a piece of paper that lays out the most compact memory map we can have, with names, origins, and sizes for each segment of the map needed by all the players in the system. That piece of paper we then use as the input for the next phase (described in Section 3), where we edit various text files to apply that information into the system's build.

Our motivation here is to make the memory needed by the Codec Engine, and its ARM-side support software like CMEM and DSPLink, as small as possible. This we want because any portion of the memory set aside for Codec Engine that is not used remains unused forever; whereas any amount of memory given to Linux will always be put to good use. Linux uses all the memory it can get for caching its disk and network data, so increasing the memory for Linux improves the overall performance of the system.

To put it in an equation,

```
total memory = DSP Server memory + CMEM memory + Linux Memory
```

from which follows that

```
Linux Memory = total memory (e.g. 64 MB) - DSP memory - CMEM memory
```

Since total memory is fixed and is determined by the hardware design of a board, we strive to give the DSP and CMEM only as much as necessary so that Linux gets as much as possible.

2.1 The Procedure

In essence, the procedure for determining the memory map is this: we make the system work with a luxurious memory map (for example, the original 256 MB). Then, for each segment we calculate or measure its actual requirements, reduce the segment size, and rerun the application. Once we have minimized the size of each segment, we compact the map and fit everything into the block of memory for the final system—128 MB, 64 MB, 32 MB, etc. Here’s a step-by-step overview:

1. Start with the original 256 MB memory map.
2. Include all the codecs needed in your DSP server (via the .cfg file), and rebuild the DSP server.
3. Determine the size of DDRALGHEAP using one of the following methods:
   – Calculate.
   – Run the ARM-side CE application and measure the worst case.
4. Determine the size of the DDR segment.
5. Determine the size of CMEM using one of the following methods:
   – Calculate.
   – Run the ARM-side CE application and measure the worst case.
6. Move RESET_VECTOR in the same MB as DSPLINKMEM or DDR.

7. Order the segments correctly and place them at proper start addresses.
8. Compute the Linux memory size = device total DDR2 memory – DDR size – DDRALGHEAP size – DSPLINKMEM (1 MB).

The following figure is an example of a before and after scenario when this procedure is applied:

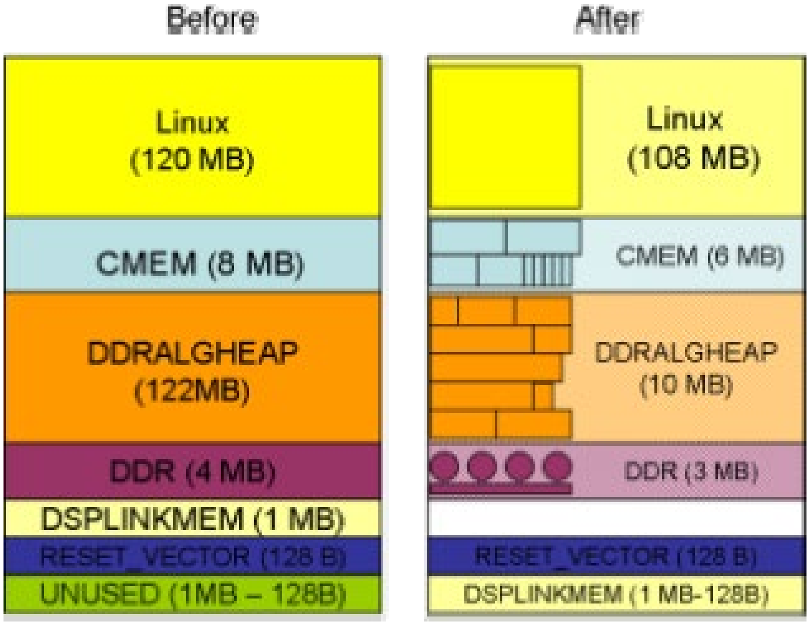

Figure 1. Before and After Memory Maps

In the "after" picture we have the actual minimum sizes for each segment. Knowing that, we can fit everything in a device with less system memory, 128 MB in the above example.

The folllowing section look at how to calculate or measure these sizes, but first we need to set some expectations.

2.2 It’s (Mostly) All About Video

Recall that the DSP server has four segments:

- Two smaller system segments called "DSPLINKMEM" and "RESETCTRL", with total size of about 1 MB (that can be shrunk to 512 KB if really necessary, as we mention in a later section).
- One medium-size segment, "DDR" (or "DDR2"), sized at typically 1-3 MB, that contains the code and static data for all the codecs plus the system.
- One large segment called "DDRALGHEAP", sized at anywhere from 2 MB to 200 MB, that holds all the dynamic memory allocated by each active codec instance running on the DSP.

The sizes of the first two segments are independent of the run-time characteristics of the system. The size of the DDR segment depends on the codecs included in the system, and this is fixed given the functionality you wish to support—include only the codecs you need and none of the codecs you don't. This requires a fixed amount of code and static data space. But the size of the DDRALGHEAP segment depends heavily on which instances of those codecs the ARM side creates and when.

When the system is first started, there are no codec instances and the total heap allocated in DDRALGHEAP is 0. When the ARM side creates a DSP codec, for example via `VIDENC_create()`, this instance allocates dynamic memory according to its spec sheets. The amount of memory usually depends on the codec creation parameters. For video processing in full resolution this may require several MBs of DDRALGHEAP for a single instance.

When an instance is deleted (for example, with `VIDENC_delete()`), all of its dynamic memory is reclaimed.

Video codecs (encoders and decoders) need by far the most dynamic memory, often several MBs, followed by imaging codecs, then audio codecs, followed by speech codecs, which typically need very little. Therefore, the way video codecs are used by the ARM side determines how big the DDRALGHEAP segment must be.

The CMEM segment's size is also very dependent on which codecs run in the system and when. The purpose of the CMEM segment is to exchange input and output codec data between the ARM and the DSP codec instances. Video buffers allocated via CMEM are much larger than speech buffers.

The bottom line is that the total required memory depends on how we use the codecs, and if we don't use all available codecs at the same time—which some classes of applications do and some don't—the total memory required will be less than the simple sum of the parts.

Here is an example to illustrate this: assume the system is a DaVinci-based digital video camera with a "video record" and a "video play" button. We could design the system to create, from the ARM side, a video encoder instance and video decoder instance on the DSP, at boot time, and have them run side-by-side. Both codec instances hold their dynamic data in DDRALGHEAP. When a user presses the "record" button, the ARM side passes raw images via CMEM to the video encoder. When the user presses the "playback" button, the ARM side passes the compressed frames and receives uncompressed images. The total of combined DDRALGHEAP and CMEM memory required for these two may be, say, 3 MB for decoder + 2 MB for encoder = 5 MB.

Alternatively, we can design the system to wait until the user presses the "record" button and then create a video encoder instance on the DSP, and delete the instance when the user presses the stop button. Similarly we create an encoder instance when the user presses the "playback" button, and delete it when he exits the playback mode.

Because we know that the user cannot record and playback at the same time, we know that the encoder instance and the decoder instance cannot exist at the same time. Therefore the total memory needs become MAX(3 MB, 2 MB) = 3 MB instead of 5 MB. Since creating a codec instance is a very fast operation, this method does not affect the system in terms of speed and power consumption.

2.3 Determining the Size of DDRALGHEAP

The total DDRALGHEAP size depends on which codecs (of which type, from which vendor) are used, how many instances of those codecs exist at the same time, and possibly with which parameters the codecs were created—for example, D1 vs. CIF video resolution changes the memory requirements.

In theory we can calculate the amount of memory needed by looking at the codec data sheets; those should list how much memory a codec instance requires based on the mode of operation.

In practice, it is better to actually measure the usage at peak time and only look at the specs to confirm that the expected numbers roughly match the measured results. This because the data sheets show the worst-case size requirements. Depending on your codec configuration, your requirements may or may not be less.

2.3.1 Measuring DDRALGHEAP Size via Engine_getUsedMem() API

This procedure applies to all versions of Codec Engine. The simplest way to measure memory usage is this:

1. In the ARM application, call `Engine_getUsedMem()` immediately after the first call to `Engine_open()`.
2. Call `Engine_getUsedMem(engineHandle)` again after creating the codecs with the heaviest memory requirements.
3. The delta between the two numbers is roughly the required size of DDRALGHEAP. The reported size is actually slightly larger than required as it includes the growth of DDR, but the latter grows only by a few KB per instance. For example, if the delta is 6.4 MB, the real DDRALGHEAP may be 6.395 MB.

2.3.2 Measuring DDRALGHEAP Size via Server_getMemStat() API

This procedure applies Codec Engine v1.20 and later.

The `Server_getMemStat()` API in CE 1.20 and later allows us to query each segment specifically, so we can do that for DDRALGHEAP.

Assuming the Codec Engine handle is in variable "hEngine", make the call below at peak load time (when worst-case codecs are created and active) to find out how big DDRALGHEAP need be:

```
hServer = Engine_getServer(hEngine);
Server_getNumMemSegs(hServer, &numSegs);

for (i=0; i<numSegs; i++) {
   Server_getMemStat(hServer, i, &memStat);

   if (strcmp(memStat.name, "DDRALGHEAP") == 0) {
       printf("DDRALGHEAP usage is %d out of %d available\n",
              memStat.size, memStat.used);
   }
}
```

2.3.3 Measuring DDRALGHEAP Size via External ALGUTIL Utility

This procedure applies to all versions of Codec Engine. A collection of tools, called “servertools” is available at https://www-a.ti.com/downloads/sds_support/applications_packages/servertools/index.htm. This collection contains, among others, a utility to instrument the DSP server and determine the exact memory needs of each algorithm for specific types of memory it needs. Follow these steps:

1. Download the tools and locate *algUtil*, which is a DSP-side utility module that prints out the memory allocated for algorithms on the heap as it is instantiated.

2. Insert the module into the codec server using `xdc.useModule('ti.sdo.apps.algutil.ALGUTIL')` in *myServer.cfg*.

3. Call `ALGUTIL_init()` in the DSP codec server *main()* function.

4. Enable tracing in your CE application, and enable algUtil tracing when invoking the application as follows. For more details on CE trace, see the CE documentation.

```
TRACEUTIL_DSP0TRACEMASK="ti.sdo.apps.algutil.ALGUTIL=4"; ./myapp
```

5. Invoke the application with one algorithm instance created at a time until data has been collected for all algorithms. The output has two lines of special interest similar to the following:

```
@0x000898fd:[T:0x8fc45144] ti.sdo.apps.algutil.ALGUTIL - EXTERNAL scratch total:best
case:(0x0), worst case:(0x0)
@0x000899a6:[T:0x8fc45144] ti.sdo.apps.algutil.ALGUTIL - External persist total:best
case:(0x29d5c), worst case:(0x29d94)
```

6. Add up the heap usage of your "worst case codec combination" to determine your total heap requirement.

2.4 Determining the Size of the DDR Section

The DDR segment holds codec and system code as well as static data. (It is called "DDR2" in CE 1.20 and later, but it is the same segment.) We find its minimum required size simply by looking at the linker map. The procedure is as follows:

1. Add all the codecs you intend to use to your DSP server's .cfg file, and no others.
2. Build your DSP server.
3. Look at the generated .map file for the codec server, and see how much is used for DDR. In the following example, it is 0x90168 bytes. The *.map* file is under the directory `package/cfg`.

```
         name              origin    length      used     unused   attr
----------------------  --------  ---------  --------  --------  ----
  ARM_RAM               10008000   00004000  00000000  00004000  RWIX
  CACHE_L2              11800000   00010000  00000000  00010000  RWIX
  CACHE_L1P             11e08000   00008000  00000000  00008000  RWIX
  L1DSRAM               11f04000   00010000  00010000  00000000  RWIX
  CACHE_L1D             11f14000   00004000  00000000  00004000  RWIX
  DDRALGHEAP            88000000   07a00000  07a00000  00000000  RWIX
  DDR                   8fa00000   00400000  00090168  0036fe98  RWIX
  DSPLINKMEM            8fe00000   00100000  00000000  00100000  RWIX
```

4. Add a little to the "used" value to allow code and data to grow some during development.

2.5 Sizing and Partitioning CMEM Memory

The module we call CMEM, as we have seen, enables us to allocate large chunks of physically contiguous memory from Linux-ARM and place data buffers in them for the DSP to process. There are two aspects to configuring CMEM:

- Knowing the total amount of memory we need for buffer exchange between ARM and DSP. This is the CMEM size.
- Knowing the exact size and count of each type of exchange buffers the application needs. This gives us the CMEM partitioning into pools of buffers.

Again, we can calculate or measure required CMEM sizes, and it is always best to do both -- using the calculations to verify the measurements.

2.5.1 Calculating CMEM Size and Partitions

As an example, assume we run one video encoder at D1 resolution and two audio encoders. To exchange buffers with these codecs, the video encoder needs one input D1-sized buffer for the raw image and one output buffer for the encoded frame. For each audio codec, we need one input buffer for raw audio data and one output buffer for compressed audio.

The size of the raw D1 image buffer we calculate knowing the format of the image; let us assume it's 812 KB. The size of the output video encoder buffer depends on the compression format, but typically it is recommended to be the same as the input buffer, that is 812 KB in this case. As for the audio, assume we similarly use a 4 KB input and 4 KB output buffer for each codec.

Our total CMEM need is then: (`812 KB x 2) + (4 KB x 2) + (4 KB x 2) = 828 KB`.

Our CMEM pool needs the following: one pool with 2 buffers of 812 KB, and one pool with 4 buffers of 4 KB:

```
 [.....812K.....][.....812K. ... ][4K][4K][4K][4K]
```

CMEM pool partitioning is important; if done improperly, it prevents us from getting the buffers we need, even if there is enough total space. (This is the disadvantage of pools; the advantage is that it prevents fragmentation where it could happen, that is, if the application were allocating and releasing many buffers of different sizes constantly.)

In this example, we have one video encoder and two audio encoder instances, all running at the same time. Let us take a look at how the exchange occurs:

1. Before the application starts, the system integrator loads the CMEM module. Assuming the CMEM area starts at 0x88000000, and is sized as above, the command to load the module is:

```
insmod cmemk.ko phys_start=0x88000000 phys_end=0x88200000 pools=2x831488,4x4096
```

   (We set aside a full 2 MB for CMEM, and split it into 2 x 812 KB and 4 x 4 KB buffer pools.)

2. The ARM application allocates its two 812 KB buffers and four 4 KB CMEM buffers via `Memory_contigAlloc()`;

3. The ARM application stores a raw image block in one 812 KB buffer and raw audio blocks in two 4 KB buffers. It passes the 812 KB video buffers in a call to `VIDENC_process()` and the 4 KB audio buffers in calls to `AUDENC_process()`. It reads the compressed video and audio frames from their buffers.

4. When the application closes, it frees up all of its CMEM buffers via `Memory_contigFree()`.

Imagine now that in addition to all of the above, the application also uses a video decoder, but never at the same time as the video encoder. (Recall the video camera example that supports record and playback modes but only one at a time.) Assuming the video decoder also processes D1-sized images—getting compressed frames and producing raw images—we'd need two 812 KB buffers for the video decoder as well. But since we never have both the encoder and the decoder processing their input data at the same time, we can use the same 812 KB input and output CMEM buffers. Therefore the total CMEM needs remain the same, and even the partitioning looks the same.

Note that it is not even necessary that the ARM application destroys one video codec on the DSP before it switches to another: the two instances can be active on the DSP at the same time, but if we never call `VIDENC_process( input 812KB buf, output 812KB buf )` while we are waiting for `VIDDEC_process(input 812KB buf, output 812KB buf)`, we are safe. We create and destroy codec instances on the DSP as needed only in order to save on memory needs on the DSP for DDRALGHEAP.

Now for a counter example. An application may need both record and playback running at the same time. In that case, the total CMEM needs are 2 x 812 KB (video encode) + 2 x 812 KB (video decode) + 2 x 4 KB (audio encode) + 2 x 4 KB (audio decode) which is 3264 KB. The command line to load the CMEM module would then be:

```
insmod cmemk.ko phys_start=0x88000000 phys_end=0x88340000 pools=4x831488,4x4096
```

(We set aside 3.25 MB, slightly larger than the 3.2 MB we need.)

2.5.2 Measuring CMEM Size and Partitions

In rare cases, your application’s use of input and output buffers is so complex that it is easier to measure your application’s needs in terms of CMEM memory by running the application itself.

1. Start with CMEM module loaded and partitioned to allow for plenty of memory and plenty of buffers: one or two really big ones, a few large ones, a number of medium ones, and many small ones. For example:

```
insmod cmemk.ko phys_start=0x88000000 phys_end=0x8A000000
pools=2x4100000,10x1100000,50x130000,100x17000
```

   This example creates two 4 MB+ buffers, ten 1 MB+ buffers, fifty 128 KB+ buffers, a hundred 16 KB+ buffers, for a total of 27 MB+, rounded to 32 MB.

2. At your application’s peak memory usage time, or several times through its life cycle, put the following in your C code and record the output:

```
system( “/bin/cat /proc/cmem” );
```

Accessing /proc/cmem as shown in the previous step causes CMEM to produce detailed information regarding exactly how many buffers it uses and of what sizes. This gives you the precise statistics for accurate sizing and partitioning.

Of course, it is always good to relax the accurate numbers, whether they are measured or calculated. Application code may change and a new or larger buffer may be needed at certain moments, but updating the "insmod cmemk.ko" line may be neglected. We use engineering intuition to decide how much headroom we want to leave in terms of numbers and sizes of the CMEM buffers the application needs. (It is also worth noting that alignment and Linux page boundary requirements may require the total area to be larger than the sum of the parts.)

2.6 Optional: Using CMEM to Dynamically Size DDRALGHEAP

In CE 1.20 and above, the new APIs Server_redefineHeap() and Server_restoreHeap() let you change the DSP-side heaps at run time. The following requirements must be met:

- The memory passed to Server_redefineHeap() needs to be contiguous (CMEM allocated).
- The heap needs to be created in the DSP server's DSP/BIOS configuration file, but can be 0 bytes initially.

These APIs let you “reuse” the memory used for DDRALGHEAP when the system is doing less stressful DSP tasks, for example, ARM-side PDF file reading.

To use this feature, you need to allocate an extra buffer of the size of DDRALGHEAP in the CMEM segment, on top of the requirements you have determined from Sections 2.5.1 and 2.5.2.

2.7 Optional: Reducing the DSPLINKMEM Segment Size from 1 MB to 512 KB

The DSPLINKMEM segment on the DSP is the system segment needed by DSPLink. By default, approximately 512 KB of the 1 MB segment is used for shared buffers and control structures. The defaults in the CE examples are larger to anticipate potential extra memory required by CE in future releases. For simplicity, it is OK to avoid modifying the defaults for DSPLink system components, but if you need to save an extra 512 KB, you can do so by reducing the size of this segment, without worrying about details of DSPLink itself.

2.8 Arranging the Segments in Correct Order and Alignment

If we have followed the outlined procedure, we now have, on paper, the minimal measured and/or calculated size for each segment: CMEM, DDRALGHEAP, DDR, and possibly DSPLINKMEM if we decided to cut it in half. Segment RESETCTRL has a fixed size of 128 bytes.

At this point, we now need to decide what the start address of each segment will be. Follow these steps:

1. Know (or decide) how much total system memory you have: 64 MB or 128 MB, etc.
2. Place the RESETCTRL segment at the highest-addressed 1 MB in the map. That is, this segment must be 1 MB aligned and we choose it to be the very last MB in the memory map.

3. Place the DSPLINKMEM segment immediately after RESETCTRL, so it gets 1 MB – 128 B (which is still fine). Now the last MB is occupied by RESETCTRL + DSPLINKMEM.
4. Place DDR before RESETCTRL. Try to make DDR's size at least a multiple of 4 KB if you can't make it more even. That is, don't use a size like 2,432,131 bytes for the DDR size. Instead, use 2.5 MB. For example, 0x280000 bytes, not 0x251C83 bytes. If possible, leave a larger amount of memory for DDR than required in case you modify the code in the future, resulting in a code size increase. This will give you the convenience of not having to shift everything in your memory map just to accommodate small code size changes.
5. Place DDRALGHEAP before DDR. Again, use round hex numbers for origin and size and leave a safety margin if you can. Such round numbers help you avoid alignment surprises and make the map easier for humans to understand and maintain.
6. Place CMEM before DDRALGHEAP. Again, use round hex numbers if you can.
7. Linux gets the rest of the memory.

It is advisable to make RESETCTRL + DSPLINKMEM occupy the last 1 MB of memory, especially if you use DSPLink 1.30. Then you will only have to rebuild DSPLink once (which is a tedious procedure and you don't want to repeat it more than absolutely necessary).

Also, it is convenient to have CMEM and DDRALGHEAP adjacent to each other, so you can resize one at the expense of the other without touching other segments. Both CMEM and DDRALGHEAP are normally unused when there are no active codecs, even though the DSP may be up and running and ready to create a codec when instructed.

If you are using the new feature in CE 1.20 to dynamically pass a CMEM buffer to the DSP to serve as its DDRALGHEAP, simply combine DDRALGHEAP into the CMEM segment.

As the final result of this phase, your drawing on a piece of paper may look something like this:

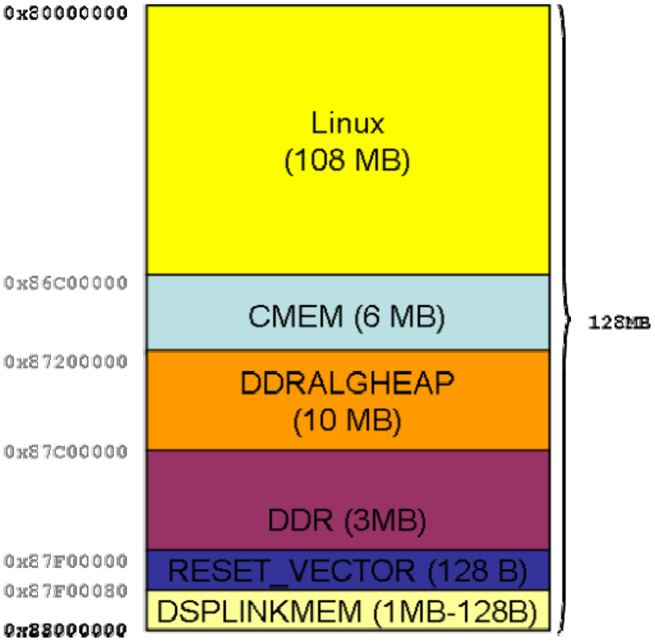

Figure 2. Example Memory Map Plan After Calculations

3 Memory-map Adaptation Instructions

This section provides a procedure to follow in order to set up the memory map you have designed. As an example, we have modified the *video_copy* example in Codec Engine to match the following memory map:

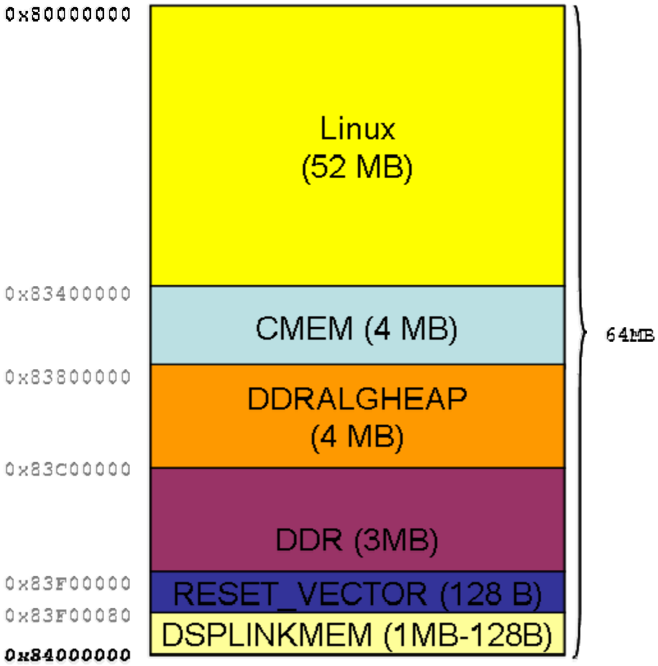

Figure 3. Modified video_copy Memory Map

For those of you who are familiar with the video_copy example, there is obviously a lot more memory here allocated to each segment than necessary for the application to run. However, the goal of this example is to simply show the files in which changes need to be made, so that you can better follow the steps we outline below.

The modified video_copy code example available with this application note comes in two flavors:

- DSPLink 1.30 and CE 1.02 based
- DSPLink 1.40 and CE 1.20 based

See the readme.txt file that accompanies the code for details on the contents.

To unzip the example into a directory of your choice, either use the `unzip` command in Linux or use WinZip in Windows.

When following this procedure for your own application, simply replace the numbers with your own memory map, and use the corresponding sizes and base addresses for it.

3.1 Determining the Version of Codec Engine

Knowing the version of CE (and DSPLink) is important, because it determines the steps to follow. CE 1.20 (or above) uses DSPLink 1.40+, while versions of CE prior to that uses DSPLink 1.30+.

If you do not know which version of CE you are using, you can easily find it out by looking at the name of the CE installation directory. For example, if the directory name is codec_engine_1_20, it means you are using CE 1.20. When this document was written, DVEVM and DVSDK software versions up through 1.20 were bundled with versions of CE older than CE 1.20.

3.2 Rebuilding DSPLink 1.30

DSPLink is a software component that enables the ARM and the DSP to communicate. Version 1.30 of DSPLink requires rebuilding the entire DSPLink when the DSP memory map is changed. DSPLink version 1.40, which is used by Codec Engine 1.20 and above, is more dynamic and requires no rebuilding.

If your DSPLink version is 1.40 or higher, you can skip to the next section. There is one much simpler replacement step you need to do when you configure the application; it is described in Section procedure, in Section 3.5, “Rebuilding the ARM-side Application—If You Use DSPLink 1.40”.

Rebuilding DSPLink is the most involved step in the sequence. Its sub-steps are listed here:

1. Move (cd) to the `<DVEVM>/dsplink_1_30_*/packages/dsplink`directory. All the paths in the remainder of this section are given relative to this directory.
2. Open the DSPLink configuration file in a text editor: `config/all/CFG_Davinci.TXT`.
3. Search for the "RESUMEADDR" text entry. You will see, by default, the value of 0x8FF00020. Change that number to the beginning of your RESET_VECTOR segment + 0x20. In this example, it should be 0x83F00020.
4. Search for the “RESETVECTOR” entry. Change its value to the beginning your RESET_VECTOR segment, which is 0x83F00000 in this example.

Note: Large hex numbers with lots of zeroes are commonly mistyped to omit one zero! Make sure the hex number is exactly eight characters wide.

5. Search for the "MEMTABLE0" set of entries. There you will find some entries that resemble your memory map, and some that don't. The ones to look for are DSPLINKMEM, RESETCTRL (same as RESET_VECTOR) , and DDR. Change their addresses (ADDRDSPVIRTUAL and ADDRPHYSICAL, which are the same) and sizes to match your new memory map. Do not worry that DDRALGHEAP isn't there—that’s because DSPLink doesn't need to know about it since its content only exists while the DSP runs and is never accessed by the ARM. The resulting MEMTABLE0 entries for this example are as follows:

```
[MEMTABLE0]

[0]
ENTRY            | N |    0                  # Entry number
ABBR             | S |    DSPLINKMEM         # Abbreviation of the table name
ADDRDSPVIRTUAL   | H |    0x83F00080         # DSP virtual address
ADDRPHYSICAL     | H |    0x83F00080         # Physical address
SIZE             | H |    0xFFF80            # Size of the memory region
MAPINGPP         | B |    TRUE               # Map in GPP address space?
[/0]

[1]
ENTRY            | N |    1                  # Entry number
ABBR             | S |    RESETCTRL          # Abbreviation of the table name
ADDRDSPVIRTUAL   | H |    0x83F00000         # DSP virtual address
ADDRPHYSICAL     | H |    0x83F00000         # Physical address
SIZE             | H |    0x00000080         # Size of the memory region
MAPINGPP         | B |    TRUE               # Map in GPP address space?
[/1]

[2]
ENTRY            | N |    2                  # Entry number
ABBR             | S |    DDR                # Abbreviation of the table name
ADDRDSPVIRTUAL   | H |    0x83C00000         # DSP virtual address
ADDRPHYSICAL     | H |    0x83C00000         # Physical address
SIZE             | H |    0x00300000         # Size of the memory region
MAPINGPP         | B |    TRUE               # Map in GPP address space?
[/2]
```

Do not worry about other segments listed in the file.

6. Edit the file `make/Linux/davinci_mvlpro4.0.mk`, which contains DSPLink build instructions for its ARM binaries, on a Linux host. Edit the following fields to match your DVEVM installation, noting the location of the Linux kernel and the ARM compiler tools:

- `BASE_BUILDOS`: location of the Linux kernel. The directory usually ends with "/lsp/ti-davinci".
- `BASE_CGTOOLS`: location of the ARM tools. The directory usually ends with " arm/v5t_le/bin".

7. Edit the file `make/DspBios/c64xxp_5.xx_linux.mk`, which contains DSPLink build instructions for its DSP binaries, on a Linux host. Edit the following fields to match your DVEVM and DSP/BIOS installation:

- `BASE_SABIOS`: location of your DSP/BIOS installation. The directory usually ends with "/bios_5_21_01" or another version number.
- `BASE_CGTOOLS`: location of your C64P compiler tools that run on Linux. The directory can end in different ways, but it invariably contains subdirectories "bin", "include", and "lib".

8. Set the environment variable DSPLINK to the directory `<DVEVM>/dsplink_1_30_*/packages/dsplink`.

9. From the current ($DSPLINK) directory, type:

```
gmake -C gpp/src
gmake -C dsp/src
```

10. Find the newly-built DSPLink kernel module in `gpp/export/BIN/Linux/Davinci/RELEASE/dsplinkk.ko`, and copy it to your DVEVM file system.

These steps should build a link server configured specifically for the memory layout you need. Keep in mind that if you ever build multiple servers, this build of DSPLink won't work for them anymore!

If you have more than one server, and they have different memory configurations, one approach you may use is to clone the entire top-level DSPLink directory under a different name, then apply all the previous steps in that directory. You will then have a DSPLink build dedicated entirely to one specific memory map.

If you chose to do so, remember that you must specify which DSPLink build you are using in the "XDCPATH", that would be the *xdcpaths.mak* file in Codec Engine examples if you build just Codec Engine examples, and Rules.make file in DVEVM installation directory if you build real DSP servers. The kernel module (*dsplinkk.ko*) also applies to just one specific memory layout.

Because of this complexity, DSPLink 1.40 eliminates all these steps and only uses one *dsplinkk.ko* kernel driver and one build for any DSP memory layout.

3.3 Rebuilding the DSP Server

Every DSP server has a DSP/BIOS configuration file (*.tcf* file) that defines the memory layout on the DSP, among other things. It also has a Codec Engine configuration file (*.cfg* file) that lists what codecs to include in the image.

Our DSP server is found in the Codec Engine *examples/servers/video_copy* (or *examples/ti/sdo/ce/examples/servers/video_copy* in more recent versions of CE). Please adjust the path appropriately for the remainder of this procedure.

The server configuration file (*video_copy.cfg*) lists what codecs to include. There are only two in the list, and we need both, so we don't change anything in this file.

However, if the codecs were real, the first step would be to edit this file and remove all the codecs we don't need. That would reduce the size of the DDR segment and allow us to make it shorter than the default of 4 MB. The only file we need to edit right now is the *video_copy.tcf* file. If you open that file in a text viewer, you will see that it imports the contents of another DSP server's .tcf file (all_codecs.tcf) because the contents are the same for both servers. Since we want to modify the video_copy example only, do the following:

1. Move (cd) to the Codec Engine `examples/servers/video_copy` directory.

2. From inside the *video_copy/* directory, copy *../all_codecs/all.tcf* to *video_copy.tcf*.

3. Edit *video_copy.tcf* and edit the "mem_ext" array for the newly-chosen memory map. That code should look like this:

```
var mem_ext = [
{
    comment:    "DDRALGHEAP: off-chip memory for dynamic algmem allocation",
    name:       "DDRALGHEAP",
    base:       0x83800000,   // 56 MB
    len:        0x00400000,   //  4 MB
    space:      "code/data"
},
{
    comment:    "DDR: off-chip memory for application code and data",
    name:       "DDR",
    base:       0x83C00000,   // 60 MB
    len:        0x00300000,   //  3 MB
    space:      "code/data"
},
{
    comment:    "RESET_VECTOR: off-chip memory for the reset vector table",
    name:       "RESET_VECTOR",
    base:       0x83F00000,   //  63 MB
    len:        0x00000080,   // 128 B
    space:      "code/data"
},
{
    comment:    "DSPLINK: off-chip memory reserved for DSPLINK code and data",
    name:       "DSPLINKMEM",
    base:       0x83F00080,   // 63 MB + 128 B
    len:        0x000FFF80,   //  1 MB - 128 B
    space:      "code/data"
},
];
```

Note! CE 1.20 uses "DDR2" instead of "DDR" in its examples.

4. Since we have defined and sized DDRALGHEAP to be a space that is solely used to store algorithm memory requests, you should change all sections, including BIOSOBJSEG, MALLOCSEG and STACKSEG to use DDR as follows (if not already done):

```
/* ==========================================================
 *  Set all data sections to use DDR
 *  ==========================================================*/
bios.setMemDataNoHeapSections (prog, bios.DDR);
bios.setMemDataHeapSections (prog, bios.DDR);

/*  ==========================================================
 *  MEM : Global
 *  ==========================================================*/
//prog.module("MEM").BIOSOBJSEG = bios.DDR;     //comment line out if present
//prog.module("MEM").MALLOCSEG  = bios.DDR;  //comment line out if present

/*  ==========================================================
 *  TSK : Global
 *  ==========================================================*/
//prog.module("TSK").STACKSEG = bios.DDR;        //comment line out if present
```

3.4 Optional: Splitting the DDR Section to Reduce Trampoline Occurrences

Trampolines are linker-generated function calls that allow code to make jumps to points further than 4 MB apart (that is, far calls). These can occur on DSP servers that are large in terms of code and data size, when points of code that call each other are separated by large chunks of static data and other code. Trampolines may cause some performance loss or other problems; consequently, it is better to tell the linker to place all code apart from all data, to minimize the number of trampolines necessary, given that the code would then be more compact. Separating code from data may be beneficial for other reasons as well (for example, better cache use).

To find out how much code vs. data you have, you can use the “cg_xml” tool, which is available at https://www-a.ti.com/downloads/sds_support/applications_packages/cg_xml/index.htm. It is a collection of Perl scripts, which complements the codegen tools, providing more information about compiled binaries.

One of the scripts provided as part of cg_xml is sectti.pl, which lists all output sections in a given compiled binary. You can run this script on your DSP image as follows:

```
ofd6x –x myImage.x64P | perl sectti.pl
```

The script summarizes your code and data sizes. For example:

```
------------------------------------------------------------
Totals by section type
------------------------------------------------------------
   Uninitialized Data :     961084  0x000EAA3C
     Initialized Data :     462114  0x00070D22
                 Code :    1169504  0x0011D860
```

The totals may need to be adjusted to discount sections not placed in DDR. For instance, DSP/BIOS may add sections marked as type “N/A” that are neither counted as data or code. If they are placed in the DDR section, they need to be added to the total data memory size.

Coming back to the .tcf file, to separate code from data, split the DDR segment into two segments: DDR that contains data only, and DDRCODE that contains code only.

To carve off a portion of DDR for strictly for code placement, change the above DDR declaration in the “mem_ext” array to the following instead:

```
{
    comment:     "DDR: off-chip memory for data",
    name:        "DDR",
    base:        0x83C00000,
    len:         0x001B0000, // 1.7MB
    space:       "code/data"
}, {
    comment:     "DDRCODE: off-chip mem. for code",
    name:        "DDRCODE",
    base:        0x83DB0000,
    len:         0x00150000, // 1.3MB
    space:       "code/data"
}
```

Your effective memory map would then become:

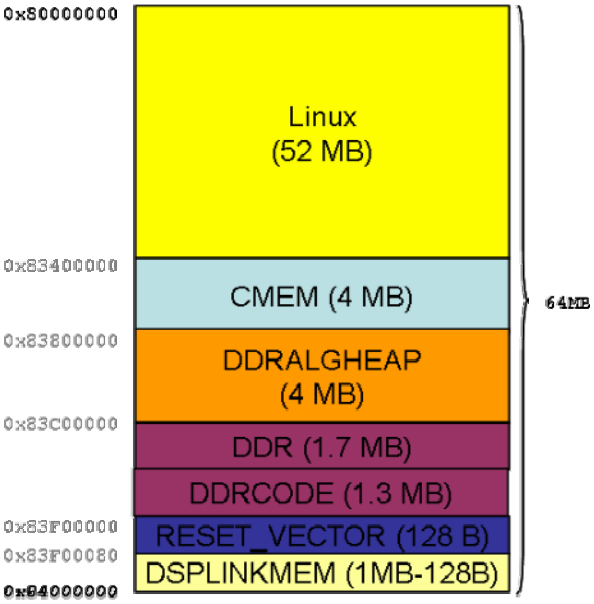

Figure 4. Resulting Memory Map After Splitting DDR

Note that it is not necessary to inform DSPLink of this change, as DDR and DDRCODE are contiguous and can be treated as a monolithic segment of writeable external memory for the DSPLink loader.

After splitting DDR, change the following line, which places most code sections in DDRCODE:

```
bios.setMemCodeSections (prog, bios.DDRCODE);
```

Running *sectti.pl* from the Code Generation Tools XML Output Utility Scripts on the resulting executable can show all code sections that are not yet placed in DDRCODE. All sections in the video_copy example are already placed using the line above. However, in case your application has some extra non-placed sections (due to the section name missing the prefix '.text:'), here's an example output from sectti.pl for your reference:

```
Name         :  Size (dec)     Size (hex)    Type    Load Addr    Run Addr
----         :  ----------      ----------    ----    ----------   ---------
.randomCode :  54816           0x0000d620    CODE    0x8fd99740   0x8fd99740
```

Assuming you have a section similar to .randomCode that lies outside of DDRCODE, you would need to manually place it in DDRCODE in the server's link.cmd file.

```
SECTIONS {
    .randomCode > DDRCODE
}
```

After you are finished with all necessary modifications, save and close the file. Rebuild the server by typing these commands from the current directory:

```
make clean
make
```

Then copy the rebuilt server image (video_copy.x64P) to your target file system.

3.5 Rebuilding the ARM-side Application—If You Use DSPLink 1.40

Users of DSPLink 1.40 did not have to rebuild Link, but they have to rebuild their ARM-side application. Users of DSPLink 1.30 can skip this section.

The change to be made is in examples/apps/video_copy/dualcpu/ceapp.cfg, the application configuration file. It needs a configuration file setting that specifies what the memory map is.

1. Open the ceapp.cfg file and add or otherwise make sure the following code exists in the file:

```
osalGlobal.armDspLinkConfig = {
    memTable: [
        ["DDRALGHEAP", {addr:  0x83800000,  size:  0x00400000,  type:  "other"}],
        ["RESET_VECTOR", {addr: 0x83F00000, size: 0x00000080, type: "reset"}],
        ["DDR2",  {addr:   0x83C00000,   size:   0x00300000,   type:   "main"  }],
        ["DSPLINKMEM", {addr: 0x83F00080, size: 0x000FFF80, type: "link" }],
    ],
};
```

2. Save and close the ceapp.cfg file.
3. Rebuild the application by executing make.

3.6 Copying Other Necessary Files to Target File System

In the final steps, we copy the remaining bits and pieces of the video_copy application to the target file system:

1. Move (cd) to the Codec Engine `examples/apps/video_copy/dualcpu/` directory. This is where the ARM application is located.

2. Copy the app.out executable to the target file system. Note that you do not have to rebuild it if you have not changed the Linux kernel supplied with the DVEVM/DVSDK software (unless you use DSPLink 1.40).

3. Copy the in.dat file, a sample input file for the application, from the current directory to the target filesystem.

4. Have your *cmemk.ko* CMEM kernel module available on your target file system. You must have rebuilt it for your Linux kernel in order to run any other Codec Engine application. If you haven't changed your Linux kernel, you can use a copy of cmemk.ko in the Codec Engine examples/apps/system_files/davinci directory.

5. If you are using DSPLink 1.40, copy your dsplinkk.ko kernel module to your target file system. You might have rebuilt it for your Linux kernel in order to run any other Codec Engine application. If you haven't changed your Linux kernel, you can use a copy of dsplinkk.ko in the Codec Engine `examples/apps/system_files/davinci` directory.

6. Have your kernel modules loading script (*loadmodules.sh*) available on your target file system. You can also find a copy of the script in the Codec Engine `examples/apps/system_files/davinci` directory.

3.7 Modifying the loadmodules.sh Script

The *loadmodules.sh* script loads the kernel module *dsplinkk.ko* and tells it where to put the DDR and DDRALGHEAP segments. That is the only flexibility DSPLink 1.30 allows without rebuilding. The DDR segment can be anywhere and of any length, and can be announced to DSPLink at the time the kernel module is loaded. DDRALGHEAP can be anywhere and of any length. However, the DSPLINKMEM and RESET_VECTOR segments cannot be moved or resized without rebuilding DSPLink.

1. Edit the *loadmodules.sh* script and remove the arguments following the “insmod dsplinkk.ko” text, so the command says:

   ```
   insmod dsplinkk.ko
   ```

   A note about the *loadmodules.sh* and *dsplinkk.ko* arguments:

   - *DSPLink 1.30* supports an optional argument pair ddr_start and ddr_size, which allow you to load and run DSP images that have a different DDR segment than the default. However, it still expects the DSPLINKMEM and RESET_VECTOR segments to match DSPLink 1.30's configuration file. This is useful when you want to experiment with increasing the DSP image's size of the DDR segment at the expense of other segments’ sizes (excluding DSPLINKMEM and RESET_VECTOR), or vice versa, without having to go through the process of rebuilding DSPLink 1.30 every time. However, it doesn't help if you need to change the limits of the entire memory map. In this case, having changed the text configuration file and rebuilt DSPLink, the ddr_start and ddr_size arguments are no longer necessary since the DSPLink 1.30 memory map configuration matches the memory map of the DSP image—though you can still use them if you subsequently want to experiment with changing the position and size of DDR.
   - *DSPLink 1.40* supports dynamic memory map configuration, so changing the start address and size of the DDR section is only a matter of modifying the ARM-side application’s .cfg file and rebuilding the application itself. Hence there is no need to specify these parameters in *loadmodules.sh*.

2. Next, you change the CMEM memory description that follows as the arguments to the `insmod cmemk.ko` command. Specify *phys_start* and *phys_end* to match your new CMEM address and size, then specify pools to match the buffer requirement of your application. The pools are configured using an `NxSize` syntax where N is the number of buffers in the pool, and Size is the size of these buffers. For the video_copy example, the following configuration would be more than sufficient for CMEM:

   ```
   insmod cmemk.ko phys_start=0x83400000 phys_end=0x83800000
   pools=20x4096,10x131072
   ```

3.8 Changing the Boot Srgument in your Linux Bootloader

When the Linux kernel is booted, we limit the amount of physical memory available to the kernel using the mem=boot argument. If you use uboot, change that portion of the bootargs variable to read `mem=52M` using the setenv command as follows:

```
> setenv bootargs ‘console=ttyS0,115200n8 root=/dev/nfs mem=52M
nfsroot=192.168.1.101:/opt/montavista/pro/devkit/arm/v5t_le/target,nolock’
```

Note! This step is critical—if Linux tries to use memory above 52 MB, it will corrupt the CMEM data and the data will corrupt the kernel. That would likely result in a quick crash.

3.9 Rebooting and Running the Application

After the system boots, type the following:

```
sh loadmodules.sh
./app.out
```

Look for this line of application output to confirm the procedure worked:

```
Application finished successfully.
```

4 Troubleshooting

If you did everything in the previous procedures correctly, the application should run and you can skip this section. Otherwise, this section provides a few troubleshooting tips to find out more about your system with the new memory map.

4.1 Checking How Much CMEM Memory is Available or Used

After running *loadmodules.sh*, directly enter the following command at the command prompt in Linux. This shows whether you have set up CMEM with the correct buffer pools.

```
/bin/cat /proc/cmem
```

To verify the amount of memory allocated from the CMEM pools at any point in your application execution (for example, when a Memory_contigAlloc call fails), you can add this line to your ARM-side application’s source code:

```
system( “/bin/cat /proc/cmem” );
```

4.2 Memory Map Mismatch

If there is a mismatch between the memory map used by the DSP/BIOS *.tcf* file and the DSPLink configuration, often this would result in a failure in the first `Engine_open()` call, which loads the DSP with the DSP server executable. If this occurs, compare the settings in the mem_ext array in the *.tcf* file with the ones in the DSPLink configuration (which resides in the *CFG_Davinci.txt* file in DSPLink 1.30 or in your application's *.cfg* file in DSPLink 1.40). It is possible there is a mismatch between the two.

In fact, it is a good practice to double-check the two configurations after you go through all the steps in the procedure of memory map configuration. It can save valuable debugging time.

4.3 Heap Sizes Too Small

One common problem is that estimated heap sizes might be too small. This could happen if the sizes were underestimated or miscalculated, resulting in memory allocation failures on the DSP.

Looking at the CE trace files and locating the point of failure should give you some indication about which heap ran out of space. For example, if the trace shows a failure while creating an algorithm using DSKT2 (part of the *Framework Components*), this points to a potential lack of space in the DDRALGHEAP. If the failure occurs while allocating/creating some other object, then it is likely that the DDR heap is too small. You can turn on the highest verbosity level in CE trace by specifying the following command line when running your executable:

```
CE_TRACE="*=01234567" TRACEUTIL_DSP0TRACEMASK="*=01234567"
TRACEUTIL_DSP0TRACEFILE="cedsp0log.txt" CE_TRACEFILEFLAGS="w"
CE_TRACEFILE="cearmlog.txt" TRACEUTIL_REFRESHPERIOD=200 ./app.out
```

Replace *app.out* with the name of your application executable. This should produce two log files corresponding to the ARM and the DSP which you can inspect after the application runs. More details on how to use the CE trace can be found in the *Codec Engine Application Developer User's Guide*.

5 A Real World Example

This section describes an actual case in which a customer tried to resize their memory map.

The application is a four-channel CIF MPEG4 Simple Profile (or H.264 Baseline Profile) Digital Video Recorder (DVR) based on the DM6446 with 64 MB of DDR2. Four channels of CIF video are encoded and one channel of CIF video is decoded. The MPEG4 (or H.264) and audio encoder and decoder conform to xDM.

In this discussion, we will focus on the video codecs. So, four CIF encoding instances and one CIF decoding instance will be created by calling the VISA API's. This example is based on Codec Engine 1.02 and DSPLink 1.30.08.02.

Here is the system block diagram:

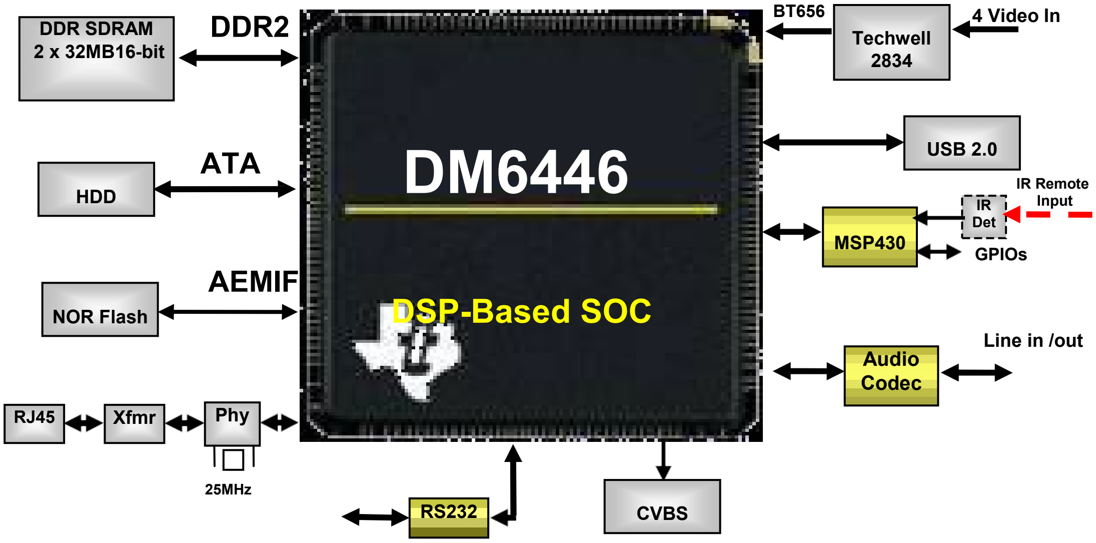

Figure 5. System Block Diagram

The final 64 MB memory map looks like this:

```
0x80000000 .. 0x83200000-1 (0-50MB; size 50MB): Linux: booted with MEM = 50M
0x83200000 .. 0x83A00000-1 (50-58MB; size 8MB): CMEM: shared ARM/DSP I/O buffers
0x83A00000 .. 0x83C00000-1 (58-60MB; size 2MB): DDRALGHEAP: codec dynamic memory
0x83C00000 .. 0x83E00000-1 (60-62MB; size 2MB): DDR: code, stack, system data
0x83E00000 .. 0x83F00000-1 (62-63MB; size 1MB): DSPLINKMEM: memory for DSPLINK
0x83F00000 .. 0x83F00080-1 (63-63MB; size 128B): RESET_VECTOR: reset vectors
0x83F00080 .. 0x84000000-1 (63-64MB; size 1MB): Unused memory
```

How did we arrive at this memory map? First of all, 1 MB DSPLINKMEM is the default size for DSPLink 1.30.08.02. It is important to correctly allocate the right memory size for CMEM, DDRALGHEAP and DDR. Then we will have enough space for DSP software and the Linux OS.

The next diagram illustrates the system application data flow.

- **Encoding data flow.** VPFE puts the video input data (to-be-encoded data) in CMEM. The encoder running on the DSP outputs the encoded data to CMEM. The encoded data is stored on the hard disk.
- **Decoding data flow.** To-be-decoded data is copied from the hard disk to CMEM first. Then, the decoder decompresses the data and outputs the result into decoded data buffers. The VPBE output resolution is CIF. Sometimes, we can use the resizer of the VPFE peripheral on DM6446 to get D1 resolution. So we need to allocate a buffer for resizer results in CMEM too.

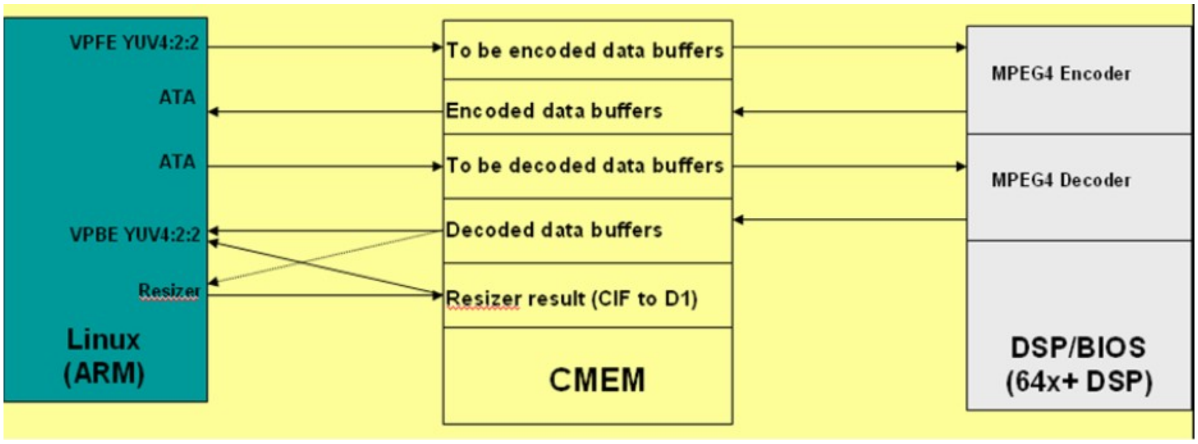

Figure 6. System Application Data Flow

5.1 Allocating CMEM Memory Space

As for to-be-encoded data buffers, the size is `((352 * 288) * 4 * 2B) * 3 = 2433024 B`. 352*288 is CIF resolution, 4 means 4 channels, one pixel in YUV4:2:2 needs 2 bytes and three `(352 * 288) * 4 * 2B` buffers are allocated for the encoder algorithm. The size of decoded data buffers is same: 2433024 bytes.

Because we encode 4 channels CIF, you can calculate the size of encoded data buffers by 50% D1 `((720 * 576 * 3 / 2 ) / 2 = 303.75 KB`, YUV4:2:0) or the standard MPEG4 compression ratio. Here we allocate 256 KB (262144 bytes) for encoded data buffers less than 303.75 KB. This is chosen based on experience. So, we configure three 256 KB buffers (786432 bytes) for to-be-decoded data buffers accordingly. As for the buffer of resizer result, we need `720 * 576 * 2B = 829440B` in YUV4:2:2. So, the *insmod cmemk* command looks like the following:

```
insmod cmemk.ko phys_start=0x83200000 phys_end=0x83A00000 pools=1x262144,
2x2433024, 1x829440,1x786432
```

5.2 Allocating DDRALGHEAP Memory Space

DDRALGHEAP is the memory allocated for codec dynamic memory requests.

Both encoder and decoder process and accept data with YUV4:2:0. One channel CIF data in YUV4:2:0 is `352 * 288 * 3 / 2 B` (one pixel with YUV4:2:0 format needs 3/2 byte). Encoder and decoder algorithms need the current frame and previous frame data. To compress or decompress one channel CIF, we need to allocate `352 * 288 * 3 / 2 * 2 B` memory for encoder and decoder respectively.

Four channels CIF will be encoded and one channel CIF will be decoded. So, the encoder needs `352 * 288 * 3 / 2 * 2 * 4B` (about 1.16 MB) of dynamic memory and the decoder needs `352 * 288 * 3 / 2 * 2 * 1B` (about 297 KB) of dynamic memory. The total of them is about 1.45 MB. To allow some extra space, 2 MB DDRALGHEAP is allocated in this example.

5.3 Allocating DDR Memory Space

DDR is the DSP-side segment including all the system code, data, stack, heaps and code and static data for the codecs. The code size for the most complex video codecs is less than several hundred KBs. We can use the script *sectti.pl* to determine DDR section size as follows:

```
ofd6x -x codec_server.x64P | perl c:\temp\cg_xml\ofd\sectti.pl >
codec_server.x64P.sectti.csv
```

The script generates a report file showing that we can get about 416 KB of the totals of data and code. So 2 MB DDR of this application is enough.

```
REPORT FOR FILE: codec_server.x64P
          Name :         Size (dec)  Size (hex)  Type     Load Addr   Run Addr
          MPEG4ENC :     23840       0x00005d20  CODE     0x83c71000  0x83c71000
          MPEG4DEC :     10784       0x00002a20  CODE     0x83c82000  0x83c82000
          .bss :         910         0x0000038e  UDATA    0x83c88000  0x83c88000
          .hwi_vec :     512         0x00000200  CODE     0x83c70c00  0x83c70c00
          .far :         204920      0x00032078  UDATA    0x83c00000  0x83c00000
          .bios :        22912       0x00005980  CODE     0x83c76d20  0x83c76d20
          .text :        123136      0x0001e100  CODE     0x83c52080  0x83c52080
          .cinit :       8196        0x00002004  DATA     0x83c84a20  0x83c84a20
          .sysinit :     1792        0x00000700  CODE     0x83c70180  0x83c70180
          .const :       21288       0x00005328  DATA     0x83c7c6a0  0x83c7c6a0
          .stack :       4096        0x00001000  UDATA    0x83c86a28  0x83c86a28

Totals by section type (about 416KB)
Uninitialized Data: 212958  0x00033fde
 Initialized Data : 30080  0x00007580
            Code : 182976  0x0002cac0
```

5.4 Allocating Linux OS Memory Space

We computed sizes by calculating the DSP needs first and subtracting that from the total amount of memory available. We know the production system has only 64 MB of memory. Given that we need 1 MB for DSPLINKMEM, 2 MB for DDR, 2 MB for DDRALGHEAP, 1 MB for RESET_VECTOR and unused memory and 8 MB for CMEM, that gives a total of 14 MB for DSP and shared buffers, leaving 50 MB for Linux.

6 Conclusion

Memory map configuration for DaVinci-based systems can be systematically performed after you have designed the memory map to suit the amount of memory available. In order for the procedure to go smoothly, a reminder is to:

- **Know your system.** Plan the memory map based on how many and which codec instances will need to be available at the same time in different modes of execution in the application. Calculate or measure the size for each segment and write down the desired memory map.
- **Be thorough.** Apply the mechanical steps to adapt the DSP server, ARM application, DSPLink, CMEM and boot loader to match the desired memory map. Always double-check the changes to ensure all numbers agree with each other.

7 References

- *Codec Engine Application Developer User's Guide* (SPRUE67)
- *Codec Engine Server Integrator User's Guide* (SPRUED5)
- *Codec Engine Algorithm Creator User's Guide* (SPRUED6)
- *DSP/BIOS Link User Guide* (LNK 058 USR)

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