Application Report
SPRAA19A — August 2005

Power Management in an RF5 Audio Streaming Application Using DSP/BIOS

Vincent Wan,
Ezell Young
Texas Instruments

ABSTRACT

Embedded applications commonly run in an environment where power consumption by the DSP and its peripherals needs to be minimized. However, since DSP programs also need to meet time deadlines, DSP power management can be a challenge.

The example provided with this application note showcases several power management techniques you can leverage in a DSP/BIOS-enabled application. As with most software techniques, no one size fits all. You should decide which techniques make sense for your own application.

The application makes use of the DSP/BIOS PWRM module, and is based on Reference Framework Level 5. As provided, the application performs audio streaming on a 5509A EVM PLUS board.

Contents

Figures

Tables

1 The Need for DSP Power Management

Power management is a hot area of research these days, as devices become smaller, lighter, and more feature-loaded. Beyond protecting the environment, there are many practical reasons to be interested in minimizing power use in embedded software applications:

- To allow a portable, battery-operated device to function longer between charges.
- To reduce heat generation.
- To allow for increased component density, resulting in a reduction in product size.
- To reduce the cost of a product.
- To reduce battery costs for users.
- To obtain approval from the FCC.

Since the DSP is a core component in your design, it is important to minimize power consumption by the DSP and the peripherals it controls. DSP programs generally need to meet hard, critical time deadlines; consequently, power management can be more of a challenge on a DSP device. It is important that power management techniques not result in an application that misses real-time deadlines, and great care is needed to ensure that such problems do not occur.

Fortunately, Texas Instruments is providing support for DSP software power management to simplify the process of saving power. At the heart of this is the DSP/BIOS Power Manager (PWRM) module, which provides kernel-level APIs to reduce the power consumption of your application.

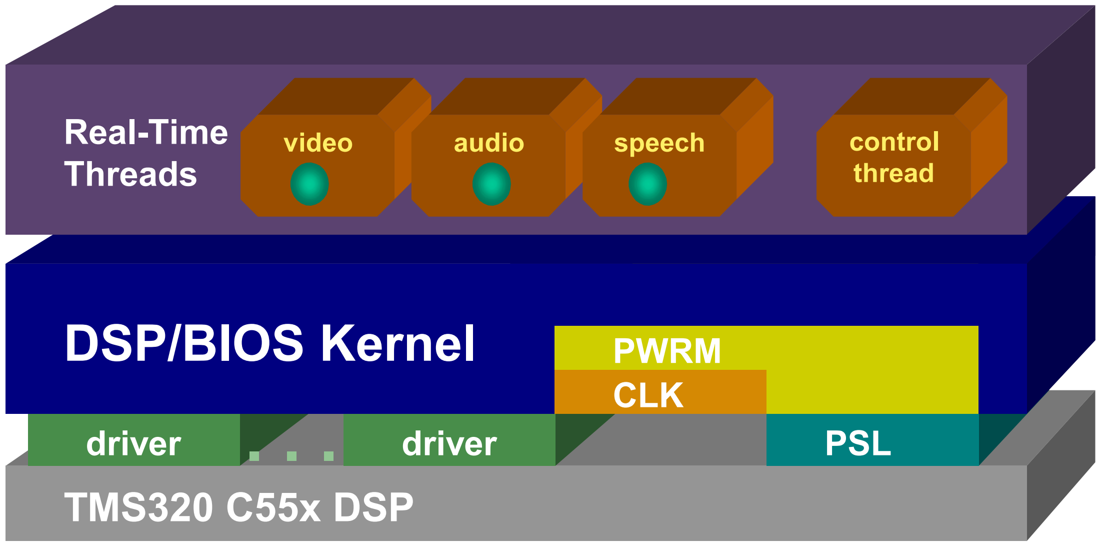

Figure 1. PWRM in an Application's Architecture

The PWRM module is a DSP/BIOS module that lets you reduce the power consumption of your application. By providing a set of easy-to-use APIs, it helps any DSP/BIOS user add power management into their design. PWRM interfaces directly to the DSP hardware by writing and reading configuration registers, and through a platform-specific Power Scaling Library (PSL) that controls the CPU clock rate and voltage-regulation circuitry. The PSL isolates PWRM and the rest of the application from low-level implementation details of the frequency and voltage control hardware.

A highly relevant paper is "Power-Optimizing Embedded Applications", *Proceedings of the Embedded Systems Conference* (See Reference 1). It contains ideas for including power management as a core consideration in both your hardware and software designs, which are the foundation for some of the techniques described in this application note. Details on the PWRM module can be found in *TMS320C5000 DSP/BIOS Application Programming Interface (API) Reference Guide*, SPRA404J, and *TMS320 DSP/BIOS User's Guide*, SPRU423F (References 2 and 3). Documentation for the PSL module can be found in Reference 4.

This application note describes power-saving techniques that can be applied using the PWRM module. An audio-streaming application is supplied as an example to illustrate the use of these techniques and of the PWRM module.

1.1 Common Terms and Principles

The total power consumption of a CMOS circuit is the sum of both active and static power consumption (see Reference 7 for more detail):

$P_{total} = P_{active} + P_{static}$

Active power consumption occurs when the circuit is active, switching from one logic state to another. Active power consumption is caused both by switching current (that needed to charge internal nodes) and through current (that which flows when both P and N-channel transistors are both momentarily on). Active power consumption is related to voltage and frequency by the following relation:

$P_{active} \sim C \times F \times V^2 \times N_{sw}$

where C is the dynamic capacitance, F is the switching frequency, V is the supply voltage, and N_{sw} is the number of bits switching. An additional relationship is that voltage (V) determines the maximum switching frequency (F) for stable operation.

Static power consumption is the component that occurs even when the circuit is not switching, due to transistor leakage. It is related to voltage as follows:

$P_{static} \sim V \times I_q$

where V is the supply voltage and I_q is the leakage current.

2 Running the Example Application

The audio-streaming application provided with this application note is based on the Reference Framework Level 5 example application (See Reference 5). It demonstrates techniques you can apply to your next power-aware DSP program. It leverages DSP/BIOS—and the PWRM module in particular—to control power consumption by the DSP and its peripherals.

2.1 Software Requirements

The following software is needed to run the audio-streaming application:

- Code Composer Studio (CCS) 3.10 or greater.
- DSP/BIOS 5.20: https://www-a.ti.com/downloads/sds_support/targetcontent/bios/bios_5_20/index.html
- C5509A Board Support Package: http://c5000.spectrumdigital.com/evm5509a
- Reference Frameworks v2.2 or greater: http://dspvillage.ti.com/docs/catalog/software/details.jhtml?templateId=5121&path=templatedata/cm/swdetail/data/rframe_source

2.2 Hardware Requirements

The following software and hardware are needed to run the audio-streaming application:

- TMS320VC5509A EVM PLUS board
- 2-4 multimeters (In theory one is enough. However, with fewer multimeters, you will need to move them around to perform all power measurements.)
- CD player (or any audio device)
- Speakers
- 1 stereo audio cable

2.3 Hardware Setup

The hardware for running the example application and measuring power usage should be set up as shown in Figure 2. Only two multimeters are shown; they can be moved when performing absolute DSP voltage and absolute supply voltage measurements as described in the list that follows the figure.

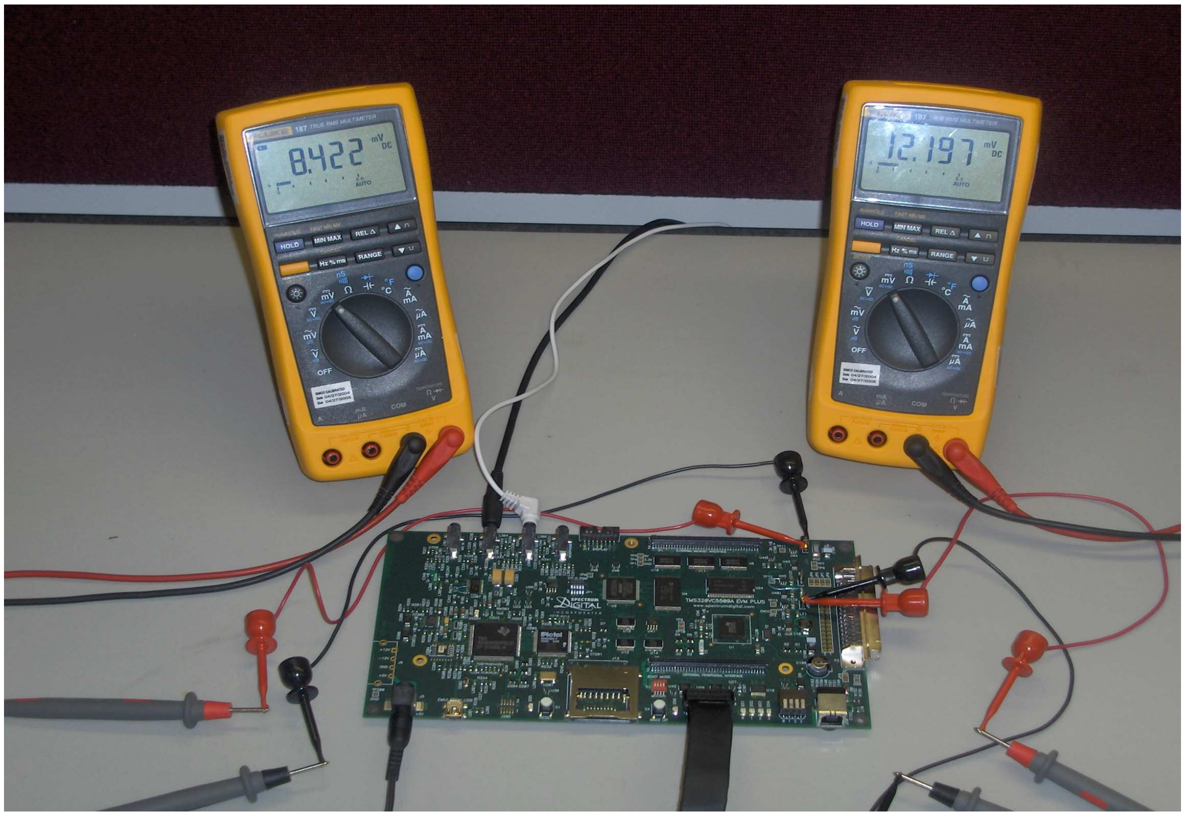

Figure 2. Hardware Arrangement for Example Application

- Connect the CD player or audio source to the line-in jack of the EVM 5509A using the audio cable.
- Connect the line-out jack to a pair of stereo speakers (or some other audio output device).

- JP2 lies on the power rail of the board's 5V supply, and JP3 lies on the DSP core's power rail. A current shunt is supplied on the board for these jumpers for measurement purposes. JP2 has a shunt resistor of 0.025 ohm. JP3 has a shunt resistor of 0.1 ohm.
- Connect one multimeter across JP2 and another across JP3 . This allows you to measure power for both the DSP and the board itself. By measuring the voltage drop across the shunt resistors, you can deduce the amount of current fed to the DSP and the overall system.
- To obtain absolute DSP core voltage measurements, connect a third multimeter across one end of the resistor on JP3 and one of the ground test points (for example, TP1).
- To obtain absolute board supply voltage measurements, connect a fourth multimeter across one end of the resistor on JP2 and one of the ground test points (for example, TP11).
- The "Wake Up" switch (S4) generates an interrupt to test the sleep mode.

2.4 Building the Application in CCStudio

To build the application using Code Composer Studio, follow these steps:

1. Install the Reference Frameworks at a location to be referred to as *RF_DIR*. Next, unzip the zip file that accompanies this application note at the same location, on top of the Reference Frameworks. Select "Yes to all" when prompted to overwrite existing files.
2. Open CCStudio Setup to configure the C5509A EVM board. Point it to use the startup GEL file sd5509evm.gel supplied in the referenceframeworks\apps\rf5_pwrm\projects\evm5509a folder. This ensures that the external memory interface is properly initialized. Refer to the CCStudio documentation if you are unsure about how to do this.
3. Use Code Composer Studio to open the app.pjt file in the *RF_DIR*\referenceframeworks\apps\rf5_pwrm\projects\evm5509a folder.
4. Ensure the BIOS_INSTALL_DIR environment variable points to your DSP/BIOS 5.20 installation directory. This is used to find the latest PSL libraries.
5. Click the Rebuild All button in CCStudio to rebuild the project.

2.5 Running the Application in CCStudio

Using Code Composer Studio, load the app.out file, which is located in the *RF_DIR*\referenceframeworks\apps\rf5_pwrm\projects\evm5509a\Debug folder. Put all DIP switches on the target board into their default position as per Table 1 in Section 3.1 before running the application.

When performing power measurements, be sure to select the **Debug->Run Free** menu command and then the **Debug->Disconnect** menu command in CCStudio instead of using **Debug->Run** to run the program. The reason for using **Debug->Run Free** is to ensure minimal interaction of the board with emulation, thus improving the accuracy of the power measurements. The **Debug->Disconnect** command is currently needed as a work-around to allow the target to disconnect properly for power measurements (SDSsq43297).

When running the application, the DSP/BIOS Real-Time Analysis tools operate in stop mode without RTDX support. You can rely on tools such as the Message Log, Statistics View and Kernel Object View for debugging only when you halt the processor. It is highly recommended that you disable RTDX when using power reduction features on a DSP, since emulation changes power measurements and could potentially disrupt the process of a DSP going into low-power (sleep) mode.

As of the publication date of this application note, STS statistics do not work when the frequency is scaled to a value different from the default of the DSP (SDSsq36471).

3 Application Structure

This section describes various aspects of the structure of the example power management application.

3.1 Data Path and DIP Switches

The data path is essentially the same as the data path for the RF5 application. An audio frame from the input codec is split into buffers for the left and right channels. These buffers are sent to two separate processing channels. The processed data is joined into a single buffer and sent out to the output codec hardware.

DIP switches (S2) on the target board are used to control the application. The can be used to activate or deactivate various power saving features.

Table 1. DIP Switches (S2)

DIP Switch	Function in Down Position	Function in Up Position	Default Position
0	Run in G.726 mode	Run in passthrough mode	Upward
1	Gate off clock domains in DSP/BIOS idle loop	Leave clock domains on in DSP/BIOS idle loop	Upward
2	Turn on calibration	Turn off calibration	Upward
3	Put DSP in deep sleep	n/a	Upward

DIP switch 0 controls the data path. In passthrough mode, two passthrough channels running the lightweight VOL (volume) algorithm are used. In G.726 mode, the channels are substituted with ones that run the G.726 encoder and decoder algorithms. See Figure 3.

DIP switch 1 controls clock domain idling in the DSP/BIOS idle loop. See Section 4.2, "Gating Off Clock Domains During DSP/BIOS Idle Loop" for details.

DIP switch 2 controls calibration. See Section 4.3, "Voltage and Frequency Scaling" for details.

DIP switch 3 can be used to put the DSP in deep sleep mode. See Section 4.4, "CPU Sleep and Wake-Up" for details.

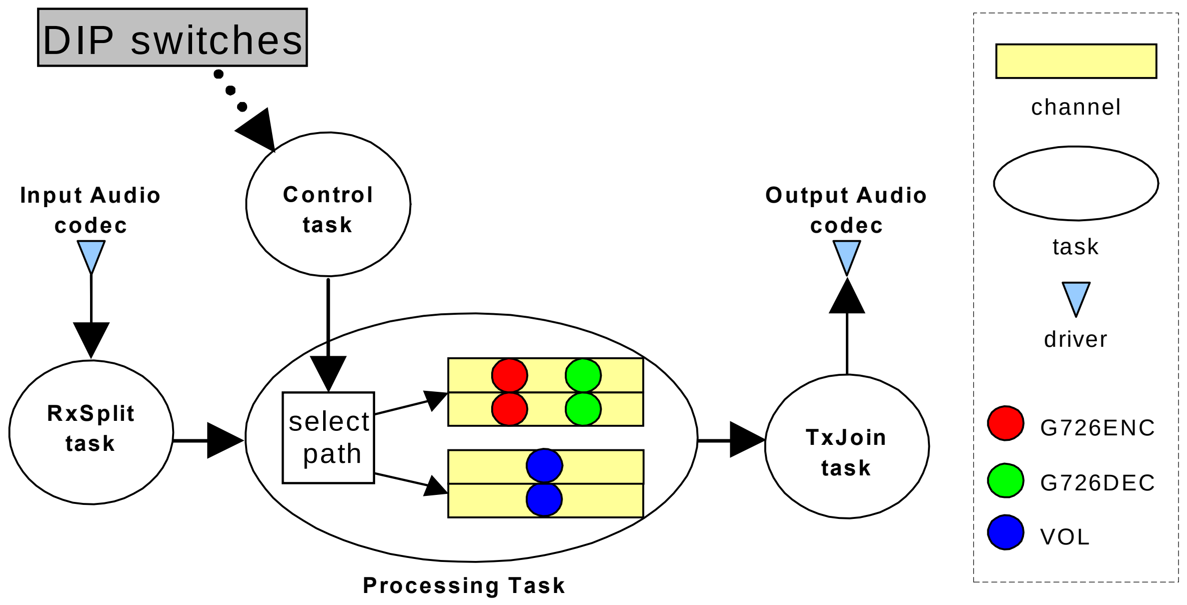

Figure 3. Application Data Path

3.2 Application's State Machine

The application can also be represented as a state machine. This representation helps you determine a good strategy for power management (See Reference 6). Different modes of operation typically have different power requirements, and state transitions are potential times to modify the power management policy of an application.

For example, when the application enters the G.726 running state, which involves heavy processing, you would likely need to set the CPU to run at a high frequency and voltage. On the other hand, when the application enters the sleep state, all peripherals should be shutdown and all clock domains should be gated off.

In our example application, state transitions are initiated either by external events such as toggling of DIP switches or external interrupts or by internal events such as the scheduler entering the DSP/BIOS idle loop. In other applications, such transitions could be triggered with button presses, the low-battery signal from a battery monitor, the ignition of a car, a user putting his finger on a fingerprint imaging sensor, software error detection, and many other conditions.

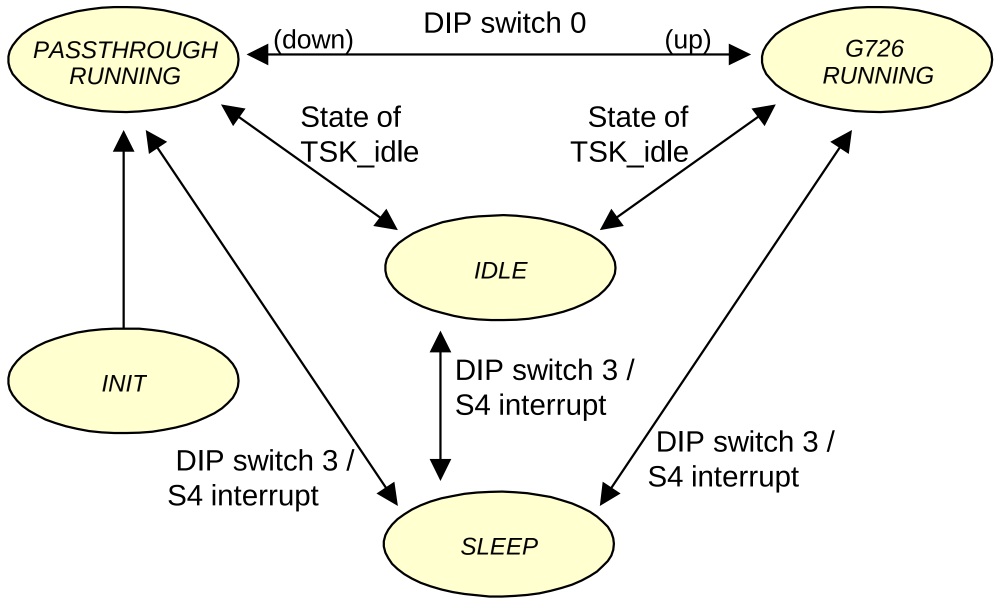

Figure 4. Application State Machine

Figure 4 assumes that all DIP switches are initially in their upward position.

In our discussion of power management techniques in Section 4, you should be able to easily map the techniques to the operation states to which they apply.

3.3 Module Hierarchy

The module hierarchy is almost identical to that in RF5. Additions include the use of the Power Manager PWRM module to implement power management techniques, and the use of the board support library (BSL) to interface with the audio codec. Moreover, the LOAD module is used to dynamically monitor the CPU load. Note that RTDX has been turned off since instrumentation through emulation affects power measurements. Furthermore, use of RTDX and emulation can adversely disrupt the process of putting a DSP into low-power (sleep) mode on some platforms, hence the importance of turning it off cannot be understated.

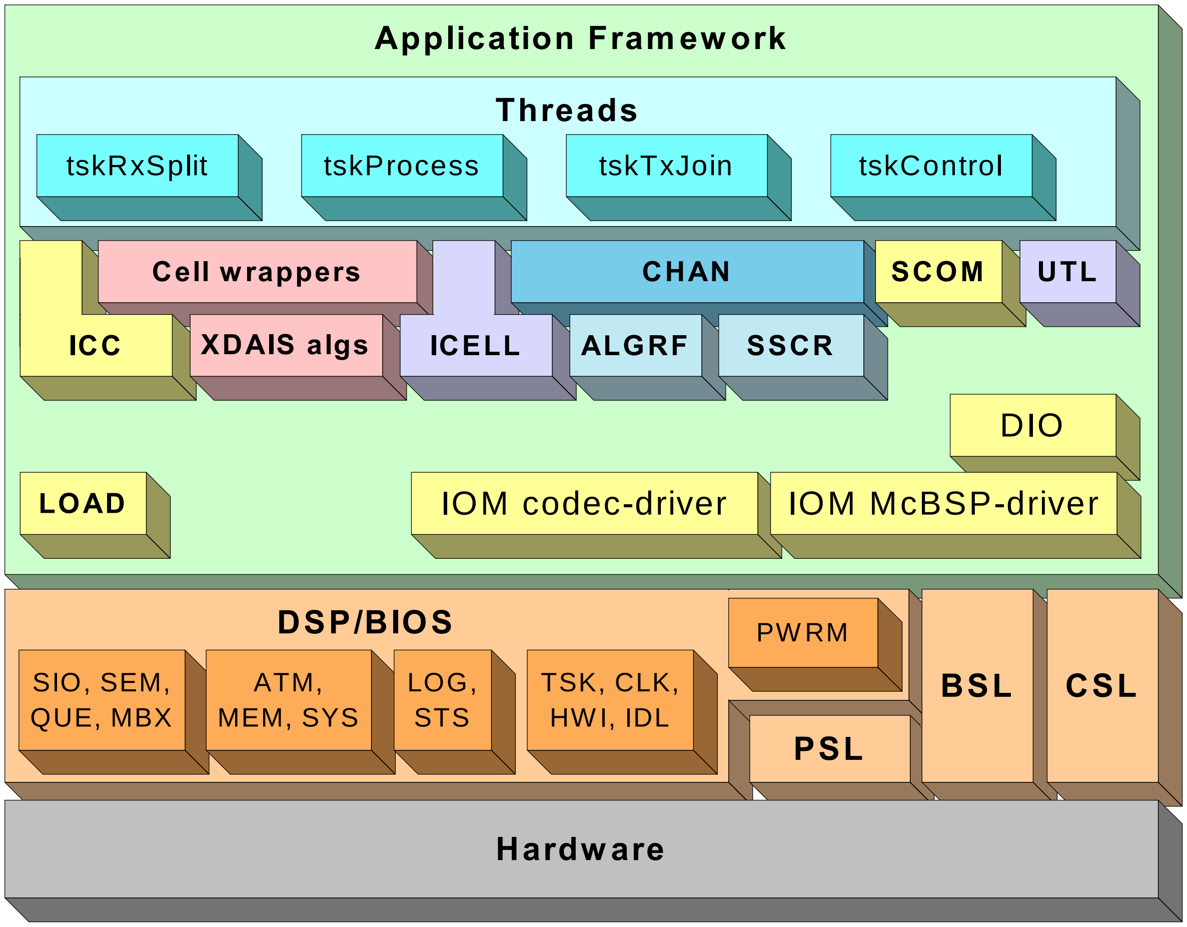

Figure 5. Modules Used in Application

3.3.1 LOAD Module

The LOAD module supplies the capability to compute the CPU load at run-time. Figure 7The following figure illustrates the way it works:

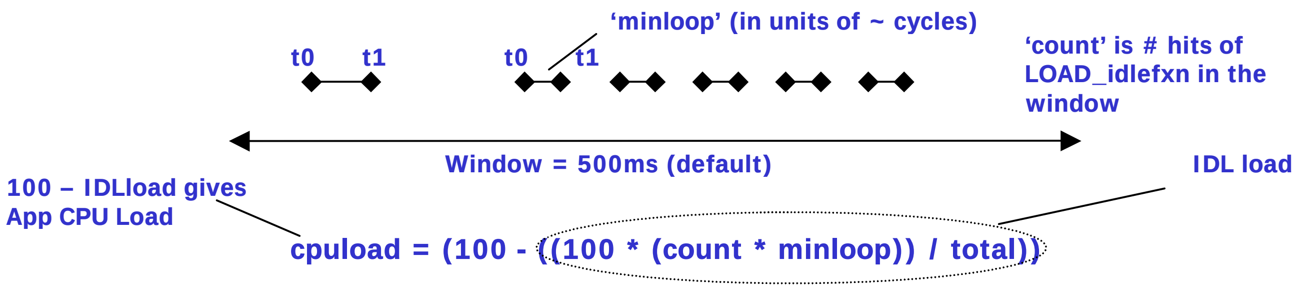
'minloop' (in units of ~ cycles)
t0
t1
t0
t1
'count' is # hits of LOAD_idlefxn in the window
Window = 500ms (default)
IDL load
100 – IDLload gives App CPU Load
cpuload = (100 - ((100 * (count * minloop)) / total))

Figure 6. CPU Load Calculation in LOAD Module

It relies on an IDL thread to be inserted in an application to calibrate the amount of time needed to run one iteration of the DSP/BIOS idle loop. It also keeps track of the number of times the idle loop runs over a time period as specified by the “window” variable in load.c, and estimates the CPU load by dividing the idled time by the time elapsed and subtracting the result from 1. The load is multiplied by 100 and reported as a percentage.

3.4 Directory Structure

Figure 7 shows the directory structure after installing the Reference Frameworks and the audio-streaming application code. It highlights the most important files the directories contain.

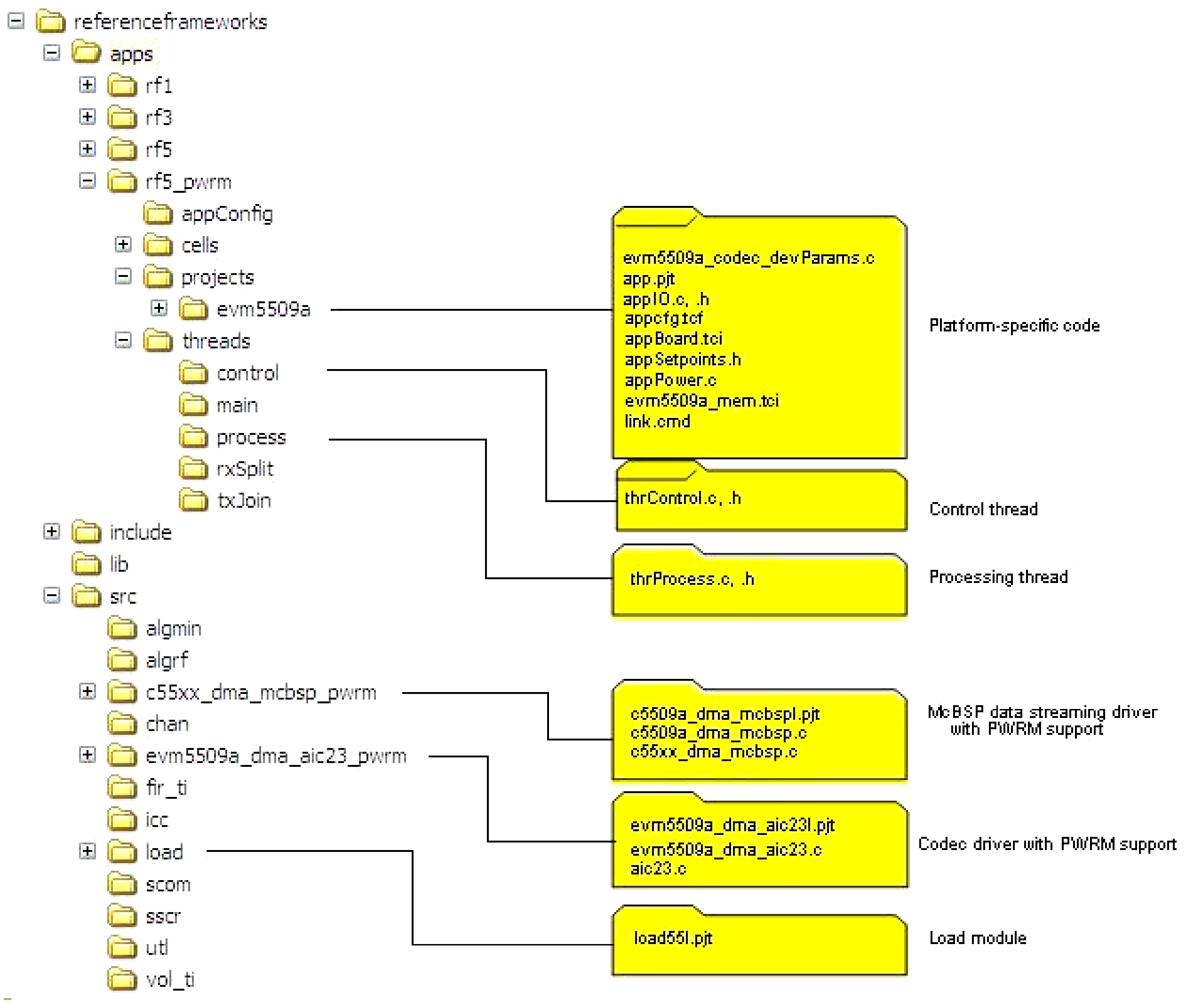

Figure 7. Directory Structure

Folders to notice include:

- **apps\rf5_pwrm.** The root folder for the audio-streaming application.
  - **projects.** Contains platform-specific files for the application. This includes board-specific configuration files, project files and linker command files. In particular, appBoard.tci contains the configuration for the PWRM module. In appSetpoints.h, you can find information pertaining to particular setpoints available on a specific target board. appPower.c is used to hold user functions that are needed to perform power management.
  - **threads.** Contains hardware-independent source files for the threads. The files thrControl.* contain code for the control task that changes power management policy based on dip switch input. The files thrProcess.* contain code that performs data processing during passthrough and G.726 modes.
- **include.** Contains a number of public header files used by the Reference Frameworks. The application uses some, but not all, of these header files. Public header files are referenced by both algorithm and framework code. In contrast, private header files are stored with the source code that includes them and are not intended for use by other modules. Each library module has one header file in this folder.
- **src.** Contains folders with source files for different modules. In particular, it includes files for the LOAD module and the PWRM-based codec and McBSP drivers. Note that in order to rebuild the drivers, you must change the headers' include path in the **Compiler** tab of the **Project→Build Options** dialog by following a procedure similar to the one described in Section 2.4 to change the DSP/BIOS include path to point to your DSP/BIOS install location (default C:\bios_5_20).

4 Power Management Techniques

The example application showcases several power management techniques you can leverage in a DSP/BIOS-enabled application. As with most software techniques, no one size fits all. You should decide which techniques make sense for your application design. To get an idea of the amount of power savings you can typically achieve, see "Appendix A: Power Measurements" for a set of power measurements performed on the provided application in various power-saving configurations.

4.1 Boot-Time Power Reduction Function

The DSP/BIOS PWRM module allows you to configure a hook function that can idle clock domains or resources that are not used before the main() function is run. Typically this can be used to control power management in the initialization state of an application, for example when a consumer switches on a cell-phone or a digital camera. It can also be used to idle clock domains or resources that are not used throughout the application.

In our example, the function is configured for DSP/BIOS in appBoard.tci:

```
tibios.PWRM.BOOTHOOKFXN = prog.extern("bootPowerReduction");
```

The bootPowerReduction function turns off the DSP CLKOUT signal, which is on by default but is never used by the application. Because the C5509a DSP is divided into multiple clock domains, a program can cause the DSP’s clock signal to be fed only to domains (sections) of the chip that are used.

```
/*
 *  ======== bootPowerReduction ========
 *  This function is called at initialization time to reduce power
 *  consumption by the DSP.
 */
Void bootPowerReduction()
{
    Uns idleStatus;
    PWRM_Status status;
    TIMER_Handle hTimer;
    Int i;

    // Call chip support library to turn OFF the DSP's CLKOUT signal
    CHIP_FSET(ST3_55, CLKOFF, 1);

    // Turn off timer 1 whenever Peripherals domain is idle
    hTimer = TIMER_open(TIMER_DEV1, TIMER_OPEN_RESET);
    TIMER_FSETH(hTimer, TCR, IDLEEN, 1);

    // Turn off LEDs
    EVM5509_LED_init();
    for (i = 0; i < NUM_LEDS; i++) {
        EVM5509_LED_off(i);
    }

    // Turn off other resources not needed...
}
```

4.2 Gating Off Clock Domains During DSP/BIOS Idle Loop

DIP switch 1 can be used to gate off the CPU domain while the application runs in the DSP/BIOS idle loop (IDLE state). Gating off clock domains in the idle loop can be thought of as "free" power savings, since interrupt-based applications typically spin in the idle loop while waiting for data or control signals. During this idle time, it is efficient to gate off the CPU clock domain since it is not doing any useful work (otherwise the idle loop would not be running).

The availability of this power-saving mechanism is one more reason to use an interrupt-based software design to maximize the amount of time spent in the idle loop.

```
     case DIP_IDLING:
         if (newDIP == 0) {
            /*
             *  Gate off clock domains during idle loop.
             *  Note that this idle mask is independent of
             *  the one used outside the idle loop.
             */
             attrs.idleMask = PWRM_IDLECPU;
         }
         status = PWRM_configure(attrs);
         UTL_assert (status == PWRM_SOK);
         break;
```

4.3 Voltage and Frequency Scaling

Voltage and frequency scaling (VFS) is another approach to saving power. When an application does not need the full amount of processing power to meet real-time deadlines, we can slow down the CPU by backing off its frequency. At lower frequencies, a lower voltage can be used. Because active power consumption by the CPU is proportional to the clock frequency and to the square of the voltage, as described in Section 1.1, this technique can potentially result in great power savings.

However, a task performed by a CPU running at a lower frequency takes longer to perform. This means potentially less time spent in the IDLE and SLEEP modes. So how does this VFS strategy compare with the strategy of running everything at highest speed, and then putting the DSP in a low power state (for example, by gating off clock domains)? If the idle/sleep mode is ideal and there is no active power drawn during that time, both strategies should result in similar power consumption. However, in practice, there could be active power consumption during sleep/idle due to device limitations or to power-off latencies being too expensive. Hence, by choosing a good voltage and frequency setpoint while an application is in its RUNNING states, you can reduce overall energy consumption, as illustrated in Figure 8:

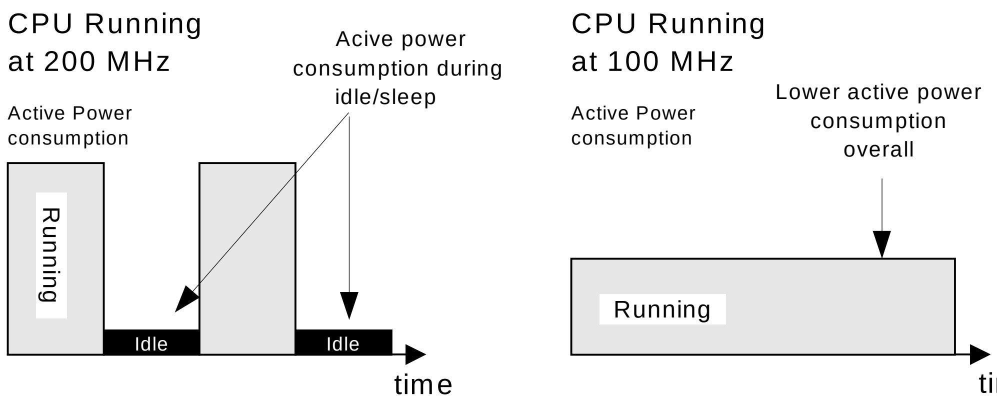

Figure 8. Active Power Consumption Comparison with VFS Using Fixed Voltage

4.3.1 Static Scaling

An easy way to leverage VFS is to determine the best voltage and frequency setpoint to use at design-time and hard-code it statically into the application. Our audio-streaming application has an array in thrControl.c to specify the actual setpoints to be used for each RUNNING state in the system:

```
#define MAXSETPOINT 15             /* max. setpoint supported on this board */

// Modify this array to suit the application
Int staticSetpoints[NUMPROCESSINGMODES] = {
   MAXSETPOINT, // for PASSTHROUGH_MODE
   MAXSETPOINT  // for G726_MODE
};
```

But how can you determine the correct setpoint for each RUNNING state (or processing mode)? One answer is to compute the CPU load theoretically based on the algorithms being run and find a frequency setpoint that should meet the real-time requirements. The other approach is to obtain this setpoint experimentally.

In the example application, putting DIP switch 2 in the "down" position places the application in calibration mode. This mode is not shown in Figure 4 because it is intended only for use in code development; it would not appear in an actual deployed system. Calibration mode causes the tskControl task to check the CPU load periodically and to adjust the VF setpoint downward until the CPU load reaches a target you specify, for example 70%. After that, whenever the CPU load exceeds this value, the frequency is increased by one step. If an application is left running this calibration for a specific processing mode during a long enough period, you can check the staticSetpoints array to find the minimum setpoint appropriate for that processing mode.

Remember that the calibration only estimates the frequency needed to meet real-time deadlines. It does not guarantee meeting real-time requirements, just as running the CPU at full speed at all times does not give that guarantee. If CPU activity occurs in bursts with durations smaller than the CPU load polling period, these bursts will be averaged out over time, and real-time misses could be undetected even with a conservative target CPU load. Similarly, a CPU load of 100% does not necessarily mean real-time was missed. For example, the polling period may be shorter than the processing time of a given frame of data. In a data streaming application, you should calibrate with multiple polling periods that are at least as long as the data frame rate but short enough so that bursts of CPU activity leading to real-time misses are not averaged out.

Another limitation of this calibration method is that it does not work when the CPU clock is gated off in the DSP/BIOS idle loop. That is, DIP switches 1 and 2 cannot be simultaneously in the "down" position. This is because the LOAD module counts the number of times the idle loop runs to compute the CPU load, but with the CPU "stuck" in the idle loop while the CPU clock is gated off, the idle loop is not run as often and the CPU load computation is inaccurate. Nevertheless, after calibrating, you can re-enable the feature that gates off the CPU clock in the idle loop.

4.3.2 Dynamic Scaling

VFS can also be performed adaptively by increasing or decreasing the frequency and voltage of the CPU periodically based on CPU load predictions in an attempt to achieve a target CPU load. However, this is much more difficult to do because it requires that you characterize how the CPU load of an algorithm changes over time. If the VF setpoint is not adjusted rapidly and accurately enough, the application might miss real-time requirements during extreme CPU load variations.

For example, in a voice-coding algorithm such as G.723, there is a component for voice-activity detection. Whenever no voice activity is detected, the algorithm uses a minimal amount of processing power and the CPU load becomes very low. A dynamic VFS strategy might scale the voltage and frequency to low values during this time. When voice is detected in the incoming data, the CPU load jumps suddenly and the scaling strategy, unless it is well designed, will miss real-time requirements.

You can experiment with dynamic VFS using the audio-streaming application. By setting an entry to USE_DVFS in the staticSetpoints array, you tell the application to perform dynamic VFS periodically for a particular RUNNING state/processing mode. In this simple scheme, the prediction algorithm uses the CPU load measured in the previous period to predict the load of the next period, and adjusts the setpoint accordingly.

Using such dynamic VFS for passthrough mode would work fine, since the load rarely changes. However, the power savings, if any, is small compared to static VFS. In fact, it is possible that scaling dynamically performs worse than the static case, as the predictive scheme might cause the setpoint to fluctuate to a value higher than the optimal one.

On the other hand, with its faster changes in CPU requirements, G.726 mode might potentially miss real-time deadlines with dynamic VFS, since the example adjusts the setpoint by single steps (although we did not experience missed deadlines during our testing). Because the smallest relative frequency adjustment on the EVM 5509A occurs during the transition from 192 MHz to 180 MHz, this scheme is valid only for cases where the CPU load changes by no more than 6.25% in a period of six seconds (twice the polling period) within a given processing mode.

```
if (staticSetpoints[thrControl.mode] == USE_DVFS) {
    /*
     *  Do DVFS. Scale only if cpu load is outside range of
     *  TARGET_CPU_LOAD +- LOAD_MARGIN
     */
    if (systemCpuLoad <= TARGET_CPU_LOAD - LOAD_MARGIN){
        // Decrement frequency and/or voltage
        if (setpoint > minSetpoints[thrControl.mode]) {
            setpoint--;
            status = PWRM_changeSetpoint(setpoint, NOTIF_TIMEOUT);
            UTL_assert(status == PWRM_SOK);
            UTL_logDebug1("New setpoint is %d", setpoint);
        }
    }
    else if (systemCpuLoad > TARGET_CPU_LOAD + LOAD_MARGIN) {
        // Increment frequency and/or voltage
        if (setpoint < MAXSETPOINT) {
            setpoint++;
            status = PWRM_changeSetpoint(setpoint, NOTIF_TIMEOUT);
            UTL_assert(status == PWRM_SOK);
            UTL_logDebug1("New setpoint is %d", setpoint);
        }
    }
}
```

In brief, dynamic VFS is of limited use without intimate knowledge of CPU load usage. The ability to predict CPU load correctly is essential. Power-users might want to explore this methodology further to obtain additional power savings, but in general it is a sufficient strategy to design an application using static VFS.

Currently, dynamic VFS does not work in the supplied application when the CPU clock is gated off in the DSP/BIOS idle loop. That is, DIP switches 1 and 2 cannot be simultaneously in the "down" position. This is because the LOAD module counts the number of times the idle loop runs to compute the CPU load, but with the CPU "stuck" in the idle loop while the CPU clock is gated off, the idle loop is not run as often and the CPU load computation is inaccurate. The impact of this limitation is minimized when using dynamic VFS, since with a good CPU load prediction algorithm, the amount of time spent in the idle loop should decrease at reduced CPU frequencies. Hence, reduction in the time spent in the idle loop lessens the power-savings contribution by the IDLE state.

4.3.3 Latency Issues in Voltage and Frequency Scaling

One drawback of voltage and frequency scaling is that there is latency involved when changing the voltage and frequency. Figure 9 shows the latencies on the 5509a EVM as an example:

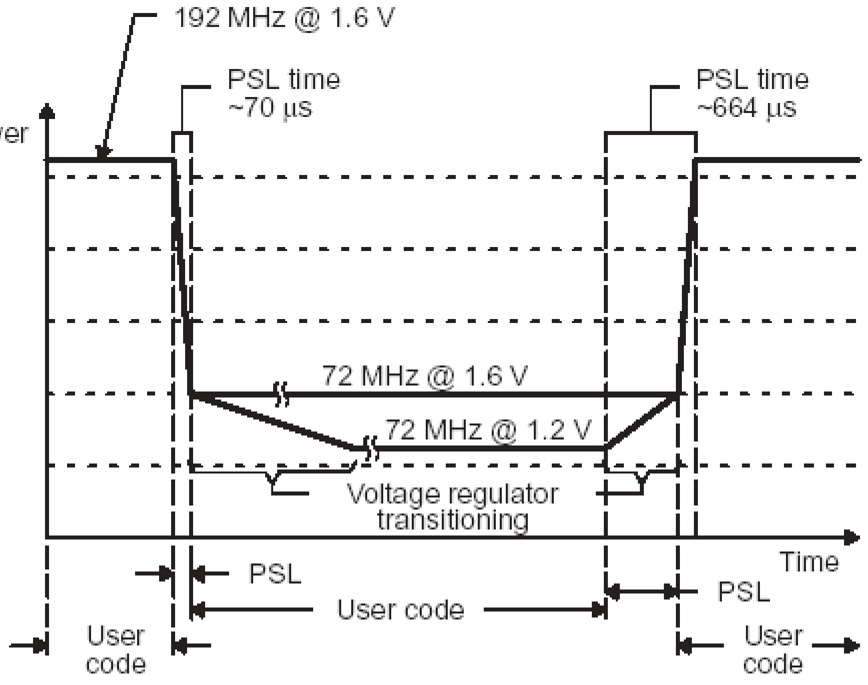

Figure 9. Latencies When Using VF Scaling

The latency varies widely across platforms. An application might run fine on one platform with certain VF setpoints, but may not run properly (for example, may miss deadlines) on a revision of the platform due to circuit component changes that alter the voltage scaling latencies. Consequently, CPU load estimation and/or calibration needs to be redone when the hardware design changes.

The feasibility of dynamic VFS is highly dependent on latencies, even more so than the static case, since voltage and frequency transitions would occur much more frequently.

4.4 CPU Sleep and Wake-Up

Placing DIP switch 3 in the "down" position causes the application to go into deep sleep (SLEEP state). The concept of deep sleep varies for different projects, but in general, it is thought of as the state in which an application does minimal to no work at all and the CPU and most of its peripherals can be put into a low-power mode. The SLEEP state differs from the IDLE state in the sense that it powers things down more extensively, resulting in a higher latency during wake-up time (See Appendix B for sleep mode latencies). For instance, it is the state of a cell-phone that has not been used for an extended period, or a car's MP3 player that's waiting for a passenger to open the door to start playing.

In the example application, the codec and the serial port data streaming drivers have been modified to register with the PWRM power manager for notification of sleep and awakening events. Figure 10 summarizes what the PWRM power manager and the drivers' notify functions do when these event occur.

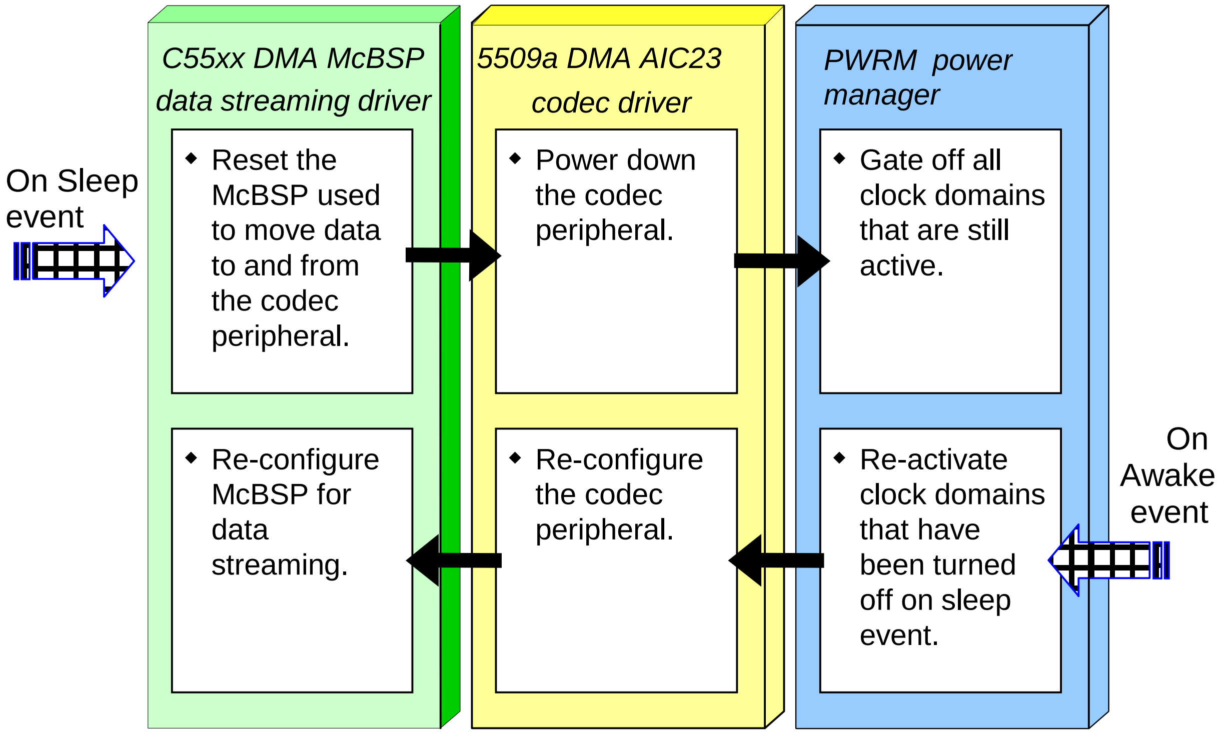
C55xx DMA McBSP data streaming driver
5509a DMA AIC23 codec driver
PWRM power manager
On Sleep event
Reset the McBSP used to move data to and from the codec peripheral.
Power down the codec peripheral.
Gate off all clock domains that are still active.
Re-configure McBSP for data streaming.
Re-configure the codec peripheral.
Re-activate clock domains that have been turned off on sleep event.
On Awake event

Figure 10. Sleep and Wake-Up Action Sequence

These actions are performed in the order shown so that the McBSP serial port does not shuttle data to or from a powered-down codec.

An interrupt is needed in order to pull the application out of deep sleep. Because we have gated off all clock domains to maximize power savings, this interrupt has to be initiated externally. For this application, we are using “Wake Up” Switch (S4). Before going to sleep, the task that initiates the sleep event should try to back off the voltage of the CPU as much as possible while maximizing frequency. This allows the CPU to use the minimum voltage during sleep—hence reducing static power consumption as per Section 1.1—while executing the sleep and wake-up code at the highest possible speed. We also need to turn off the scheduler during this going-to-sleep and wake-up process so that we do not get preempted after voltage and frequency have been scaled down.

```
#define MAXFREQ_MINVOLT_SETPOINT 8

case DIP_SLEEP:
    if (newDIP == 0) {
        /*
         *  Turn off scheduler so that we do not get preempted,
         *  which results in subsequent processing to be done
         *  at a frequency that is too low.
         */
        SWI_disable();
        // Go to sleep after backing off voltage
        status = PWRM_changeSetpoint(MAXFREQ_MINVOLT_SETPOINT,
            NOTIF_TIMEOUT);
        UTL_assert(status == PWRM_SOK);
        status = PWRM_sleepDSP(PWRM_DEEPSLEEP, 0,
            NOTIF_TIMEOUT);
        UTL_assert(status == PWRM_SOK);
        status = PWRM_changeSetpoint(
            staticSetpoints[thrControl.mode], NOTIF_TIMEOUT);
        UTL_assert(status == PWRM_SOK);
        SWI_enable();
        UTL_logDebug("I am awake");
    }
    break;
```

5 System-Level Considerations and Other General Techniques

There are other considerations in power management beyond using the PWRM module to obtain power savings. Since the DSP is never the only power consumer in a real-world system, it is important to expand our scope to see what other savings we can obtain from a DSP software perspective.

5.1 Use Memory Closest to the CPU

In a hierarchical memory model, one way to save power is to place code and data in the memory closest to the CPU. Typically, by using internal memory, cache, instruction buffers, and the like, you can produce an application that consumes less power. External memory access should be avoided not just to improve performance, but also to save power. For example, in the audio-streaming application, all code and data are in internal memory (DARAM and SARAM memory sections in the DSP/BIOS configuration), including what is conceptually the external memory heap. As a result, XDAIS algorithms that request external memory for data buffers are actually given internal memory. Moreover, you might want to use memory overlays if possible to reduce the need for external memory usage.

5.2 Optimize Code Size and Performance

Another tip is to optimize your application's code size and performance. A smaller application typically has less need to put code in external memory. Optimizing for performance also reduces power consumption if it results in fewer non-idle instructions being run over time. However, if performance improvements are achieved via parallelism through the use of more functional units on a DSP, power consumption may actually increase if more resources must to be powered up for the processing to be performed.

5.3 Know Your Peripherals

Modern DSP processors often control a number of on-chip and on-board peripherals. It is important to find out if low-power modes exist for these peripherals. Sometimes it is enough just to know that they exist and can be turned off! For example, we turned off the DSP CLKOUT signal via the boot hook function in Section 4.1. The knowledge that this signal exists and is not being used to drive any external peripheral allowed us to turn it off and obtain power savings. Here are examples of additional things you might do to reduce peripheral power consumption:

- Make sure the idle enable bits in control registers for unused peripherals, such as timers and McBSPs, are on to allow them to go idle during IDLE and/or SLEEP states.
- Micro-manage the on/off status of the backlight for an LCD display.
- Use an external clock if one is available to drive serial ports instead of the DSP clock. This allows you to have the DSP gate off the PERIPHS and CLKGEN domains.
- Turn off unnecessary power consumers during boot-up time.
- Use self-refresh modes for dynamic memories and displays where the device will manage the refresh operation on its own.

6 Portability

The RF5-based application supplied is designed to be easily portable to platforms other than the 5509A EVM that are/will be supported by the PWRM module. Porting involves performing the following modifications:

1. Replace drivers with the ones for your target platform. This includes modifying the drivers to register notification functions with the PWRM module for sleep and wakeup events.
2. Update appSetpoints.h to use valid setpoints for your target platform. This includes modifying the value of constants such as MAXFREQ_MINVOLT_SETPOINT and MAXSETPOINT.
3. Update the bootPowerReduction function if necessary, since some of the peripherals such as Timer 1 and CLKOUT signal might not be applicable on your target platform.
4. Obtain libraries targeted for your DSP architecture. For example, the supplied G.726 algorithms work only on the C55x DSPs. Hence, you will need to use a different version if you use a different family of DSPs. This includes having the correct version of PSL that works with your target device.
5. Update the DSP/BIOS configuration to match the configuration of this new platform.

7 Power Management Quick Reference

Table 2 lists some questions you should ask when designing the power management of your application.

Table 2. Questions for Power Management Design

Question	If the Answer is Yes:
Does your application have a user-induced stand-by mode in which the DSP is not required to perform work and simply waits for an interrupt event?	Use PWRM to lower the voltage and gate off clock domains during deep sleep. Shut down all unused peripherals during this mode.
Does your DSP code occasionally go idle while your application is active? Are there unused cycles during data processing? Do you want a quick way to automatically save power during this idle time?	Use PWRM to gate off unused clock domains during the idle mode.
Are there unused domains or peripherals during boot-up time?	Gate or power them off using the boot hook function in the PWRM module specified under DSP/BIOS configuration.
Does your DSP code ever go idle while your application is running? Do you need to maximize the amount of power you save during this idle time?	Consider using the PWRM module to lower the frequency and voltage of the CPU. You will need to determine the minimum frequency at which you can run each processing mode in the application without missing real-time.
Are power-savings on the DSP itself not enough to meet power requirements?	Look elsewhere in the system. Put code in memory closest to the CPU. Optimize your code for both performance and footprint. Investigate peripherals and clock signals that can be shut down or put into power-saving modes. You may want to leverage PWRM's registration/notification mechanism to automatically notify drivers of power events (e.g. deep sleep), so they can manage the power consumption of the peripherals they drive.
Have you tried everything and yet still need to save more power?	Characterize the amount of power needed by your application over time. Use this characterization to predict your CPU load over time. Dynamically lower the frequency as much as possible whenever the application does not use the CPU fully.

For power consumption by the DSP and its on-chip peripherals, Texas Instruments has provided Microsoft Excel spreadsheets for most DSPs to allow users to estimate the theoretical power consumption for their DSP, based on %CPU utilization, peripheral idle status, temperature, CPU frequency, and other variables. For example, see Reference 7 for the spreadsheet of the C5509A.

8 Conclusion

In this application note we discussed the use of OS abstractions in DSP/BIOS to implement different power management policies. The PWRM module provides facilities that specify power management policies in various operation states, as demonstrated in the audio-streaming application provided.

Start by identifying the operation states in your application. Then choose methodologies to use with your application depending on the semantics of the possible states and the underlying hardware.

It is important to realize that there is no one strategy that works for every application. When producing DSP software, keep system-level considerations in mind to lower the power consumption for your complete design. Think about power early. This does not necessarily mean you need to turn on power management in your initial software development. In fact, it is suggested that you add power management after the code works. Early on, you should investigate how you can save power, identify the states in your application, experiment with hardware, and put in the hooks needed to ease the change in power management policies during state transitions. Then enjoy the power savings.

9 References

1. "Power-Optimizing Embedded Applications", *Proceedings of the Embedded Systems Conference*, Scott Gary, March 2004.
2. *TMS320C5000 DSP/BIOS Application Programming Interface (API) Reference Guide*, SPRA404J, Texas Instruments.
3. *TMS320 DSP/BIOS User's Guide*, SPRU423F, Texas Instruments.
4. *Using the Power Scaling Library,* SPRA848A, Texas Instruments
5. *Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High-Density System*, SPRA795A, Texas Instruments.
6. *System-Level Power Optimization: Techniques and Tools*, ISLPED99, Luca Benini, Giovanni DeMicheli, 1999.
7. *TMS320VC5509A Power Consumption Summary*, SPRAA04A

Appendix A: Power Measurements

Power measurements were performed with the application using the setup described in Section 2.3. All numbers in this appendix were obtained under the following conditions:

• Platform:	EVM 5509A PLUS Rev A
• Sampling rate:	8 kHz
• Samples per frame:	80
• Debug flags:	-g
• Optimization:	none in the application (-o2 in algorithms)
• DSP/BIOS:	Version 5.20
• RTDX:	Off
• Emulator:	Specrum Digital XDS510PP+ connected
• Run mode:	Run Free and Disconnect in CCS 3.1
• Built-in shunt resistors:	DSP core (JP3) 0.1 ohm, supply (JP2) 0.025 ohm
• CPU load at 192 MHz:	3% for passthrough, 50% for G.726
• Multimeters:	Two FLUKE 187 True RMS Multimeters for voltage drops across shunt resistors and for absolute DSP and supply voltages

Table 3. Power Measurements in Passthrough Mode

	Setup	Supply current (mA)	Supply voltage (V)	Board power (mW)	DSP Core current (mA)	DSP Core voltage (V)	DSP Core power (mW)	DSP Power Reduction Relative to Case 1 (%)	Total Power Reduction Relative to Case 1 (%)
1	CPU @ 192 MHz, passthrough mode, **all code off-chip,** Boot power reduction fxn: off, idle loop: domains active	399.8	5.11	2043	123.8	1.646	203.8	0	0
2	CPU @ 192 MHz, passthrough mode, **all code on-chip,** Boot power reduction fxn: off, idle loop: domains active	348.1	5.11	1779	123.1	1.646	202.6	0.57	12.9
3	CPU @ 192 MHz, passthrough mode, all code on-chip, **Boot power reduction fxn: on,** idle loop: domains active	330.4	5.12	1692	121.9	1.647	200.8	1.47	17.2

	Setup	Supply current (mA)	Supply voltage (V)	Board power (mW)	DSP Core current (mA)	DSP Core voltage (V)	DSP Core power (mW)	DSP Power Reduction Relative to Case 1 (%)	Total Power Reduction Relative to Case 1 (%)
4	CPU @ 192 MHz, passthrough mode, all code on-chip, Boot power reduction fxn: on, **idle loop: domains idled**	314.0	5.12	1608	65.21	1.652	107.7	47.1	21.3
5	**CPU @ 12 MHz,** passthrough mode, all code on-chip, Boot power reduction fxn: on, **idle loop: domains active**	281.6	5.15	1450	6.58	1.247	8.21	96.0	29.0
6	**CPU @ 12 MHz,** passthrough mode, all code on-chip, Boot power reduction fxn: on, **idle loop: domains idled**	281.7	5.04	1420	4.85	1.247	6.05	97.0	30.5
7	**Using dynamic VFS,** passthrough mode, all code on-chip, Boot power reduction fxn: on, idle loop: domains active	278.9	5.05	1408	6.51	1.247	8.12	96.0	31.1

Observations:

- About 13% board power reduction was observed simply by moving the .text code section from external to internal memory.
- Idling clock domains in the DSP/BIOS idle loop resulted in an extra 45% improvement in DSP power reduction in the passthrough mode.
- Setpoints used for frequency and voltage scaling in cases 5 and 6 were optimized by using the calibration methodology described in Section 4.3.1.
- Static frequency and voltage scaling combined with idling in the DSP/BIOS idle loop gives best power savings during run time (shown in bold in Table 3)
- Dynamic voltage and frequency scaling did not produce significant power savings over the static case in passthrough mode, since the CPU load varies little. Ultimately the application runs at the same setpoint as in the static case.

Table 4. Power Measurements in G.726 Mode

	Setup	Supply current (mA)	Supply voltage (V)	Board power (mW)	DSP Core current (mA)	DSP Core voltage (V)	DSP Core power (mW)	DSP Power Reduction Relative to Case 1 (%)	Total Power Reduction Relative to Case 1 (%)
1*	CPU @ 192 MHz, G.726 mode, **all code off-chip,** Boot power reduction fxn: off, idle loop: domains active	431.7	5.14	2219	126.3	1.645	207.8	0	0
2	CPU @ 192 MHz, G.726 mode, **all code on-chip,** Boot power reduction fxn: off, idle loop: domains active	349.4	5.12	1789	123.5	1.645	203.2	2.22	19.4
3	CPU @ 192 MHz, G.726 mode, all code on-chip, **Boot power reduction fxn: on,** idle loop: domains active	332.2	5.10	1694	122.1	1.647	201.1	3.21	23.6
4	CPU @ 192 MHz, G.726 mode, all code on-chip, Boot power reduction fxn: on, **idle loop: domains idled**	325.4	5.11	1663	94.1	1.649	155.2	25.3	25.1
5	**CPU @ 144 MHz**, G.726 mode, all code on-chip, Boot power reduction fxn: on, **idle loop: domains active**	314.1	5.13	1611	79.8	1.452	115.9	44.2	27.4
6	**CPU @ 144 MHz**, G.726 mode, all code on-chip, Boot power reduction fxn: on, **idle loop: domains idled**	312.2	5.14	1605	68.5	1.452	99.5	52.1	27.7
7	**Using dynamic VFS,** all code on-chip, Boot power reduction fxn: on, idle loop: domains active	312.7	5.12	1601	80.1	1.452	116.3	44.0	27.9

*Code missed real-time

Observations:

- Power reduction is more modest than in the passthrough case, since the CPU load is higher.
- More than 19% power reduction was observed simply by moving the .text code section from external to internal memory.

- Idling clock domains in the DSP/BIOS idle loop still resulted in an extra 22% improvement in DSP power reduction in G.726 mode, even though the CPU load is much higher than in the passthrough case.
- Setpoints used for frequency and voltage scaling in cases 5 and 6 were optimized by using the calibration methodology described in Section 4.3.1.
- Static frequency scaling combined with idling in the DSP/BIOS idle loop again gives the best power savings during run time
- Dynamic voltage and frequency scaling did not produce significant power savings over the static case in passthrough mode, since the CPU load varies little. Ultimately the application runs at the same setpoint as in the static case.

Table 5. Comparison Between Idle and Sleep Modes

	Setup	**Supply current (mA)**	**Supply voltage (V)**	**Board power (mW)**	**DSP Core current (mA)**	**DSP Core voltage (V)**	**DSP Core power (mW)**	**DSP Power Reduction Relative to Case 1 in passthrough mode (%)**	**Total Power Reduction Relative to Case 1 in passthrough mode (%)**
1	**No processing**, all code on-chip, Boot power reduction fxn: on **idle loop: CPU domain idled**	310.28	5.13	1592	56.3	1.651	92.95	54.4	22.1
2	**Sleep mode**, codec powered down, with voltage reduction during sleep, all code on-chip, Boot power reduction fxn: on	266.12	5.08	1352	0.29	1.244	0.361	99.8	33.8

Observations:

- Sleep mode produces the highest power savings since there is no audio and the system is not doing any useful work. By operating at the lowest voltage available, resetting the McBSP, powering down the codec, and gating off all clock domains, we were able to dramatically reduce the DSP power consumption to 0.361 mW and the board power consumption by more than 33% when compared with case 1 in Table 3.
- When compared with case 1, in which we simply run no processing in which the main function is empty and the application tasks are suspend leaving the code to spin in the idle loop with CPU and CACHE domains idled, sleep mode saves about 11% more in board power consumption and more than 45% in DSP power consumption, at the cost of high going-to-sleep and awakening latencies (see Appendix B).

Appendix B: Sleep Mode Latencies

Sleep mode latencies were measured using the Profiler Clock in Code Composer Studio and tabulated in Table 6. CPU frequency is fixed at 192 MHz. Interrupts are disabled while running the notification functions.

Table 6. Sleep Mode Latencies in Number of Clock Cycles

	On Sleep Event	On Awake Event
McBSP data streaming driver notification function	1983 cycles	1245 cycles
Codec driver notification function	24880 cycles	34142 cycles
Overall latency	26967 cycles	35493 cycles

Observations:

- It takes longer to wake up the DSP than for it to go to sleep. Much of the latency comes from re-configuring the hardware audio codec upon wakeup.
- Latency can be reduced by selectively doing less. For example, if we leave the codec powered up during sleep mode, the latency of the codec driver notification function on the awake event can be improved drastically. This is a tradeoff between power consumption and latency.