Application Report
SPRAB87–December 2009

TMS320DM365 Preview of Codec Porting on Linux

Yashwant Dutt and Karimulla Shaik *MM Codecs*

ABSTRACT

TI codec software was traditionally run on a DSP, which had DSP/BIOS™ software kernel foundation as an operating system. With the new range of upcoming SOCs, where the architecture is getting more hardware accelerator centric, the use of digital signal processing (DSP) is reduced. The codec can operate directly from the main processor, which also hosts the primary operating system. Therefore, the interaction of codecs with operating system integrities becomes inevitable. DM365 is one such where the Host ARM926 runs the operating system and application and also has a portion of codec software. This application report discusses in detail the various steps to make the codec work in such an environment. It considers Linux® as the operating system running on the Host ARM926. This application report is written with respect to the DM365 codec and software architecture, but would also be helpful to users who want to port their Code Composer Studio™ standalone/other OS-based codec/algorithm on Linux on DM365 or related platforms.

Contents

List of Figures

List of Tables

1 Introduction

DM365 is a digital multimedia SOC that is primarily targeting video security, video conferencing, PMP and other related applications. It has an ARM subsystem, two video co-processor sub-systems and scores of peripherals. This application report is also helpful to users who want to port their Code Composer Studio standalone/other OS-based codec/algorithm on Linux. For more details, see the *TMS320DM365 Digital Media System-on-Chip (DMSoC) Data Manual* (SPRS457).

As part of the software offering in the DM365 digital video software development kit (DVSDK), the ARM subsystem hosts a wide range of codec and integrated sample demo application running on Linux.

Codecs are important components in the overall software ecosystem. Many times the codec and application run on Linux, which is an open-source operating system that comes with a reasonable licensing agreement.

This application report discusses the various issues and points of porting a codec on Linux.

Section 2.1 provides the overview of the DM365 codec software. The sections following that detail the changes needed in the memory, compiler, and general interactions from Linux perspective. The last section discusses a few debugging tips, which are helpful when running the codec on Linux.

2 DM365 Software Architecture

The DM365 DVSDK consists of multiple components stitched together in a demo application. It includes Linux as the operating system, device drivers, codecs, framework component and a demo application.

2.1 Overview

DM365 codecs running on Linux interact with FC for resource allocation and fulfilling its operating system specific needs. The framework component, in turn, interacts with the operating system drivers using Linux utilities making the codec implementation operating system agonistic. Figure 1 explains the interaction of codec related software components in the Linux system.

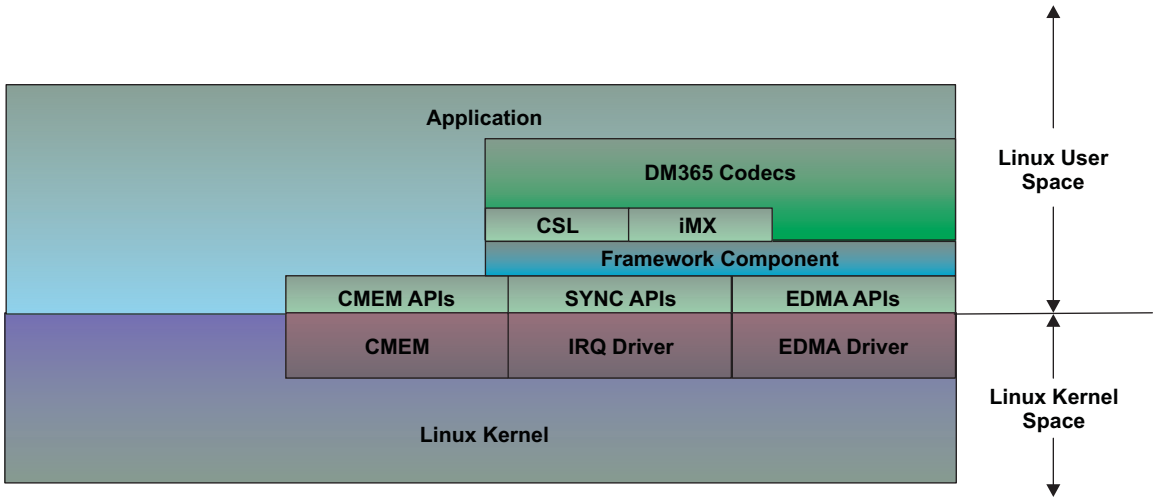

Figure 1. DM365 Software Architecture

2.2 Codec Software Architecture

In DM365, the bulk of the processing is done on the accelerators: video image co-processor(VICP) or high-definition video image co-processor(HDVICP); only the frame level operation is done on Host ARM. Figure 2 explains host to coprocessor communication.

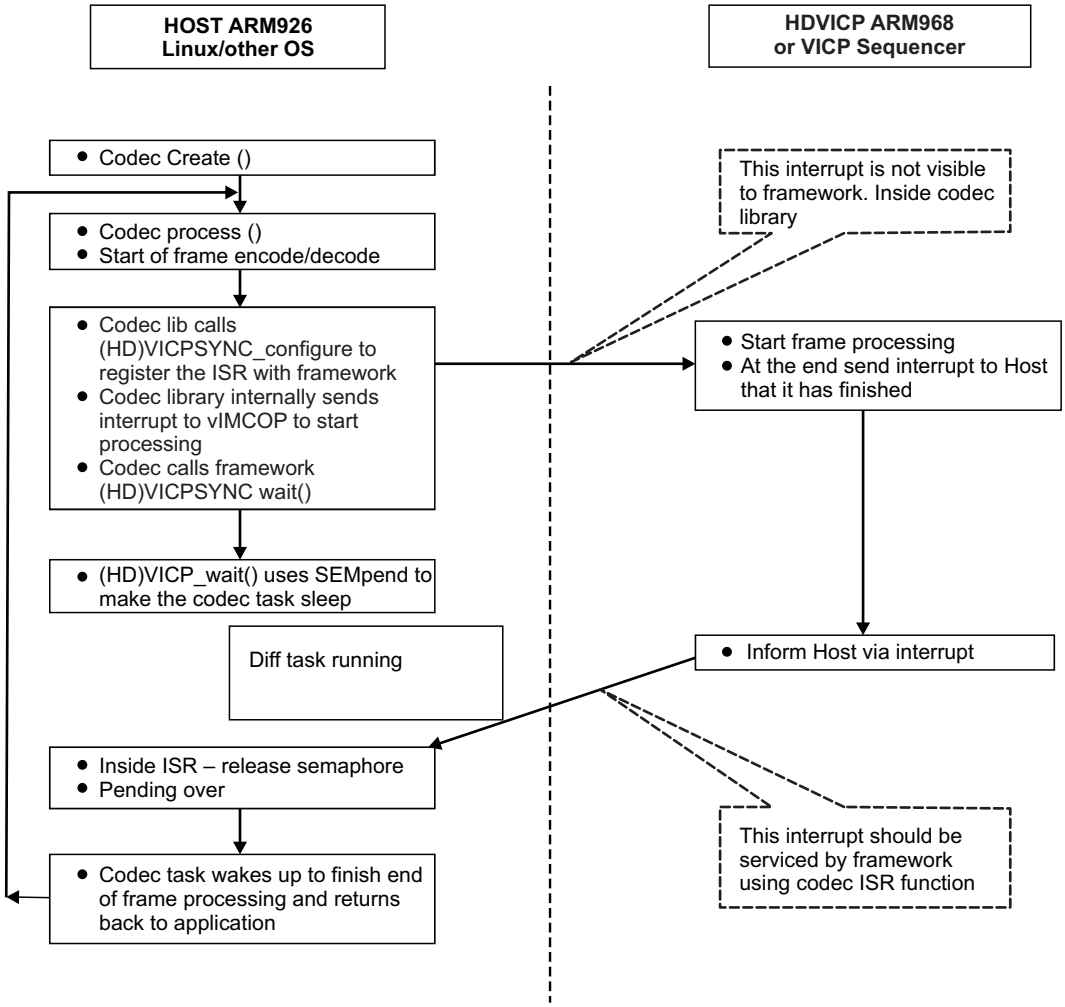

Figure 2. DM365 Codec Architecture, Host ← → Co-Processor Communication

The host processor runs the mainstream operating system like Linux. Since the frame level module of the codec runs on the Host, the codec has to be the operating system aware. But the codec software is still OS agonostic as it interacts with the framework component for its resource and operating system specific needs, HDVICP or VICP, where the bulk of frame processing happens, also has an ARM/sequencer but not an operating system. Therefore, code running in HDVICP/VICP interacts directly with the hardware.

The following sections explain the various points to take when porting codec on the Linux operating system environment.

3 Memory Related Changes Inside and Outside the Codec

This section describes the various memory related changes that need to be done when running the codec in Linux.

3.1 Virtual and Physical Address

All software running on Linux operates in virtual address space. But the HDVICP ARM968 or the IMCOP sequencer would need addresses in physical space for its operation. Similarly, DMA param space also needs source and destination addresses in physical space. Therefore, you need to maintain dual address space in the codecs.

Memory access that needs virtual address:

- Code running on ARM926 and doing memory access using CPU

Memory access that needs physical address:

- DMA source and destination on both ARM926 and ARM968
- All memory access in ARM968

When HDVICP, VICP or EDMA resources are requested by framework components, it provides virtual-base address pointers for these resources. To facilitate getting the physical address of the various addresses, framework component provides an API that converts physical address to virtual address.

- API to convert virtual to physical address: * void MEMUTILS_getPhysicalAddr(* void)

For an optimized implementation, whenever possible, it is advisable to store all the needed physical address in some internal data structure so that multiple calls to this API are avoided, as the API intern calls the Linux system call is more cycle consuming.

3.2 Memory Allocation Using CMEM

The current *MEMUTILS_getPhysicalAddr* can translate virtual addresses, which are allocated using CMEM. Therefore, all the memory allocation in the codec application needs to be done using CMEM.

- All memTabs[ ], the memory requests of codec during its instance creation needs to be through CMEM.
- I/O buffers passed to the codec during process call should be allocated using CMEM.

3.3 Static Table Inside the Code

During compilation of codec, static or constant tables get placed in the data section. The data section is in DDR memory. If you want to DMA this table to some internal memory of VICP/HDVICP, you need to have the physical address of these tables. MEMUTILS cannot be used for address translation as the memory is not allocated using CMEM.

A workaround for this is as follows:

1. Compute the size of the total tables, which goes into .data section. You can use the sizeof() operator over the table arrays to find the total size.
2. Request for CMEM memory using MemTab[ ] at the time of codec instance creation for the size computed in .
3. Copy the tables from .data section to the memory allocated using MemTab using memcpy() at the time of instance initialization.
4. Assign new pointers to the tables and store them in some codec internal data structure. Use these new table pointers further in code.

Since memcpy() happens only during creation time, the overhead is limited to application startup. DMA of the table to internal VICP/HDVICP memory can be done at create or process call time depending on the use case. In case of single instance operation, the DMA can be done at the create time. In case of multiple instance or different codec format, one needs to DMA the table at every process calls time.

4 Compilation

For running on Linux, the ARM926 library needs to be compiled using the gcc tool chain. In case the previous version of the library was compiled using the TI ARM tool chain, some changes have to be made to compile using gcc.

In HDVICP codecs, the ARM926 library compilation is done using gcc and HDVICP ARM968 compilation is done using TI ARM tool chain.

Below are the list of such changes; the list may not be exhaustive:

- U64 (unsigned long long) and its variant data type is compiler dependent. Every compiler has its own way of alignment for U64. This has to be carefully treated. In case a structure containing U64 is compiled both in gcc and TI ARM tool chain, the structure alignment and individual element offset may be different in case it contains U64 variable in between. When structures are DMAed form ARM926 to ARM968 DTCM, the de-referencing of structure element may go wrong. Therefore, it is advisable not to use non-standard data type like U64.
- Assembly intrinsics are tool chain dependent. Many of the intrinsics that work fine in the TI ARM tool chain do not work with gcc. An equivalent intrinsic needs to be used or replaced with C code.
- Any header file inclusion path that contains forward slash should be replaced with backward slash.
- Avoid case mismatches as the gcc compilation is case sensitive while header files inclusion.

5 Interaction With Linux System

Codec interacts with Linux during its operation. Below are the points that need to be taken.

5.1 System Resource Access

When Linux runs on ARM926, it owns chip resources similar timer and interrupts. The software design should exercise some constraint while using those resources.

- In case ARM968 tries to access the timer register through the config bus for its internal needs, i.e., profiling, it may disrupt the normal functioning of the kernel and put it into an unpredictable state. Only shared resources like enhanced direct memory access (EDMA) should be accessed inside ARM968.
- Code running on ARM926 should not directly modify interrupt and other system registers. You should only access the system resource by codec if you are sure that it is not owned by Linux.

5.2 Booting of Co-Processors

Linux initialization powers up all of the modules of the DM365, which is controlled using the PSC block. But it may not do any operation that requires the setting of some registers internal to the modules (e.g., VICP/HDVICP). Quite often, these initializations are done in gel files used when running the codec using Code Compser Studio software. You need to make a C equivalent function for the same and call it inside code. As an example, if you need to access VICP buffers inside HDVICP codec, it requires clocking of some internal module of VICP. This is not done as part of the standard Linux boot. Therefore, such initializations need to be done inside codec when running the code in Linux.

For initializations done inside codec when running the code in Linux, do the following steps:

1. Check the initiation done in the gel file when running the code using Code Compser Studio.
2. Find out which of the gel files is done in the standard Linux boot.
3. Do the necessary changes in the codec, for the remaining.

5.3 Interaction With Kernel Modules (KO)

As explained earlier, DM365 interacts with FC to fulfill its operating system related needs. FC interacts with Linux utilities. The Linux utilities needs to interact with Linux in kernel mode; there is need of inserting kernel modules when codecs are running. The kernel modules used for DM365 codec are the following: *cmem*, *edma* and *irq*.

These kernel modules need to be inserted using the insmod command before running the codec. Some kernel modules like cmem also needs details of pool allocation for memory allocation. This also needs to be done while inserting the kernel module.

6 Debugging Tips

Debugging in Linux environment is different from that of Code Composer Studio. The following are the ways that can be used for debugging.

6.1 Use gdb for Debugging on ARM926

To debug on Linux (ARM926), you need to use command line gdb. You can use the various command line gdb options to view, step through, and other debug operations. There are several aspects that can make debug easy.

The LSP gdb does not allow you to view the memory allocate using CMEM. This is a known limitation since codec memory is generally allocated using CMEM, this becomes a problem while debugging. The following steps explain a workaround solution to debug.

1. Create a global variable object of the structure whose memory is allocated using CMEM (temporary - for debug purpose only). Since the structure object is a global variable, its memory would be allocated in the .bss section.
2. Make a function that copies the CMEM allocated structure into a normal .bss section structure.
3. Call the function that copies to the new structure whenever you want to see the variable state in the original CMEM state function, and you can see the variable state in the original CMEM state function.

   The following is code snippet:

```
struct data {
  int big;
  int foo;
  long moo;
};

struct data * myData = `pointer to CMEM memory`;
struct data b;

Then in gdb:

(gdb) define bd
Type commands for definition of "bd".
End with a line saying just "end".
>call memcpy(&b, &myData, sizeof(struct data))
>print b.$arg0
>end
(gdb) bd foo
```

In ARM926, CPU accesses to memory are using virtual address. There are some tools that can be used to view content of registers/memory in physical address space. To view the memory of the various physical locations while the debug is in progress, you can open a telnet terminal to the evm and use the window in parallel.

6.2 Use Code Composer Studio for Debugging on ARM968

When Linux is running on ARM926, you can still connect Code Composer Studio to ARM968 and debug. To do this, you need the correct Code Composer Studio driver, which supports both ARM926 and ARM968. The following procedure explains how to connect Code Composer Studio to ARM968:

1. Open the Code Composer Studio parallel debug manager (assumption is that the JTAG and other Code Composer Studio related drivers/connections are installed and are working correctly).
2. Run the codec using gdb on the Linux command prompt.
3. Insert a break point just before sending the interrupt to HDVICP and Run.
4. Once the control reaches the breakpoint, Connect ARM968 (do not connect ARM926).
5. Load the debug version of the code, which is supposed to run on ARM968. Insert a breakpoint at the desired location.
6. Press F5 on ARM968. The code starts executing and waits for an interrupt from ARM926.

7. Type “c” on the gdb prompt to Continue debug. This sends an interrupt to ARM968 and the control stops at the breakpoint on ARM968. Continue the debug on ARM968.

Since the bulk of processing happens on ARM968, this method provides a good debugging methodology.

7 References

- *TMS320DM365 Digital Media System-on-Chip (DMSoC) Data Manual* (SPRS457)
- H.264 High Profile Decoder on DM365 User's Guide (SPRUEV0)
- H.264 High Profile Encoder on DM365 User's Guide (SPRUEU9)

IMPORTANT NOTICE

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

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

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

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

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

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

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

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

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

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