# Using the Power Scaling Library


## ABSTRACT

Power consumption is a key concern for embedded system developers. By developing low-power solutions, developers can deliver products that have longer battery life. One technique that can be used to save power is frequency and voltage scaling of the processor.

Since the power consumption of a DSP is proportional to the system clock switching speed, running the device at the lowest possible frequency, while continuing to meet all of the application's timing requirements, can save power by minimizing the idle time. In addition, batteries have a non-linear discharge pattern, where higher currents drain the battery quicker. Thus, executing at lower frequencies can extend battery life. Since lower frequencies require less voltage, an even greater decrease in power consumption can be achieved if the voltage is also lowered when the frequency is lowered.

This application report describes the power scaling library (PSL) and how to use it in your own C55x applications.

Do note that this tooling currently supports the C5503, C5507, C5509A, and C5510 Rev 2.x silicon only.


## Contents

**List of Figures**



**List of Tables**



# 1 Overview

The power scaling library (PSL) is a software library that allows embedded systems programmers to manage both frequency and voltage scaling through an easy to use API. The PSL provides hardware abstraction, portability, and a standard API that can be used among different TI devices. The result is shortened development time for the end user.

Included in the API are routines that initiate scaling operations, and various query routines that provide information on current settings and available frequency/voltage settings. Frequency changes are initiated directly by the user. Voltage changes are performed indirectly by the PSL when a frequency change occurs. Specifically, the PSL will automatically scale the voltage to the minimum level required by a frequency. Since voltage changes are only initiated indirectly, the PSL can ensure a legal frequency/voltage setting at all times.

This document illustrates the frequency/voltage scaling features.

The PSL is delivered as libraries for large model only:

- PSL _<chip>.a55L
- PSL_debug_<chip>.a55L
- PSL_cfg_<chip>.a55L
- PSL_cfg_<board>.a55L

**PSL_<chip>.a55L** contains the actual implementation of the power scaling library. A different PSL_<chip>.a55L is required for each device/voltage regulator control scheme combination. This is necessary because different devices across an instruction set architecture (ISA) may have different clock generators, and different regulators may have different methods of control. Although PSL_<chip>.a55L is device/regulator and control/scheme-specific, a separate library containing user configurable data will enable it to be used with custom target boards.

**PSL_debug_<chip>.a55L** contains the same implementation of the power scaling library with the exception that error returns are checked and user input is validated. This library has a slightly larger footprint and slightly slower execution speed, but does provide user input validation. This library should be used in place of PSL_<chip>.a55L.

**PSL_cfg_<chip or board>.a55L** provides system and board-specific data to the PSL. The source files that were used to build it are also delivered to users, who can then modify the data in these files and rebuild the library so that the PSL can be used with custom boards. Alternatively, users can modify these files and include them directly in their applications.

## 1.1 PSL Functionality

The PSL provides the following functionality:

- Scaling operation to scale frequency and voltage or frequency only
- Query operations that return current frequency and voltage settings
- Query operations that return available frequencies settings and the required voltage settings for those frequencies
- Query operation that returns the latencies associated with a scaling operation
- Callbacks to user code before and after scaling operations. These callbacks will enable users to perform any necessary peripheral modifications that may be required as a result of the upcoming/just completed scaling operation.

## 1.2 PSL Configuration Data

The PSL provides the following configuration data:

- Input frequency
- Maximum frequency
- Table of supported frequencies (including clock mode)
- Frequency/voltage table
  **Note:** This table is subject to change.
- Latencies associated with frequency and voltage scaling
- Information relating to voltage regulator control, such as address of control register, or the exact GPIO pins that are used to control the regulator. This will add some flexibility to the default voltage regulator implementation that is supplied with the PSL.
- Hook functions to override the default voltage regulator control implementation. This will allow the PSL to be used in systems that require a voltage regulator control implementation that is different than that described in the reference design.

## 1.3 User Characteristics

The expected user of the PSL has the following characteristics:

- Understands the timing and scheduling issues of the system, knows when frequency can be decreased, and when it must be increased
- Understands the effects that frequency scaling may have on the peripherals, and how to reprogram the peripherals accordingly

## 1.4 Constraints

The PSL has the following constraints:

- Voltage scaling cannot be done unless a voltage regulator is present. Note that frequency scaling can still be done without voltage scaling.
- Must have code generation tools to rebuild configuration data
- The PSL cannot be used on the simulator

The PSL does not control any of the peripheral clocks. It only controls the main CPU clock(s).

## 1.5 Power Savings Example Using the PSL

Figure 1 shows how the power consumption can be affected by frequency and voltage scaling. The graph below shows the power savings on the evm5509a that were obtained when scaling only the frequency, and the power savings that were obtained when scaling both the frequency and the voltage. In this example, the power savings achieved by lowering the frequency from 192 MHz to 72 MHz was 62 percent. Additional savings were realized when the voltage was also lowered from 1.6 v to 1.2 v. In this case, the overall power savings were 77 percent.

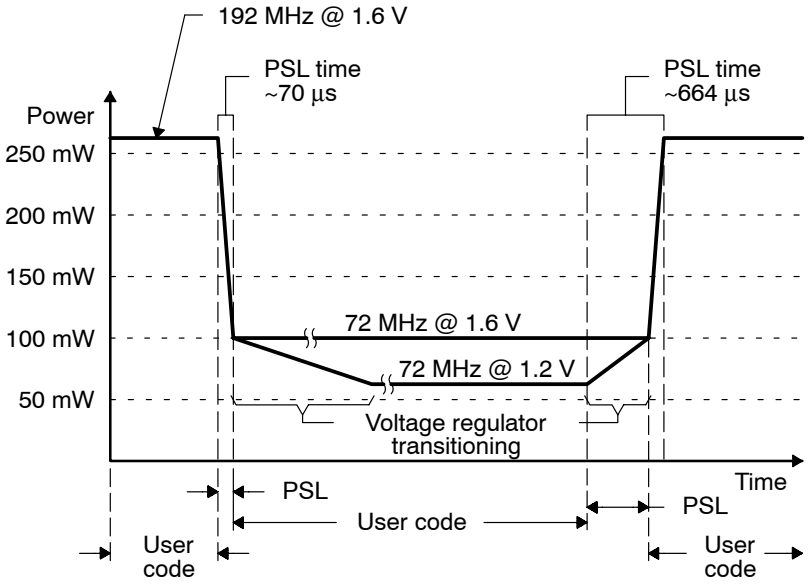


**Figure 1. Effects of Scaling on Power Consumption**

The graph also shows the execution flow of a scaling operation and the amount of time spent in the PSL. In the case where both the frequency and voltage are scaled, the user code calls into the PSL to lower the frequency from 192 MHz to 72 MHz. The PSL lowers the frequency to 72 MHz and *automatically* lowers the voltage to 1.2V, which is the lowest voltage required for 72 MHz. We can see that the PSL waits for the new frequency to be reached, but not the voltage. The time spent in the PSL in this case was ~70 μs, which is the amount of time it takes the PLL to lock to the new frequency.

Looking at the second call to the PSL, the user code calls into the PSL to scale the frequency from 72 MHz to 192 MHz. First, the PSL will *automatically* increase the voltage to 1.6V, and then increases the frequency to 192 MHz. The PSL does not return until both the voltage and frequency have been increased. In this case, the PSL must wait for the voltage increase to complete because it must be increased before the frequency is increased. The time spent in the PSL was ~664 microseconds (~500 for the voltage increase, and ~164 for the PLL to lock to the new frequency).

# 2 Getting Started

## 2.1 Using the PSL

The configuration library that is delivered for specific chips and boards may be used without modification. In this case, follow the steps below to use the PSL in your application:

1. Include the header file PSL.h in all files that will reference the PSL API.
2. Modify your source code to initialize the PSL library by adding a call to the PSL_initialize function. This function should be called before any other PSL function.
   **Note:** The initial clock frequency on the evm5509a is 12 MHz. The initial voltage following power-up is 1.2v. These values should be specified in the call to `PSL_initialize`. See the description of the `PSL_initialize` function for more details.
3. Modify your source code to call other PSL functions as needed.
4. For applications built with the large memory model, include PSL _<chip>.a55L and PSL_cfg_<chip or board>.a55L.
5. Add the directory containing the PSL header files to your “include” search path so that the required header files can be located.
6. Rebuild your application.

Although no changes are required to use the delivered configuration libraries you may want to change the set of default frequencies that the library supports. To modify the frequencies, the frequency table that is located in the file PSLclk_cfg_<chip or board>.c will need to be modified. See sections 4 and 5 for more details.

## 2.2 Using the PSL With Custom Boards

Before using the PSL on a custom board, the configuration data in the files PSLclk_cfg_<chip or board> .c and PSLvolt_cfg_<board> .c must be reviewed. If the values specified in these files are not correct for your board, then you must modify these files prior to using them in your application. See sections 4 and 5 for more details. In this case, follow the steps below to use the PSL in your application:

1. Review the data in the configuration files PSLclk_cfg_<chip or board>.c and PSLvolt_cfg_<board>.c.
2. Change the data in these files to the values that are appropriate for your system.
3. Either rebuild the configuration library or include the configuration files directly in your application build.
4. Include the header file PSL.h in all files that will reference the PSL API.
5. Modify your source code to initialize the PSL library. Adding a call to the PSL_initialize function does this. This function should be called before any other PSL function.
6. Modify your source code to call other PSL functions as needed.
7. For applications built for the large memory model, include PSL_<chip>.a55L and PSL_cfg_<board>.a55L. Note: if you are including the modified configuration files directly in your application, then there is no need to include the configuration library.
8. Add the directory containing the PSL header files to your include search path so that the required header files can be located.
9. Rebuild your application.

# 3 Frequency Scaling in DSP/BIOS Applications

The frequency scaling operations in DSP/BIOS applications as described below are for demonstration and evaluation purposes only (at this time). Future DSP/BIOS and PSL revisions may eliminate the restrictions noted.

The frequency scaling operations performed by the PSL will directly impact the timekeeping and scheduling operations within a typical DSP/BIOS application. Specifically, the following will be affected:

1. Low and high resolution CLK APIs (CLK_gethtime and CLK_getltime) will return invalid values.
2. User clock functions that have been configured to run on each clock interrupt may run faster or slower than expected.
3. Periodic functions scheduled to run periodically based on system clock ticks may run faster or slower than expected.
4. Timeouts for blocking API calls (e.g., SEM_pend, etc.) may return early or late.
5. The CPU load graph and other RTA displays will be in error.
6. Any drivers managing peripheral devices driven by the scaled clock will need to be updated to reprogram the peripheral registers around a frequency scaling operation. The application will need to coordinate this with the individual drivers.

If you are building a BIOS application and making direct calls to the PSL, turn off the BIOS Clock Manager to eliminate the above uncertainties. To do this,

- Right click on the PRD – Periodic Function Manager, and select Properties.
- Uncheck the box, Use CLK Manager to drive PRD. Click OK.
- Right click on the CLK – Clock Manager Properties.
- Select Properties, and uncheck the box, Enable CLK Manager.
- Click OK.

Turning off the clock manager will result in the following:

1. CLK APIs will return (non–incrementing) “stuck” time values.
2. User clock functions will not run.
3. Periodic functions will not run.
4. Blocking API calls will never timeout.
5. RTA displays will be in error.

The application must be modified to accommodate these limitations.

If an external clock is available that is not affected by the PSL frequency scaling operation, this signal could be configured to provide the system ticks (see section 4.8 , Timers, Interrupts, and the System Clock, in SPRU423). Using an external clock in this fashion, periodic functions could be made to run, and blocking APIs could time out.

# 4 Configuration Data

The PSL is delivered as separate libraries, one of which is a configuration library. The configuration library, PSL_cfg_<chip or board>.a55L, provides system and target board-specific data to the PSL. The PSL_cfg_<chip or board>.a55L library is built with configuration data that is both device and target board-specific. It may be used without modification. The source files that are used to build this library are also supplied as part of the PSL. You can modify the data in these files and rebuild the library so that the PSL can be used on custom boards. Alternatively, you can modify these files and include them directly in your application. The configuration data files are:

- PSLclk_cfg_<chip or board>.c
- PSLvolt_cfg_<board>.c
- PSLvolt_cfg_null.c

The file PSLclk_cfg_<chip or board>.c contains configuration data relating to the clock(s) that will be controlled by the PSL. The data in this file is device-specific, but typically, it will include items such as input frequency, maximum operating frequency, the table of operating frequencies that will be supported by the PSL, and perhaps some latency information relating to frequency scaling operations. The variable declarations for this data, as well as the type definitions that define the structure of this data, are provided in PSLclk_cfg.h. See section 4.1 for more details.

The file PSLvolt_cfg_<board>.c contains configuration data relating to the operating voltages that are supported by the device, and data relating to the voltage regulator controller. This file will typically include a table of voltages and their corresponding maximum frequencies, data that specifies how the voltage regulator is controlled, and latency information relating to voltage scaling operations. The variable declarations for this data, as well as the type definitions that define the structure of this data, are provided in PSLvolt_cfg.h.

The file PSLvolt_cfg_null.c contains configuration data for a device that does not have a voltage regulator.

The PSL does not require a specific voltage regulator control scheme. The PSL provides built-in support for a default control scheme, and a mechanism that allows users to override the built-in support with their own support. See section 4.3 for more details.

## 4.1 PSLclk_cfg_<chip or board>.c

The file PSLclk_cfg_<chip or board>.c contains configuration data relating to the clock(s) that will be controlled by the PSL. The variable declarations for this data, as well as the type definitions that define the structure of this data, are provided in PSLclk_cfg.h. The remainder of this section describes the clock configuration data for the C5509a device, along with the initial values that are used for the evm5509a. Note that in the case of the C5509a, there is only one clock.

```
const unsigned PSL_clkmdRegAddr = 0x1C00;
```

PSL_clkmdRegAddr specifies the address of the clock mode register in I/O space.

```
const float PSL_cpuInputFreq =  12.0f;
```

PSL_cpuInputFreq specifies the input frequency (CLKIN) in MHz units. The input frequency on the evm5509a is 12 MHz.

```
const float PSL_cpuMaxFreq   = 200.0f;
```

PSL_cpuMaxFreq specifies the maximum frequency, in MHz units, at which it is safe to operate the CPU. The maximum frequency of the CPU on the evm5509a DSK is 200 MHz. The maximum frequency should be obtained from the device's data sheet.

```
const unsigned PSL_cpuFreqCnt = 16;
```

PSL_cpuFreqCnt specifies the number of frequencies that will be supported by the PSL. It also specifies the number of entries in the frequency table that is shown below. Acceptable values are those in the range 1 ...16.

```
PSL_CPUFreq PSL_cpuFreqTable[] = {
   {0, 0, 3, PSL_BYPASS}, //   3 MHz (input freq / 4), bypass mode
   {0, 0, 1, PSL_BYPASS}, //   6 MHz (input freq / 2), bypass mode
   {0, 0, 0, PSL_BYPASS}, //  12 MHz (input freq / 1), bypass mode

   { 4, 0, 0, PSL_LOCK},  //  48 MHz (input freq * ( 4 / 1)), lock mode
   { 5, 0, 0, PSL_LOCK},  //  60 MHz (input freq * ( 5 / 1)), lock mode
   { 6, 0, 0, PSL_LOCK},  //  72 MHz (input freq * ( 6 / 1)), lock mode
   { 7, 0, 0, PSL_LOCK},  //  84 MHz (input freq * ( 7 / 1)), lock mode
   { 8, 0, 0, PSL_LOCK},  //  96 MHz (input freq * ( 8 / 1)), lock mode
   { 9, 0, 0, PSL_LOCK},  // 108 MHz (input freq * ( 9 / 1)), lock mode
   { 10, 0, 0, PSL_LOCK}, // 120 MHz (input freq * ( 10 / 1)),lock mode
   { 11, 0, 0, PSL_LOCK}, // 132 MHz (input freq * (11 / 1)), lock mode
   { 12, 0, 0, PSL_LOCK}, // 144 MHz (input freq * (12 / 1)), lock mode
   { 13, 0, 0, PSL_LOCK}, // 156 MHz (input freq * (13 / 1)), lock mode
   { 14, 0, 0, PSL_LOCK}, // 168 MHz (input freq * (14 / 1)), lock mode
   { 15, 0, 0, PSL_LOCK}, // 180 MHz (input freq * (15 / 1)), lock mode
   { 16, 0, 0, PSL_LOCK}, // 192 MHz (input freq * (16 / 1)), lock mode
};
```

PSL_cpuFreqTable contains the frequencies that will be supported by the PSL. This table can contain from 1 to 16 frequencies. Internally, the PSL will create a setpoint for each frequency. The ordering of the setpoints will match the ordering of the frequencies in the table. See the description of the PSL_Setpoint type for more information on setpoints. Each entry in the table is of type PSL_CPUFreq, which is defined in the file PSLclk_cfg.h. The definition of PSL_CPUFreq is:

```
typedef struct {
    unsigned    PLL_mult;
    unsigned    PLL_div;
    unsigned    bypass_div;
    PSL_ClkMode mode;
} PSL_CPUFreq;
```

PLL_mult is a value in the range 2 ... 31. This value, in conjunction with the input frequency and PLL_div, determines the CPU clock frequency when operating in lock mode.

PLL_div is a value in the range 0 ... 3. This value, in conjunction with the input frequency and PLL_mult, determines the CPU clock frequency when operating in lock mode.

bypass_div is a value in the range 0 ... 3. It specifies the input frequency divider when operating in bypass mode.

The clock generator on the evm5509a device has two operating modes: bypass and lock mode. The type PSL_ClkMode, which is defined in PSLclk_cfg.h, represents these modes. The definition of PSL_ClkMode is:

```
typedef enum {    // clock's operating mode
   PSL_BYPASS,
   PSL_LOCK
} PSL_ClkMode;
```

In PSL_BYPASS mode, the PLL is bypassed and the frequency of the output clock signal is equal to the frequency of the input clock signal divided by 1, 2, 3, or 4. Because the PLL is disabled in this case, this mode consumes less power. In PSL_LOCK mode, the input frequency can be both multiplied and divided to produce the desired output frequency.

In bypass mode, the clock frequency can be calculated using the following equation:

$$Clock\ frequency = \frac{input\ frequency}{(bypass\_div + 1)}$$

In lock mode, the clock frequency can be calculated using the following equation:

$$Clock\ frequency = \frac{PLL\_mult}{(PLL\_div + 1)} * input\ frequency$$

See chapter 2 of the *TMS320C55x DSP Peripherals Reference Guide* (SPRU317) for more information on the clock generator.

## 4.2 PSLvolt_cfg_<board>.c

The file PSLvolt_cfg_<board>.c contains configuration data relating to the operating voltages that are supported by the device, and data relating to the voltage regulator controller. The variable declarations for this data, as well as the type definitions that define the structure of this data, are provided in PSLvolt_cfg.h. The remainder of this section describes the configuration data and initial values that are used for the evm5509a.

```
const unsigned PSL_voltCnt = 3;
```

PSL_voltCnt specifies the number of voltage points that are supported by the voltage regulator. The evm5509a has three different voltages at which it can operate, so the value PSL_voltCnt is 3 in this case.

```
PSL_GpioVoltRegTable PSL_gpioVoltRegTable[] = {
  {1.2f, 0x00},   // set GPIO bit 5 to a 1, bit 6 to a 0 for 1.2v
  {1.4f, 0x40},   // set GPIO bit 5 to a 0, bit 6 to a 1 for 1.4v
  {1.6f, 0x60},   // set GPIO bit 5 to a 1, bit 6 to a 1 for 1.6v
};
```

PSL_voltTable lists the supported voltage points and their associated maximum frequencies. These values should be obtained from the device's data sheet. The voltages must be listed in increasing order starting with the smallest. Each entry in the table is of type PSL_VoltTable, which is defined in the file PSLvolt_cfg.h. The definition of PSL VoltTable is:

```
typedef struct {
   float volt;  // voltage
   float freq;  // frequency for PSL_CPU_CLK
}
   PSL_VoltTable;
```

where volt specifies a voltage point, and freq specifies the maximum operating frequency for this voltage. The maximum operating frequency for a given voltage should be obtained from the device's data sheet.

```
PSL_voltDecreaseLatency = 2000; // 2.0 millisecond latency on voltage drop
PSL_voltIncreaseLatency = 500;  // 500 microsecond latency on voltage increase
```

These variables specify the maximum latencies incurred during voltage scaling operations. The latency is given in microseconds. Following the initiation of a voltage scaling operation, the latency is the time required before the new voltage has been reached. These latencies are system-specific and will have to be measured for each different target board.

```
PSL_VoltRegInitFunc PSL_voltRegInitFunc = PSL_gpioVoltRegInit_<board>;
```

PSL_voltRegInitFunc specifies the function that performs any one-time initialization that may be required before the voltage regulator can perform scaling operations. If the default implementation, which uses the GPIO pins to control the regulator, is NOT being used, the initialization function should be changed to the correct user-supplied initialization function.

Note that the GPIO implementation contains some functionality that is board specific. Use the initialization routine, PSL_gpioVoltRegInit_<board>, whose source code is delivered with the PSL, for a board-generic GPIO implementation that will work with custom boards. See section 4.3.2, Alternative Control Methods, for more details.

```
PSL_VoltRegScaleFunc PSL_voltRegScaleFunc = PSL_gpioVoltRegScale_<board>;
```

PSL_voltRegScaleFunc specifies the function that performs voltage scaling. If the default implementation, which uses the GPIO pins to control the regulator, is NOT being used, the scaling function should be changed to the correct user-supplied scaling function.

Note that the GPIO implementation contains some functionality that is specific to that board. Use the voltage scaling routine, PSL_gpioVoltRegScale_<board>, whose source code is delivered with the PSL, for a board-generic GPIO implementation that will work with custom boards. See section 4.3.2, Alternative Control Methods. for more details.

```
const unsigned PSL_gpioIodirAddr  = 0x3400;
const unsigned PSL_gpioIodataAddr = 0x3401;
```

These variables specify the addresses of GPIO IODIR and IODATA registers in I/O space. They are only used if the default implementation, which uses the GPIO pins to control the regulator, is being used.

```
const unsigned PSL_gpioPinsMask = 0x60;   // GPIO bits 5 and 6
```

PSL_gpioPinsMask is a mask that specifies which GPIO pin(s) is/are used to control the voltage regulator. The mask is only used if the default implementation, which uses the GPIO pins to control the regulator, is being used.

```
PSL_GpioVoltRegTable PSL_gpioVoltRegTable[] = {
  {1.2f, 0x00},  // set GPIO bit 5 to a 1, bit 6 to a 0 for 1.2v
  {1.4f, 0x40},  // set GPIO bit 5 to a 0, bit 6 to a 1 for 1.4v
  {1.6f, 0x60},  // set GPIO bit 5 to a 1, bit 6 to a 1 for 1.6v
};
```

The GPIO voltage regulator table contains the GPIO bit settings for each voltage that is supported by the regulator. This table is used only if the default implementation, which uses the GPIO pins to control the regulator, is being used.

## 4.3 Voltage Regulator Control

The PSL does not require a specific voltage regulator control scheme. The PSL provides built-in support for a default control scheme, and a mechanism that allows users to override the built-in support with their own support. These two cases are described further in the following sections.

### *4.3.1 Default Support*

PSL provides built-in support for controlling the voltage regulator via the GPIO pins.

If the voltage regulator on your board is controlled via GPIO pins, the configuration data in PSLvolt_cfg_<board>.c allows you to specify the GPIO pin(s) that will be used to control the regulator and the pin values for each voltage. As an example, consider the default values that are provided in PSLvolt_cfg_evm5509a.c for the evm5509a, where the voltage regulator is controlled by GPIO pins 5 and 6 and supports three voltage points. The values for PSL_gpioPinsMask and PSL_gpioVoltRegTable are:

```
const unsigned PSL_gpioPinsMask = 0x60;   // GPIO bits 5 and 6
PSL_GpioVoltRegTable PSL_gpioVoltRegTable[] = {
  {1.2f, 0x00},  // set GPIO bit 5 to a 1, bit 6 to a 0 for 1.2v
  {1.4f, 0x40},  // set GPIO bit 5 to a 0, bit 6 to a 1 for 1.4v
  {1.6f, 0x60},  // set GPIO bit 5 to a 1, bit 6 to a 1 for 1.6v
};
```

where PSL_gpioPinsMask specifies the GPIO pins and PSL_gpioVoltRegTable specifies the value of the GPIO pin for each supported voltage. In this case, the regulator is controlled by a two pins, which are GPIO pin 5 and pin 6.

### *4.3.2 Alternative Control Methods*

The PSL allows users to override the default voltage regulator support with their own support. The configuration data provides function pointers that enable users to supply their own voltage regulator control functions. In the case of the evm5509a, the function pointers refer to the functions that use GPIO pins to control the regulator. The default implementation can be overridden by changing the function pointers to refer to user-supplied functions. Two functions are required: an initialization function and a scaling function.

The initialization function is of type PSL_VoltRegInitFunc, which is defined as:

**PSL_VoltRegInitFunc** *Performs any one-time initialization*

**Function**
```
typedef void (* PSL_VoltRegInitFunc)(void);
```

**Description** Function that performs any one-time initialization that may be required before the voltage regulator can perform scaling operations.

**Parameters** none

**Return Value** none

The scaling function of type PSL_VoltRegScaleFunc, which is defined as:

**PSL_VoltRegScaleFunc** *Scales the voltage to the specified voltage*

**Function**
```
typedef void (* PSL_VoltRegScaleFunc)(float currVoltage,
                                      float newVoltage,
                                      float currFrequency,
                                      int   wait);
```

**Description** Function that scales the voltage to the specified voltage. If wait is TRUE, wait until the new voltage has been reached.

**Parameters**

currVoltage [in] The current voltage.

newVoltage [in] The new voltage.

currFrequency [in] The current clock frequency of the device that is executing this routine. The frequency may be needed to implement a delay loop in cases where wait is TRUE and the voltage regulator provides no notification as to when the new voltage has been reached.

wait [in] TRUE if this routine should wait for the new voltage to reach the regulation point. FALSE otherwise.

**Return Value** none

# 5 Modifying the Configuration Data

The PSL configuration data is contained in files PSLclk_cfg_<chip or board>.c and PSLvolt_cfg_<board>.c. The values specified in these files can be modified as necessary so that the PSL can be used on custom target boards. Once these files have been modified, you can either rebuild the appropriate configuration library, or include the files directly in your application.

The libraries can be rebuilt within gmake style make files, which are provided to rebuild PSL_cfg_<chip>.a55L and PSL_cfg_<chip or board>.a55L.

As an alternative to rebuilding the configuration library, the configuration files can be rebuilt directly into your application. In this case, there is no need to link the configuration library into your application. The configuration files are rebuilt with the rest of your application. However, you must add the PSL include directory to your “include” search path so that the required header files can be located.

If the voltage regulator setup on your target board requires you to override the default voltage regulator support, you must supply your own implementation of the `PSL_voltRegInitFunc and PSL_voltRegScaleFunc` functions (see section 4.3.2, Alternative Control Methods, for details). When doing so, you may add these functions to the configuration library or directly to your application. These functions can be added to the configuration library by adding the appropriate source files to the gmake style makefile.

# 6 PSL API

## 6.1 PSL Types

There are several PSL types that are used by the PSL API:

- PSL_ClkID
- PSL_Setpoint
- PSL_Status
- PSL_PrologueFunc
- PSL_EpilogueFunc
- PSL_ClkMode

These types are described below.

### *6.1.1 PSL_ClkID*

```
typedef enum {
   PSL_CPU_CLK = 0
} PSL_ClkID;
```

PSL_ClkID defines the different clocks that are supported by the PSL. In the case of the C5509a, there is only one clock. Multi-core devices that have more than one clock will define multiple clocks. The definition of PSL_ClkID is located in PSLclk_cfg.h.

### *6.1.2 PSL_Setpoint*

```
typedef unsigned PSL_Setpoint;
```

PSL_Setpoint is an unsigned integer type that is used to refer to a discrete frequency and voltage operating point (i.e., a setpoint) that is supported by the PSL. The voltage of a setpoint is the minimum operating voltage that is required to support the frequency of the setpoint.

All PSL operations are performed on setpoints. Each clock that is supported by the PSL has a separate set of setpoints. The number of setpoints associated with a specific clock corresponds directly to the number of entries in the clock's frequency table, which is located in PSLclk_cfg_<chip or board>.c. The ordering of the setpoints also corresponds directly to the ordering specified by the frequency table. For the evm5509a device, the PSL supports one clock, which is referred to as PSL_CPU_CLK. Thus, the evm5509a device has only one set of setpoints. This type is defined in PSL.h.

### 6.1.3 PSL_Status

```
typedef enum {
   PSL_OK,
   PSL_INVALID_CLK,
   PSL_INVALID_FREQ,
   PSL_INVALID_INITIAL_FREQ,
   PSL_INVALID_INITIAL_VOLTAGE,
   PSL_INVALID_SETPOINT,
   PSL_MAX_FREQ_EXCEEDED,
   PSL_MAX_VOLTAGE_EXCEEDED,
   PSL_INCOMPATIBLE_VOLTAGE,
   PSL_INCOMPLETE_INITIALIZATION,
   PSL_CANNOT_CHANGE_SETPOINT
   PSL_NOT_INITIALIZED
} PSL_Status;
```

PSL_Status specifies the return status of several PSL functions. If the return status is PSL_OK, the function executed without error. A return value other than PSL_OK indicates that the function encountered an error during execution. This type is defined in PSL.h.

### 6.1.4 PSL_PrologueFunc

```
typedef void (* PSL_PrologueFunc)(unsigned     count,
                                  PSL_ClkID    *clks,
                                  PSL_Setpoint *currentSetpoints,
                                  PSL_Setpoint *newSetpoints);
```

PSL_PrologueFunc is a pointer to a function that is called immediately before a scaling operation (i.e., immediately before a setpoint change). This callback allows users to perform any necessary peripheral modifications that may be required as a result of the upcoming scaling operation. For example, the user may need to stop a timer prior to changing the clock frequency. This type is defined in PSL.h.

### 6.1.5 PSL_EpilogueFunc

```
typedef void (* PSL_EpilogueFunc)(unsigned     count,
                                  PSL_ClkID    *clks,
                                  PSL_Setpoint *oldSetpoints,
                                  PSL_Setpoint *currentSetpoints);
```

PSL_EpilogueFunc is a pointer to a function that is called immediately after a scaling operation (i.e., immediately after a setpoint change). This callback allows users to perform any necessary peripheral modifications that may be required as a result of the just completed scaling operation. For example, the user may need to reprogram and restart a timer after changing the clock frequency. This type is defined in PSL.h.

### 6.1.6 PSL_ClkMode

```
typedef enum {    // clock's operating mode
   PSL_BYPASS,
   PSL_LOCK
} PSL_ClkMode;
```

This type specifies the different operating modes of the CPU clock. The clock on the C5509A devices can operate in either bypass or lock mode. The definition of PSL_ClkMode is located in PSLclk_cfg.h.

## 6.2 PSL Functions Overview

**Table 1. PSL Functions**

| Function | Description |
|---|---|
| PSL_initialize | Initializes the PSL. |
| PSL_getNumSetpoints | Returns the number of valid setpoints for the specified clocks. If a clock has n valid setpoints, the valid setpoints for that clock are those in the range 0 ... n–1. |
| PSL_getSetpoints | Returns the current setpoint for each of the specified clocks. |
| PSL_changeSetpoints | For each of the specified clocks, initiates a scaling operation to the new set point. This includes setting the CPU clock frequency and clock mode, and possibly the voltage to those specified by the clock's new setpoint. |
| PSL_querySetpoints | Returns the clock frequency, clock mode, and voltage that are associated with each of the specified setpoints. |
| PSL_querySetpointFrequencies | Returns the clock frequency that is associated with each of the specified set points. |
| PSL_querySetpointVoltages | Returns the voltage that is associated with each of the specified setpoints. |
| PSL_querySetpointModes | Returns the clock mode that is associated with each of the specified setpoints. |
| PSL_querySetpointTransitions | Returns the maximum scaling latencies that are associated with each of the specified setpoints changes. |
| PSL_getFrequencies | Returns the current clock frequency for each of the specified clocks. |
| PSL_getModes | Returns the current clock mode (e.g. PSL_BYPASS or PSL_LOCK) for each of the specified clocks. |
| PSL_getVoltage | Return the current voltage. |

## 6.3 PSL Functions Descriptions

**PSL_initialize** *Performs any initialization required by the power scaling library*

**Function**

```
PSL_Status PSL_initialize(unsigned    count,
                          PSL_ClkID *clks,
                          unsigned  *initFrequencies,
                          float      initVoltage)
```

**Description**

Perform any initialization required by the power scaling library.

The initial clock frequency and operating mode for each clock are determined according to the values specified in the initFrequencies array. The values in this array are indexes into a clock's associated frequency table, which is part of the user configurable data located in PSLclk_cfg_<chip or board>.c. This routine does actually change the frequency of any clock. An initial frequency MUST be supplied for every clock that is defined by the enum type PSL_ClkID, which is located in PSLclk_cfg.h. The initial voltage is specified by initVoltage. This routine does change the voltage. The initial voltage must match one of the voltages specified in the voltage table located in PSLvolt_cfg_<board>.c.

The initial setpoint for each clock will specify the clock's initial frequency and the minimum voltage required for that frequency.

This routine should be called once during target initialization. If this routine is called multiple times, all calls after the first successful call will return PSL_OK. In this case, no re-initialization occurs and the current setpoints are not changed. If none of the previous calls were successful, subsequent calls will attempt initialization as described above.

**Parameters**

| | |
|---|---|
| count [in] | Specifies the number of clocks pointed to by clks. The count MUST specify the number of clocks defined by the enum type PSL_ClkID, which is located in PSLclk_cfg.h. |
| *clks [in] | Pointer to locations that specify the clocks. The number of clocks referred to by the pointer should match the count. Every clock that is defined by the enum type PSL_ClkID, which is located in PSLclk_cfg.h, MUST be present in the array. |

| | |
|---|---|
| *initFrequencies [in] | Pointer to locations that specify the initial frequency of each clock. The values in this array are indexes into a clock's associated frequency table, which is part of the user configurable data located in PSLclk_cfg.c. The initial frequency for clks[0] is specified by initFrequencies[0], the initial frequency for clks[1] is specified by initFrequencies[1],etc. |
| initVoltage [in] | The initial voltage. |

### Return Value

| | |
|---|---|
| PSL_OK | If all initialization required for the correct operation of the scaling library succeeds. If initialization does not succeed, all setpoints for all clocks are considered invalid. |
| PSL_INVALID_CLK | If any of the specified clocks are invalid. |
| PSL_INVALID_FREQ | If any of the clock frequencies tables are empty, there are more than 16 entries in any of the frequencies tables, or any of the values (mult, div, mode) for a specific clock are invalid. The clock frequency tables are part of the user configurable data located in PSLclk_cfg.c. |
| PSL_INVALID_INITIAL_FREQ | If any of the values specified in the initFrequencies array are invalid indexes into the corresponding clock's frequency table. The clock frequency tables are part of the user configurable data located in PSLclk_cfg.c. |
| PSL_MAX_FREQ_EXCEEDED | If any of the frequencies specified in a clock's frequency table exceed the maximum operating frequency of the device that the clock is controlling. The maximum frequencies are part of the user configurable data located in PSLclk_cfg_<cpu or board>.c. |
| PSL_INCOMPATIBLE_VOLTAGE | If the initial voltage as specified by initVoltage is less than the voltage required by any of the initial setpoints. |
| PSL_INVALID_INITIAL_VOLTAGE | If the initial voltage as specified by initVoltage is not one of the voltages specified in the voltage table located in PSLvolt_cfg_<board>.c. |

| | |
|---|---|
| PSL_MAX_VOLTAGE_EXCEEDED | If any of the values in the user configurable data located in PSLvolt_cfg.c are beyond the maximum supported voltage. |
| PSL_INCOMPLETE_INITIALIZATION | If an initial frequency is not supplied for every clock that is defined by the enum type PSL_ClkID, which is located in PSLclk_cfg.h. |

## PSL_getNumSetpoints *Returns the number of valid setpoints for the specified clocks*

**Function**

```
PSL_Status PSL_getNumSetpoints(unsigned   count,
                               PSL_ClkID   *clks,
                               unsigned   *numSetpoints)
```

**Description**

This function returns the number of valid setpoints for the specified clocks. If a clock has n valid setpoints, the valid setpoints for that clock are those in the range (0…n−1). No setpoint for any clock is considered valid until the power scaling library has been successfully initialized.

**Parameters**

| | |
|---|---|
| count [in] | Specifies the number of clocks pointed to by clks. |
| *clks [in] | Pointer to locations that specify the clocks. The number of clocks referred to by the pointer should match the count. |
| *numSetpoints [out] | Pointer to locations to store the setpoint count for each of the clocks referred to by the clks pointer. The number of valid setpoints for clks[0] will be returned in numSetpoints[0], the valid number of setpoints for clks[1] will be returned in numSetpoints[1], etc. |

**Return Value**

| | |
|---|---|
| PSL_OK | If all of the specified clocks are valid. |
| PSL_INVALID_CLK | If any of the specified clocks are invalid. |
| PSL_NOT_INITIALIZED | If PSL has not been initialized. |

## PSL_getSetpoints *Returns the current setpoint for each of the specified clocks*

**Function**

```
PSL_Status PSL_getSetpoints(unsigned         count,
                            PSL_ClkID        *clks,
                            PSL_Setpoint     *setpoints)
```

**Description**

This function returns the current setpoint for each of the specified clocks.

### Parameters

| | |
|---|---|
| count [in] | Specifies the number of clocks pointed to by clks. |
| *clks [in] | Pointer to locations that specify the clocks. The number of clocks referred to by the pointer should match the count. |
| *setpoints [out] | Pointer to locations to store the current setpoint for each of the clocks referred to by the clks pointer. The current setpoint for clks[0] will be returned in setpoints[0], the current setpoint for clks[1] will be returned in setpoints[1], etc. |

### Return Value

| | |
|---|---|
| PSL_OK | If all of the specified clocks are valid. |
| PSL_INVALID_CLK | If any of the specified clocks are invalid. |
| PSL_NOT_INITIALIZED | If PSL has not been initialized. |

## PSL_changeSetpoints *Initiates a scaling operation to the new setpoint*

### Function

```
PSL_Status PSL_changeSetpoints(unsigned count,
                               PSL_ClkID      *clks,
                               PSL_Setpoint   *newSetpoints,
                               int             scaleVoltage,
                               int             waitForVoltScale,
                               PSL_PrologueFunc prologueFunc,
                               PSL_EpilogueFunc epilogueFunc)
```

### Description

For each of the specified clocks, this function initiates a scaling operation to the new setpoint. This includes setting the CPU clock frequency and clock mode to those specified by the clock's new setpoint.

If scaleVoltage is TRUE and the current voltage is not sufficient for any of the new setpoints, then the voltage will be increased to the lowest level that will support all the new setpoints. In this case, the new voltage will also be sufficient for any current setpoint that is not being changed. If a lower voltage is sufficient for all new setpoints as well as all current setpoints that are not being changed, the voltage will be decreased to the lowest level that will support all of these setpoints.

This routine will not return until the clocks are generating the new frequencies specified by the setpoints. If waitForVoltScale is TRUE and the voltage was actually scaled, then this routine will also wait until the new voltage is reached. In addition, if a voltage increase was required as part of the setpoint changes, or if the device is in an unstable state until the new voltage is reached, then this routine will also wait for the voltage scaling to complete,

regardless of waitForVoltScale.

Prior to initiating any scaling operations, this routine will call the function referenced by prologueFunc. If prologueFunc is NULL, no function is called.

Similarly, following the scaling operations, this routine will call the function referred to by epilogueFunc. The call to epilogueFunc will not occur until the clocks are generating the new frequencies. If this routine must wait for the new voltage to be reached, then the call to epilogueFunc will not occur until the voltage has been reached. If epilogueFunc is NULL, no function is called.

### Parameters

| | |
|---|---|
| count [in] | Specifies the number of clocks pointed to by clks. |
| *clks [in] | Pointer to locations that specify the clocks. The number of clocks referred to by the pointer should match the count. |
| *newSetpoints [in] | Pointer to locations that specify the new setpoint for each of the clocks referred to by the clks pointer. The new setpoint for clks[0] is specified by newSetpoints[0], the new setpoint for clks[1] is specified by newSetpoints[1], etc. |
| scaleVoltage [in] | TRUE if the voltage should be scaled when necessary.<br>FALSE if the voltage should not be scaled. |
| waitForVoltScale [in] | TRUE if this routine should wait for the new voltage to be reached after initiating the voltage scaling.<br>FALSE if waiting is not required. Note that this parameter is ignored if a voltage increase is required or if the device is in an unstable state until the new voltage is reached. In these cases, this routine will always wait for the voltage scaling to complete. |
| prologueFunc [in] | Function called prior to scaling operations. NULL if no function is to be called. |
| epilogueFunc [in] | Function called after the scaling operations have completed. NULL if no function is to be called. |

### Return Value

| | |
|---|---|
| PSL_OK | If the setpoint changes were successful. |
| PSL_INVALID_CLK | If any of the specified clocks are invalid. |
| PSL_INVALID_SETPOINT | If any of the new setpoints are invalid. A clock's valid setpoints are those in the range (0 … n–1), where n is the number of valid setpoints returned by PSL_getNumSetpoints(). No scaling operations are performed if any of the setpoints are invalid. |

| | |
|---|---|
| PSL_INCOMPATIBLE_VOLTAGE | If scaleVoltage is FALSE and the current voltage is less than the voltage required by any of the new setpoints. No scaling operations are performed in this case. |
| PSL_CANNOT_CHANGE_SETPOINT | If the setpoint could not be changed. |
| PSL_NOT_INITIALIZED | If PSL has not been initialized. |

## PSL_querySetpoints *Returns the clock frequency, clock mode, and voltage*

### Function

```
PSL_Status PSL_querySetpoints(unsigned        count,
                              PSL_ClkID      *clks,
                              PSL_Setpoint   *setpoints,
                              float          *frequencies,
                              float          *voltages,
                              PSL_ClkMode    *modes)
```

### Description

This functions returns the clock frequency, clock mode, and voltage that are associated with each of the specified setpoints.

### Parameters

| | |
|---|---|
| count [in] | Specifies the number of clocks pointed to by clks. |
| *clks [in] | Pointer to locations that specify the clocks. The number of clocks referred to by the pointer should match the count. |
| *setpoints [in] | Pointer to locations that specify the setpoints that are being queried. The setpoint for clks[0] is specified by setpoints[0], the setpoint for clks[1] is specified by setpoints[1], etc. |
| *frequencies [out] | Pointer to locations to store the frequency associated with each setpoint. The frequency for setpoints[0] will be returned in frequencies[0], the frequency for setpoints[1] will be returned in frequencies[1], etc. |
| *voltages [out] | Pointer to locations to store the voltages associated with each setpoint. A setpoint's voltage is the minimum voltage required for the setpoint's frequency. Note that this voltage may not be equal to the current voltage if voltage scaling was not performed during PSL_changeSetpoint, or if the current setpoint for another clock required a higher voltage. The voltage for setpoints[0] will be returned in voltages[0], the voltage for setpoints[1] will be returned in voltages[1], etc. |
| *modes [out] | Pointer to locations to store the clock mode associated with each setpoint (e.g. PSL_BYPASS or PSL_LOCK). The clock mode for setpoints[0] will be returned in modes[0], the clock mode for setpoints[1] will be returned in modes[1], etc. |

### Return Value

| | |
|---|---|
| PSL_OK | If the specified clocks and setpoints are valid. |
| PSL_INVALID_CLK | If any of the specified clocks are invalid. |
| PSL_INVALID_SETPOINT | If any of the setpoints are invalid. A clock's valid setpoints are those in the range 0 … n–1, where n is the number of valid setpoints returned by PSL_getNumSetpoints(). |
| PSL_NOT_INITIALIZED | If PSL has not been initialized. |

## PSL_querySetpointFrequencies *Returns the clock frequency*

### Function

```
PSL_Status PSL_querySetpointFrequencies(unsigned        count,
                                        PSL_ClkID     *clks,
                                        PSL_Setpoint  *setpoints,
                                        float        *frequencies)
```

### Description

This function returns the clock frequency that is associated with each of the specified setpoints.

### Parameters

| | |
|---|---|
| count [in] | Specifies the number of clocks pointed to by clks. |
| *clks [in] | Pointer to locations that specify the clocks. The number of clocks referred to by the pointer should match the count. |
| *setpoints [in] | Pointer to locations that specify the setpoints that are being queried. The setpoint for clks[0] is specified by setpoints[0], the setpoint for clks[1] is specified by setpoints[1], etc. |
| *frequencies [out] | Pointer to locations to store the frequency associated with each setpoint. The frequency for setpoints[0] will be returned in frequencies[0], the frequency for setpoints[1] will be returned in frequencies[1], etc. |

### Return Value

| | |
|---|---|
| PSL_OK | If the specified clocks and setpoints are valid. |
| PSL_INVALID_CLK | If any of the specified clocks are invalid. |
| PSL_INVALID_SETPOINT | If any of the setpoints are invalid. A clock's valid setpoints are those in the range 0 … n–1, where n is the number of valid setpoints returned by PSL_getNumSetpoints(). |
| PSL_NOT_INITIALIZED | If PSL has not been initialized. |

## PSL_querySetpointVoltages *Returns the voltage*

**Function**

```
PSL_Status PSL_querySetpointVoltages(unsigned       count,
                                     PSL_ClkID     *clks,
                                     PSL_Setpoint  *setpoints,
                                     float          *voltages)
```

**Description**

This function returns the voltage that is associated with each of the specified setpoints.

**Parameters**

| | |
|---|---|
| count [in] | Specifies the number of clocks pointed to by clks. |
| *clks [in] | Pointer to locations that specify the clocks. The number of clocks referred to by the pointer should match the count. |
| *setpoints [in] | Pointer to locations that specify the setpoints that are being queried. The setpoint for clks[0] is specified by setpoints[0], the setpoint for clks[1] is specified by setpoints[1], etc. |
| *voltages [out] | Pointer to locations to store the voltages associated with each setpoint. A setpoint's voltage is the minimum voltage required for the setpoint's frequency. Note that this voltage may not be equal to the current voltage if voltage scaling was not performed during PSL_changeSetpoint, or if the current setpoint for another clock required a higher voltage. The voltage for setpoints[0] will be returned in voltages[0], the voltage for setpoints[1] will be returned in voltages[1], etc. |

**Return Value**

| | |
|---|---|
| PSL_OK | If the specified clocks and setpoints are valid. |
| PSL_INVALID_CLK | If any of the specified clocks are invalid. |
| PSL_INVALID_SETPOINT | If any of the setpoints are invalid. A clock's valid setpoints are those in the range 0 … n–1, where n is the number of valid setpoints returned by PSL_getNumSetpoints(). |
| PSL_NOT_INITIALIZED | If PSL has not been initialized. |

## PSL_querySetpointModes *Returns the clock mode*

**Function**

```
PSL_Status PSL_querySetpointModes(unsigned       count,
                                  PSL_ClkID     *clks,
                                  PSL_Setpoint  *setpoints,
                                  PSL_ClkMode   *modes)
```

**Description**

This function returns the clock mode that is associated with each of the specified setpoints.

### Parameters

| | |
|---|---|
| count [in] | Specifies the number of clocks pointed to by clks. |
| *clks [in] | Pointer to locations that specify the clocks. The number of clocks referred to by the pointer should match the count. |
| *setpoints [in] | Pointer to locations that specify the setpoints that are being queried. The setpoint for clks[0] is specified by setpoints[0], the setpoint for clks[1] is specified by setpoints[1], etc. |
| *modes [out] | Pointer to locations to store the clock mode associated with each setpoint (e.g. PSL_BYPASS or PSL_LOCK). The clock mode for setpoints[0] will be returned in modes[0], the clock mode for setpoints[1] will be returned in modes[1], etc. |

### Return Value

| | |
|---|---|
| PSL_OK | If the specified clocks and setpoints are valid. |
| PSL_INVALID_CLK | If any of the specified clocks are invalid. |
| PSL_INVALID_SETPOINT | If any of the setpoints are invalid. A clock's valid setpoints are those in the range 0 … n–1, where n is the number of valid setpoints returned by PSL_getNumSetpoints(). |
| PSL_NOT_INITIALIZED | If PSL has not been initialized. |

## PSL_querySetpointTransitions *Returns the maximum scaling latencies*

### Function

```
PSL_Status PSL_querySetpointTransitions(unsigned     count,
                                        PSL_ClkID    *clks,
                                        PSL_Setpoint  *fromSetpoints,
                                        PSL_Setpoint *toSetpoints,
                                        unsigned  *freqScalingLatencies,
                                        unsigned  *voltageScalingLatency)
```

### Description

This function returns the maximum scaling latencies that are associated with each of the specified setpoints changes.

### Parameters

| | |
|---|---|
| count | Specifies the number of clocks pointed to by clks. |
| *clks [in] | Pointer to locations that specify the clocks. The number of clocks referred to by the pointer should match the count. |
| *fromSetpoints [in] | Pointer to locations that specify the source setpoints. The source setpoint for clks[0] is specified by fromSetpoints[0], the source setpoint for clks[1] is specified by fromSetpoints[1], etc. |

| | |
|---|---|
| *toSetpoints [in] | Pointer to locations that specify the destination Setpoints. The destination setpoint for clks[0] is specified by toSetpoints[0], the destination setpoint for clks[1] is specified by toSetpoints[1], etc. |
| *freqScalingLatencies [out] | Pointer to locations to store the maximum latencies associated with each of the frequency scaling operations that will occur during the specified setpoint changes. The latencies are specified in microseconds. Following the initiation of a frequency scaling operation, the latency is the time required before the clock starts generating the new frequency. The latency for the setpoint change associated with clks[0] is specified by freqScalingLatencies[0], the latency for the setpoint change associated with clks[1] is specified by freqScalingLatencies[1], etc. |
| *voltageScalingLatency [out] | Location to store the maximum latency associated with the voltage scaling that may occur during the specified setpoint changes. The latency is given in microseconds. Following the initiation of the voltage scaling operation, the latency is the time required before the new voltage has been reached. |

### Return Value

| | |
|---|---|
| PSL_OK | If the specified clocks and setpoints are valid. |
| PSL_INVALID_CLK | If any of the specified clocks are invalid. |
| PSL_INVALID_SETPOINT | If any of the setpoints are invalid. A clock's valid setpoints are those in the range (0 ... n–1), where n is the number of valid setpoints returned by PSL_getNumSetpoints(). |
| PSL_NOT_INITIALIZED | If PSL has not been initialized. |

## PSL_getFrequencies *Returns the current clock frequency*

### Function

```
PSL_Status PSL_getFrequencies(unsigned     count,
                              PSL_ClkID    *clks,
                              float        *frequencies)
```

### Description

This function returns the current clock frequency for each of the specified clocks.

### Parameters

| | |
|---|---|
| count [in] | Specifies the number of clocks pointed to by clks. |
| *clks [in] | Pointer to locations that specify the clocks. The number of clocks referred to by the pointer should match the count. |
| *frequencies [out] | Pointer to locations to store the current frequency of each of the specified clocks. The current frequency of a clock is the same as the frequency returned by PSL_querySetpointFrequencies for that clock when that function is called with the clock's current setpoint (i.e., the current frequency of a clock is always the same as the frequency of the clock's current setpoint). The current frequency for clks[0] will be returned in frequencies[0], the current frequency for clks[1] will be returned in frequencies[1], etc. |

### Return Value

| | |
|---|---|
| PSL_OK | If all of the specified clocks are valid. |
| PSL_INVALID_CLK | If any of the specified clocks are invalid. |
| PSL_NOT_INITIALIZED | If PSL has not been initialized. |

## PSL_getModes

*Returns the current clock mode*

**Function**

```
PSL_Status PSL_getModes(unsigned      count,
                        PSL_ClkID    *clks,
                        PSL_ClkMode *modes)
```

**Description**

This function returns the current clock mode (e.g., PSL_BYPASS or PSL_LOCK) for each of the specified clocks.

**Parameters**

| | |
|---|---|
| count [in] | Specifies the number of clocks pointed to by clks. |
| *clks [in] | Pointer to locations that specify the clocks. The number of clocks referred to by the pointer should match the count. |
| *modes [out] | Pointer to locations to store the current mode of each of the specified clocks. The current operating mode of a clock is the same as the mode returned by PSL_querySetpointModes when that function is called with the clock's current setpoint (i.e., the current mode of a clock is always the same as the mode of the clock's current setpoint). The current mode for clks[0] will be returned in modes[0], the current mode for clks[1] will be returned in modes[1], etc. |

**Return Value**

| | |
|---|---|
| PSL_OK | If all of the specified clocks are valid. |
| PSL_INVALID_CLK | If any of the specified clocks are invalid. |
| PSL_NOT_INITIALIZED | If PSL has not been initialized. |

## PSL_getVoltage

*Returns the current voltage*

**Function**

```
float PSL_getVoltage()
```

**Description**

This function returns the current voltage.

**Return Value**

This function returns the current voltage. If voltage scaling was not performed in any of the calls to PSL_changeSetpoint, the current voltage is assumed to be the initial voltage as specified in the user configurable data located in PSLvolt_cfg.c. If voltage scaling is being done, the current voltage will be the lowest voltage that is sufficient for all of the current setpoints.

# 7 PSL “debug” Library

The PSL_debug_<chip>.a55L library performs many error checks that prevent the device from running at unsupported frequencies and voltages and unsupported frequency/voltage combinations. The checks guard against situations such as over clocking and other situations that could cause damage to the device. However, these error checks do increase the code size of the PSL. For this reason separate libraries have been provided that do not contain these error checks. These libraries are located in the same directory as PSL_<chip>.a55L.

The default PSL that BIOS links with is PSL_<chip>.a55L and does not check for the following errors:

1. PSL_INVALID_CLK
2. PSL_INVALID_FREQ
3. PSL_INVALID_INITIAL_FREQ
4. PSL_INVALID_INITIAL_VOLTAGE
5. PSL_INVALID_SETPOINT
6. PSL_MAX_FREQ_EXCEEDED
7. PSL_MAX_VOLTAGE_EXCEEDED
8. PSL_INCOMPATIBLE_VOLTAGE
9. PSL_INCOMPLETE_INITIALIZATION

It is recommended that development be done using the PSL_debug_<chip>.a55L library that contains the error checks. Error checking can be eliminated by using the PSL_<chip>.a55L library when there is no possibility that the error conditions listed above will occur.

# 8 Examples

The examples below assume that the configuration library was built using the configuration data shown here (note that only the data required to follow the examples is shown).

```
const float PSL_cpuInputFreq =  12.0f; // 12 MHz input clock (CLKIN)
                                       // frequency
const float PSL_cpuMaxFreq   = 200.0f;    // 200 MHz max operating
                                          // frequency

PSL_CPUFreq PSL_cpuFreqTable[] = {
   {0, 0, 3, PSL_BYPASS}, //  3 MHz (input freq / 4), bypass mode
   {0, 0, 1, PSL_BYPASS}, //  6 MHz (input freq / 2), bypass mode
   {0, 0, 0, PSL_BYPASS}, // 12 MHz (input freq / 1), bypass mode

   { 4, 0, 0, PSL_LOCK},  // 48 MHz (input freq * ( 4 / 1)), lock mode
   { 5, 0, 0, PSL_LOCK},  // 60 MHz (input freq * ( 5 / 1)), lock mode
   { 6, 0, 0, PSL_LOCK},  // 72 MHz (input freq * ( 6 / 1)), lock mode
   { 7, 0, 0, PSL_LOCK},  // 84 MHz (input freq * ( 7 / 1)), lock mode
   { 8, 0, 0, PSL_LOCK},  // 96 MHz (input freq * ( 8 / 1)), lock mode
   { 9, 0, 0, PSL_LOCK},  //108 MHz (input freq * ( 9 / 1)), lock mode
   {10, 0, 0, PSL_LOCK},  //120 MHz (input freq * (10 / 1)), lock mode
   {11, 0, 0, PSL_LOCK},  //132 MHz (input freq * (11 / 1)), lock mode
   {12, 0, 0, PSL_LOCK},  //144 MHz (input_freq * (12 / 1)), lock mode
   {13, 0, 0, PSL_LOCK},  //156 MHz (input freq * (13 / 1)), lock mode
   {14, 0, 0, PSL_LOCK},  //168 MHz (input_freq * (14 / 1)), lock mode
   {15, 0, 0, PSL_LOCK},  //180 MHz (input freq * (15 / 1)), lock mode
   {16, 0, 0, PSL_LOCK},  //192 MHz (input_freq * (16 / 1)), lock mode
};

PSL_VoltTable PSL_voltTable[] = {
  {1.2f, 108.0f}, // 0 MHz up to, and including 08 MHz, require a
                  // a minimum voltage of 1.2v.
  {1.4f, 144.0f}, //frequencies > 108 MHz up to and including 144 MHz
                  // require a minimum voltage of 1.4v.
  {1.6f, 200.0f}, // frequencies > 144 MHz up to the max frequency
                  // require a minimum voltage of 1.6V.
};
```

All examples assume that the initial frequency and voltage at system startup are 192 MHz and 1.6v respectively. (Note that these are also the reset frequency and voltage settings on the evm5509a.) The call to the PSL initialization routine specifies these initial settings. The initial frequency is specified by supplying an index to an entry in PSL_cpuFreqTable[]. In the configuration data shown here, 192 MHz is entry 15 in the table.

## 8.1 Example 1: Basic Usage

The first example highlights some of the basic PSL operations. It shows how the PSL is initialized, and how frequency and voltage changes are initiated by the changing of a setpoint.

This example will call three functions: func1, func2, and func3. It will execute func1 and func3 at 192 MHz and 1.6v. It will execute func2 at 72 MHz and 1.2v.

```
#include "PSL.h"

extern void func1();
extern void func2();
extern void func3();

void main (void)

{
   PSL_Status status;

   // Variable specifying PSL_ClkID that will be used
   // in all PSL calls.
   PSL_ClkID clk = PSL_CPU_CLK;

   // Index into PSL_cpuFreqTable[] that specifies initial freq
   // of 192 MHz.
   unsigned initFreqIndex = 15;

   // PSL_cpuFreqTable[5] represents 72 MHz
   // PSL_cpuFreqTable[15] represent 192 MHz
   PSL_Setpoint _72MHzSetpoint = 5;
   PSL_Setpoint _192MHzSetpoint = 15;

   // Initialize the PSL. The frequency following reset is
   // specified by PSL_cpuFreqTable[15]. The voltage following
   // reset is 1.6v.
   status = PSL_initialize(1, &clk, &initFreqIndex, 1.6f);

   if (status != PSL_OK)

   {
      // handle error;
      ...
     return;
   }

   // Execute func1 at 192 MHz (i.e., the initial frequency)
   func1();

   //Change frequency to 72 MHz
   status = PSL_changeSetpoints(1,
                                &clk,
                                &_72MHzSetpoint,
                                TRUE,     // change voltage also
                                FALSE,
                                NULL, NULL);
   if (status != PSL_OK) {
      // handle error
      ...
     return;
   }
       // Execute func2 at 72 MHz
      func2();

   // Change frequency back to 192 MHz
   status = PSL_changeSetpoints(1,
                                &clk,
```

```
                                    &_192MHzSetpoint,
                                    FALSE,   // do not change voltage
                                    FALSE,
                                    NULL, NULL);
   if (status != PSL_OK) {
      // handle error
      ...
     return;
   }
   // Execute func3 at 192 MHz
   func3();
}
```

When the required frequencies are statically known, as is the case in this example, the setpoints can be assigned values that correspond to indexes into PSL_cpuFreqTable[].

This is possible because the ordering of the setpoints will directly match the ordering of the frequencies in PSL_cpuFreqTable[]. Therefore, locating the setpoint that corresponds to a particular frequency does not require calls to the query routines in this case. For example, the setpoint _72MhzSetpoint is assigned the value 5 since PSL_cpuFreqTable[5] corresponds to 72 MHz. Similarly, the setpoint _192MHzSetpoint is assigned the value 15, which corresponds to the frequency specified by PSL_cpuFreqTable[15].

Separate calls are not required to change both the frequency and the voltage. Instead users initiate a frequency, and possibly a voltage change, through a single call to PSL_changeSetpoints. If the user instructs PSL_changeSetpoints to change the voltage, the voltage will be changed automatically to the voltage specified by the setpoint. This voltage will be the lowest voltage that is required to support the new frequency. In the above example, the initial frequency is 192 MHz and 1.6v. When the frequency is changed to 72 MHz, the voltage is automatically changed to 1.2v by the PSL. Similarly, when the frequency is changed back to 192 MHz, the PSL will automatically increase the voltage to 1.6v.

The PSL can operate in a mode that changes frequency only. This mode is useful if your target board does not have a voltage regulation capability. The fourth parameter of PSL_changeSetpoints specifies whether voltage scaling should be done. So for example, if the following call had been used to change the frequency to 72 MHz, the frequency would have been changed to 72 MHz, but the voltage would have remained at 1.6v.

```
status = PSL_changeSetpoints(1,
                             &clk,
                             &_72MHzSetpoint,
                             FALSE,     // do not change voltage
                             FALSE,
                             NULL, NULL);
```

A very important feature of the PSL is that it will always maintain a valid frequency/voltage setting. In the above example, the PSL would not allow the user to enter a state where the frequency is 192 MHz and the voltage is 1.2v . For example, if the second call to PSL_changeSetpoints had been:

```
status = PSL_changeSetpoints(1,
                             &clk,
                             &_192MHzSetpoint,
                             FALSE,     // do not change voltage
                                  FALSE,
                             NULL, NULL);
```

The return status would have been PSL_INCOMPATIBLE_VOLTAGE and no scaling operations would have been performed. This is because the frequency/voltage setting at the point of the call is 72 MHz and 1.2v. This call instructs the PSL to increase the frequency to 192 MHz, and leave the current voltage of 1.2v unchanged. However, since 192 MHz requires 1.6v, the PSL will not perform the frequency change and will return an error.

## 8.2 Example 2: Reprogramming Peripherals

When changing the frequency, one must consider the effects that the frequency change will have on the rest of the system. Obviously, frequency changes will affect the amount of time it takes to complete a certain operation. Therefore, frequency changes can only occur if the application's timing requirements continue to be satisfied. Frequency changes can also effect the operation of the peripherals. For example, the timer period or the EMIF may need to be reprogrammed as a result of a frequency change.

This example shows how to use the PSL's callback hooks to perform any necessary peripheral modifications that may be required as a result of an upcoming/just completed scaling operation. Callback hooks are simply function pointers that are called by the PSL_changeSetpoints function immediately before a scaling operation is initiated and immediately after the scaling operation completes. A function that is called immediately before a scaling operation has the following type:

```
typedef void (* PSL_PrologueFunc)(unsigned     count,
                                  PSL_ClkID    *clks,
                                  PSL_Setpoint *currentSetpoints,
                                  PSL_Setpoint *newSetpoints);
```

A function that is called immediately after a scaling operation completes is of type:

```
typedef void (* PSL_EpilogueFunc)(unsigned     count,
                                  PSL_ClkID    *clks,
                                  PSL_Setpoint *oldSetpoints,
                                  PSL_Setpoint *currentSetpoints);
```

In this example, a function of type PSL_PrologueFunc is used to stop Timer0 immediately before the scaling operation. A function of type PSL_EpilogueFunc is used to reprogram and restart Timer0 when the scaling operation completes. This example uses the CSL to program the timer. Note that the timer setup code is not shown below.

```
#include "PSL.h"

TIMER_Handle timer0Handle;

//-------------------------------------------------------
// Function to stop the timer. Called immediately before a
// scaling operation is initiated.
//-------------------------------------------------------
```

```
void StopTimer0(unsigned     count,
                PSL_ClkID    *clks,
                PSL_Setpoint *currentSetpoints,
                PSL_Setpoint *newSetpoints) {

   TIMER_stop(timer0Handle);

}

//-----------------------------------------------------------
// Function to reprogram and restart the timer. Called
// immediately after a scaling operation completes.
//
// Assumes that Timer0 is already stopped
//-----------------------------------------------------------
void RestartTimer0(unsigned     count,
                   PSL_ClkID    *clks,
                   PSL_Setpoint *oldSetpoints,
                   PSL_Setpoint *currentSetpoints) {

   float currFreq;
   unsigned long cycles;

   Uint16 timer0TCR;

   // Set timer loading (TLB) bit prior to initializing the period
   // and prescale registers.
   timer0TCR = TIMER_RGETH(timer0Handle, TCR);
   timer0TCR |= TIMER_FMK(TCR,TLB,1);           // TLB = 1;
   TIMER_RSETH(timer0Handle, TCR, timer0TCR);

   // Reprogram the period and prescale register such that the
   // interrupt period is 10 microseconds. The actual number of
   // CPU cycles is determined based on the current CPU frequency.

   PSL_querySetpointFrequencies(1, clks, currentSetpoints, &currFreq);
   cycles = (unsigned long)(10.0f * currFreq);

   // Write PRD register
   TIMER_FSETH(timer0Handle, PRD, PRD, cycles & 0xFFFF);

   // Write TDDR field of PRSC register
   TIMER_FSETH(timer0Handle, PRSC, TDDR, (cycles >> 16) & 0xF);

   // Restart the timer
   TIMER_start(timer0Handle);
}

#include "PSL.h"

extern void func1();
extern void func2();
extern void func3();

void main (void)

{
```

```
PSL_Status status;

// Variable specifying PSL_ClkID that will be used
// in all PSL calls.
PSL_ClkID clk = PSL_CPU_CLK;
// Index into PSL_cpuFreqTable[] that specifies initial freq
// of 192 MHz.
unsigned initFreqIndex = 15;

// PSL_cpuFreqTable[5] represents 72 MHz
// PSL_cpuFreqTable[15] represent 192 MHz
PSL_Setpoint _72MHzSetpoint = 5;
PSL_Setpoint _192MHzSetpoint = 15;

// Initialize the PSL. The frequency following reset is
// specified by PSL_cpuFreqTable[15]. The voltage following
// reset is 1.6v.
status = PSL_initialize(1, &clk, &initFreqIndex, 1.6f);

if (status != PSL_OK)
{
   // handle error;
   ...
  return;
}

// Execute func1 at 192 MHz (i.e., the initial frequency)
func1();

//Change frequency to 72 MHz.  Stop/Restart the Timer before/after the
// scaling operation.
status = PSL_changeSetpoints(
         1,
         &clk,
         &_72MHzSetpoint,
         TRUE,           // change voltage also
         FALSE,
         StopTimer0,     // Stop timer before scaling operation
         RestartTimer0); // Reprogram and start timer after scaling

if (status != PSL_OK) {
   // handle error
   ..
  return;
}
   // Execute func2 at 72 MHz
  func2();

// Change frequency to 192 MHz. Stop/Restart the Timer before/after the
// scaling operation.
  status = PSL_changeSetpoints(
         1,
         &clk,
         &_192MHzSetpoint,
         TRUE,           // change voltage also
         FALSE,
```

```
            StopTimer0,     // Stop timer before scaling operation
            RestartTimer0); // Reprogram and start timer after scaling

   if (status != PSL_OK) {
      // handle error
      ..
     return;
   }

   // Execute func3 at 192 MHz;
   func3();
}
```

The prologue and epilogue functions are passed to the PSL as the last two parameters of PSL_changeSetpoints. In this example, StopTimer0, the prologue function, is called immediately before the scaling operation is initiated. RestartTimer0, the epilogue function, is called immediately after the scaling has completed. Notice that the prologue function is passed the current and new setpoints, and the epilogue function is passed the old and current setpoints. Since the StopTimer0 function simply stops the timer, it does not use these parameters. However, the RestartTimer0 function determines the frequency of the current setpoint so that the period register of the timer can be reprogrammed correctly.

This example handles only one peripheral. Multiple peripherals can be handled by supplying a wrapper function that calls other routines to adjust the peripherals as necessary. For example, an epilogue function that restarts Timer0 and reprograms the EMIF might resemble the following:

```
void EpilogueFunc (unsigned     count,
                   PSL_ClkID    *clks,
                   PSL_Setpoint *oldSetpoints,
                   PSL_Setpoint *currentSetpoints) {

   // Determine frequency of current setpoint
   float freq;
   PSL_getSetPointFrequencies(1, clks, currSetpoints, &freq);

   // Reprogram and restart Timer1 based on the current frequency
   RestartTimer0(freq);

   // Reprogram EMIF based on current frequency
   ReprogramEMIF(freq);
}
```

## 8.3 Example 3: Query Operations

In the first two examples, the frequencies were statically known. In both cases the program switched back and forth between 192 MHz and 72 MHz. Since the frequencies were known upfront, the setpoints were directly assigned values that corresponded to indexes into PSL_cpuFreqTable[]. Locating the setpoint that corresponds to a particular frequency did not require the use of the query routines.

However, there may be certain situations where the frequency is dynamically determined. Consider an application that dynamically determines the lowest possible frequency based on information relating to the system's timing requirements and the timing information of each task or operation. In this case, after calculating the lowest possible frequency, the application must determine the setpoint that corresponds to the closest frequency that is greater than or equal to the calculated frequency. For example, assuming the configuration data shown earlier, if the calculated frequency were 70 MHz, the setpoint that corresponds to 72 MHz would have to be used since there is no setpoint that corresponds to 70 MHz.

The sample routine FindSetpoint shown below illustrates how to find the correct setpoint by using the query routines that are supplied by the PSL. The sample routine ChangeToLowestFreq further illustrates how query functions can be used.

```
//-----------------------------------------------------------
// Function to find a setpoint that corresponds to the
// closest frequency that is greater than or equal to the
// specified frequency.
//-----------------------------------------------------------
PSL_Setpoint FindSetpoint (float desiredFreq)
{
   unsigned i, numSetpoints;

   float freq,

   float closetFreq = PSL_cpuMaxFreq;
   PSL_Setpoint closestSetPoint;

   // Determine number of setpoints
   PSL_getNumSetpoints(1, &clk, &numSetpoints);

   for (i = 0; i < numSetpoints; i++) {
      // Determine frequency of setpoint i
     PSL_querySetpointFrequencies(1, &clk, &i, &freq);

      if ( (freq >= disiredFreq) && (freq < closestFreq ) {
         closestSetPoint = i;
         closestFreq = freq;
      }
   }

   return closestSetPoint;
}

//-----------------------------------------------------------
// Function that changes setpoint to lowest frequency that
// still meets real-time requirements.
//-----------------------------------------------------------

void ChangeToLowestFreq () {

   float currFreq, desiredFreq;

   unsigned freqScalingLatency, voltScalingLatency;

   PSL_Setpoint currSetpoint, desiredSetpoint,

   // Determine lowest frequency that meets timing requirements

   desiredFreq = CalcLowFreq();
```

```
    // Find nearest setpoint whose frequency is greater than or
    // equal to the desired frequency
    desiredSetpoint = FindSetpoint(desiredFreq);
    // Is the desired setpoint equal to the current setpoint?
    PSL_getSetpoints(1, &clk, &currSetpoint);

    if ( currSetpoint == desiredSetpoint) {
       // nothing to change
       return;
    }
    // Determine the latencies involved when scaling from the current
    // setpoint to the new setpoint
    PSL_querySetpointTransitions(1,
                                 &clk,
                                 &currSetpoint,
                                 &desiredSetpoint,
                                 &freqScalingLatency,
                                 &voltScalingLatency);
    // Determine if real-time requirements are still met
    // when latencies are considered
    if ( !CanScale(freqScalingLatency + voltScalingLatency) ) {
       return;
    }
    // Change setpoint
    status = PSL_changeSetpoints(
             1,
             &clk,
             &desiredSetpoint,
             TRUE,           // change voltage also
             FALSE,
             NULL,
             NULL);
    if (status != PSL_OK) {
       // handle error
       ..
      return;
    }

}
```

# Appendix A evm5509a Voltage Regulator Setup Reference

## A.1 1.6 Power Supply

The evm5509a operates from a single +5V external power supply connected to the main power input (J5). Internally, the +5V input is converted into +1.6V and +3.3V using a dual voltage regulator. The +1.6V supply is used for the DSP core while the +3.3V supply is used for the DSP's I/O buffers and all other chips on the board. The power connector is a 2.5mm barrel−type plug.

The core voltage on the evm5509a is selectable based on the output of GPIO5 and GPIO6 or CPLD control registers. If GPIO5 and GPIO6 are high or configured as an input the core voltage will remain at +1.6V. If GPIO5 and GPIO6 are driven low the voltage will drop to +1.2V.

| GPIO6 | GPIO5 | Core Voltage Selected |
|---|---|---|
| 0 | 0 | 1.2v |
| 0 | 1 | 1.4v |
| 1 | 0 | 1.4v |
| 1 | 1 | 1.6v |