

# Intelligent Power Management Framework

White Paper

xāplōs™

**Prepared By: Mark F Rodriguez**

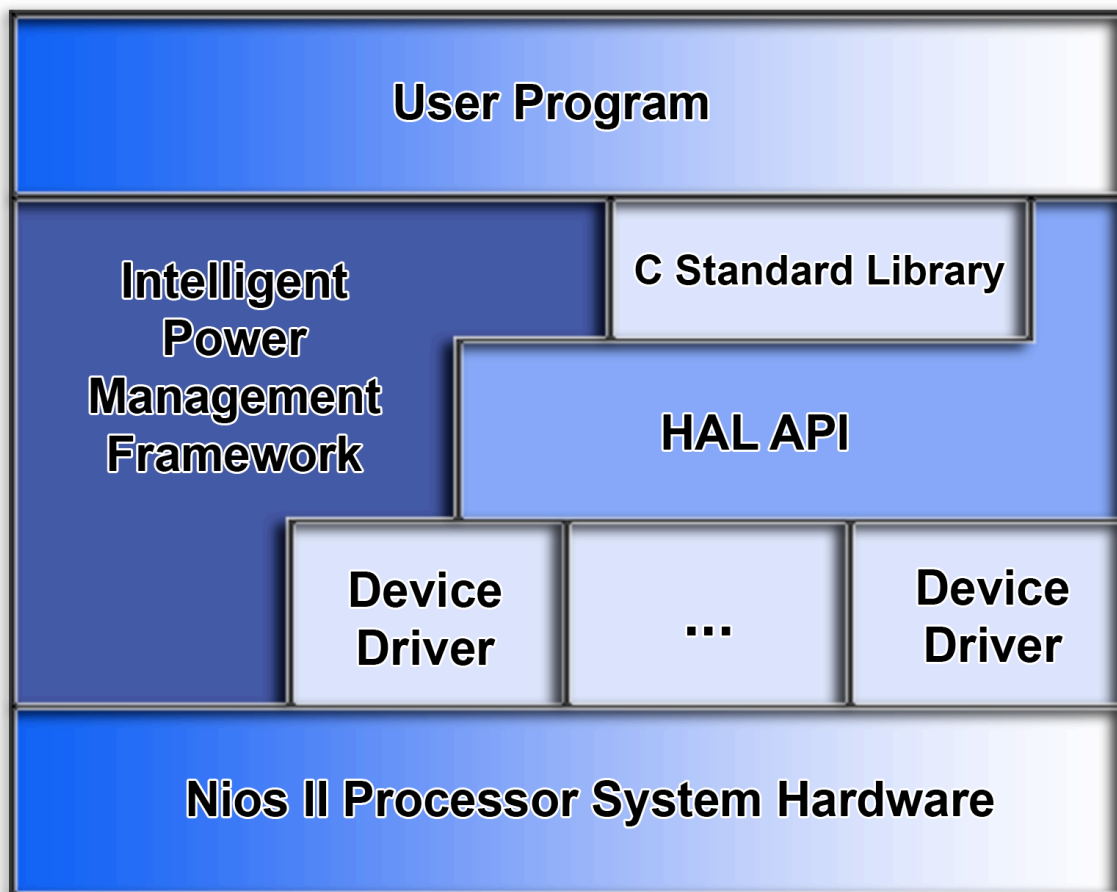
**Date: April 15, 2008**

## 1. Background

An increasing number of FPGA based products are including embedded processors, such as Altera's Nios II soft core. The ability to manage power consumption dynamically is becoming ever more important to reduce thermal dissipation or extend battery life. From a system design perspective, power can be conserved by reducing or halting the system clock, or by hibernating the embedded application by completely removing power to the FPGA and connected peripherals. The Intelligent Power Management Framework (IPMF) is a simple, flexible, light-weight library which integrates with Altera's NIOSII core and hardware abstraction layer (HAL). The IPMF provides support for managing power consumption on the FPGA dynamically with its application programming interface (API), which is simple to use in application development and is completely expandable with support for customizations via callback routines.

## 2. Summary

The IPMF is designed to seamlessly coexist with Altera's HAL, making it easy for application developers to immediately benefit from the framework's power savings.



**Figure 1.** Block diagram of NIOS system with IPMF.

The IPMF is responsible for saving and restoring the device's state during different operating modes. **Table 1** outlines the supported power modes within the IPMF.

Mode	Name	Description
Cold	Hibernate	Complete power-down of all peripherals, including the CPU/FPGA.
Warm	Sleep	Temporary suspension of all peripherals, but CPU remains powered.
Suspend	Partial Sleep	Systematically suspend power to individual peripherals.
Low	Reduced	Dynamically adjust system clock frequency

**Table 1.** IPMF power modes.

The framework provides routines necessary to manage all aspects of power management, including:

- Registering / unregistering devices
- Suspending / resuming system peripherals
- Saving / restoring peripheral registers
- Saving / restoring internal CPU registers
- Setting custom callback routines

The IPMF is a complete system solution consisting of the following core elements:

- Boot loader
- Library / Core
- Public API

Function	Description
pm_dev_reg	Register device with IPMF.
pm_dev_unreg	Unregister device with IPMF.
pm_cold_boot	Request IPMF to perform a cold boot or hibernate.
pm_warm_boot	Request IPMF to perform a warm boot or sleep.
pm_set_pause_fn	Register user-defined callback routine called by IPMF after all devices are suspended.
pm_enter_low_power_mode	Reduce system clock.
pm_leave_low_power_mode	Return system clock to normal operating frequency.
pm_suspend_device	Suspend a particular device.
pm_resume_device	Resume a particular device.

**Table 2.** Public IPMF functions exposed via the API.

### 3. Getting Started

When using the IPMF, several system design decisions must be made. One of the first decisions is where to store state information for powered down peripherals. Depending on the mode, items like CPU internal registers, peripheral control registers, and interrupt enabled registers need a place to reside that is not impacted by the loss or reduction of power.

Next, the engineer needs to decide which peripherals to register with the IPMF. Since the IPMF is designed for Altera's HAL, all devices registered with the IPMF utilize the standard HAL `alt_dev` datatype. Typical Altera devices, like UART, PIO, SPI, DMA, timers, etc., which are derived from `alt_dev`, are already supported by the IPMF. Devices not supported by the IPMF can easily be incorporated by utilizing Altera's HAL `alt_dev` data structure.

Finally, the application engineer makes a handful of function calls into the IPMF to support the desired system behavior and logic.

## 4. Application

The application programming interface (API) for the IPMF is simple and can be easily exercised with a few function calls (reference **Table 2**) and core data structures (reference **Listing 1** below).

```
typedef struct pm_dev
{
    alt_llist      llist;    // standard HAL linked list object
    alt_dev*      dev;      // pointer to standard HAL alt_dev
    pm_suspend_fn suspend; // function pointer to user supplied suspend routine
    pm_resume_fn  resume;  // function pointer to user supplied resume routine
} pm_dev;
```

**Listing 1.** Definition of `pm_dev`.

The first task the application engineer performs is device registration with the IPMF, which is accomplished by setting the `pm_dev` structure elements and calling the public routine `pm_dev_reg`. Altera's HAL provides an entry point before the main 'C' entry point (i.e., `alt_sys_init`) where devices can be registered with the IPMF, as shown in **Listing 2**.

```
ALTERA_AVALON_JTAG_UART_INIT( JTAG_UART_0, jtag_uart_0 );
static pm_dev pm_jtag_uart;
pm_jtag_uart.dev = (alt_dev*)&jtag_uart_0;
pm_jtag_uart.resume = pm_jtag_uart_resume;
pm_jtag_uart.suspend = NULL;
pm_dev_reg(&pm_jtag_uart);
```

**Listing 2.** Code snippet of JTAG UART device registration with IPMF in `alt_sys_init`.

While one system may hibernate by prompting the user via a button press or menu selection, another may do so after a certain period of time or some event (i.e., temperature greater than X). Since suspending or pausing a system can happen in a number of different ways, depending on the available hardware, the IPMF supports a custom callback mechanism that the developer can hook into the IPMF so that custom

system logic can be injected into the suspend operation of the IPMF. The application developer can setup a custom handler as follows (reference Listing 3):

```
// setup callback with IPMF
pm_set_pause_fn(hibernate_callback, NULL);

// User-supplied callback executed right before IPMF finishes
static int hibernate_callback(void* data, pm_event event)
{
    printf("Now it is safe to turn off the power\n");
    return 0;
}
```

**Listing 3.** User-supplied pause callback registration and with simple example.

The `pm_event` argument of the callback in **Listing 3** defines the type of power-off event (i.e., cold, warm, suspend, low). The developer can use this information to perform final system maintenance, such as change clock settings, suspend individual peripherals via `pm_suspend_device`, or put the system in sleep mode (i.e., `pm_warm_boot`). The IPMF handles the case for hibernate (i.e., `pm_cold_boot`) a little differently, as it also attempts to save memory.

The IPMF supports both RAM and flash for storing state information during hibernate. When using flash, the application developer specifies the flash device name, region, and block index where the state information should be saved, while RAM only requires a known physical location in the address space. **Listing 4** shows code samples for hibernating to both flash and RAM.

```
// 1. hibernate set to use Flash storage
pm_storage storage;

storage.type = PM_FLASH;
storage.flash.name = EXT_FLASH_NAME;
storage.flash.region = 0;
storage.flash.block = 0;
pm_cold_boot(&storage);

// 2. hibernate set to use RAM storage
pm_storage storage;

// suspend the device
storage.type = PM_RAM;
storage.ram.ptr = (void*)SYSTEM_MEMORY_BASE;
pm_cold_boot(&storage);
```

**Listing 4.** Code examples for hibernating to RAM and flash.

## 5. Device Driver

Besides registering the device, as shown above in `alt_sys_init` with the `pm_dev_reg` function, the device driver developer only needs to provide callback routines for suspend and resume if required. All callbacks are optional and usually the resume callback is required. The best place to register for the resume and suspend callbacks would be in the driver's initialization routine; however, since one of the goals

of the IPMF was to not modify any Altera HAL source, the registration is manually done in `alt_sys_init`, as shown in **Listing 2**.

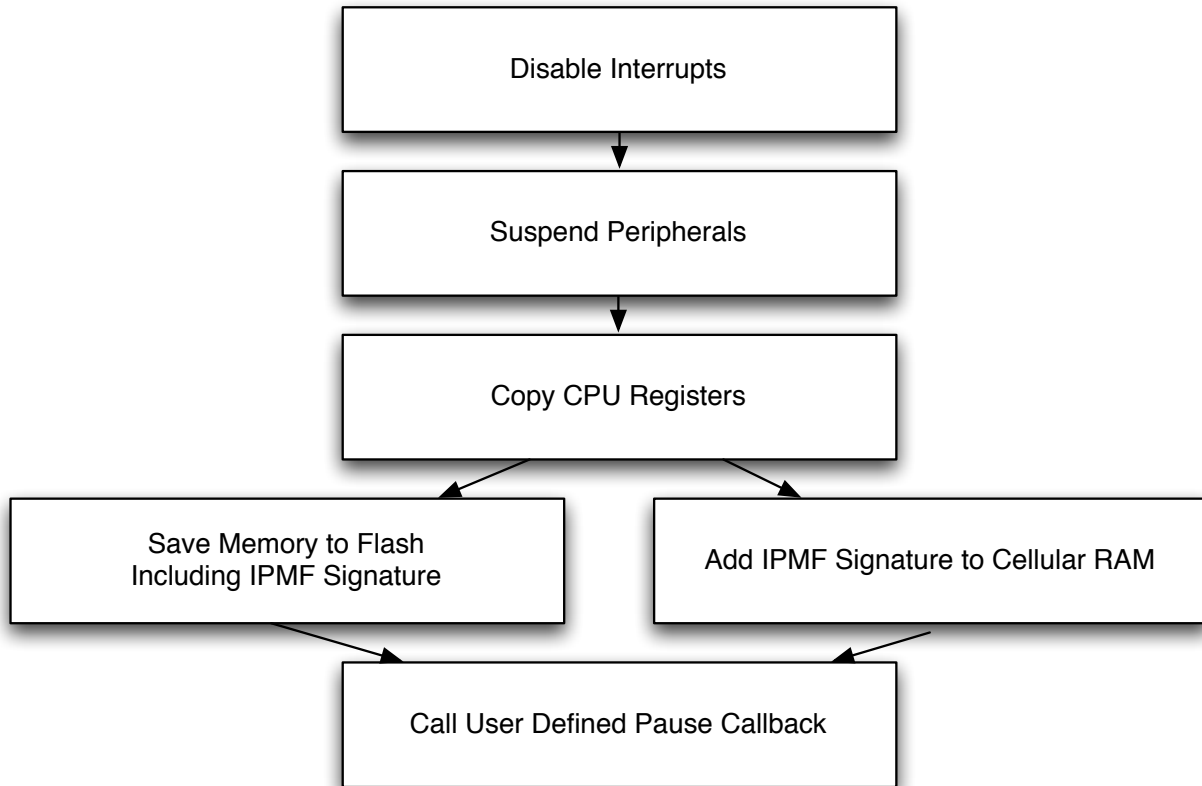
Since the only two callbacks required for a device driver are suspend and resume, the process of porting either an existing Altera HAL component or a custom peripheral to the IPMF should not involve too much effort. Most suspend or resume callbacks only need to store/restore control or enable type registers, which only require a few lines of code. **Listing 5** shows the resume callback for the JTAG UART which uses the `alt_dev` structure to restore the UARTs control register.

```
int pm_jtag_uart_resume(alt_dev* dev, pm_event event)
{
    altera_avalon_jtag_uart_dev* dev2 = (altera_avalon_jtag_uart_dev*)dev;
    unsigned base = dev2->state.base;
    IOWR_ALTERA_AVALON_JTAG_UART_CONTROL(base, dev2->state.irq_enable);
    return 0;
}
```

**Listing 5.** Example resume callback for the JTAG UART.

## 6. System / Core

The IPMF core is primarily concerned with facilitating the registered user callback routines for suspend and resume of peripherals and managing the storage and loading of state information within flash or RAM. After the call to the routine `pm_cold_boot`, the IPMF iterates through its list of registered peripherals and either automatically saves their state data into the specified storage location, if using a supported Altera component, or invokes a user-supplied 'suspend' callback that will save off data according to the provided suspend routine. The IPMF supports the concept of priorities, so the developer can control the order in which peripherals are "taken down" to prevent potential conflicts. The NIOS core is one of the last peripherals processed by the IPMF saving off its registers and program counter. To handle any system failures during the suspend operation (i.e., certain peripherals could not be suspended, insufficient flash space, etc.), the IPMF supports a rollback mechanism which restores all peripherals in reverse order, leaving the system in a working state. Reference **Figure 2** for the general sequence implemented by the IPMF during the suspend operation.

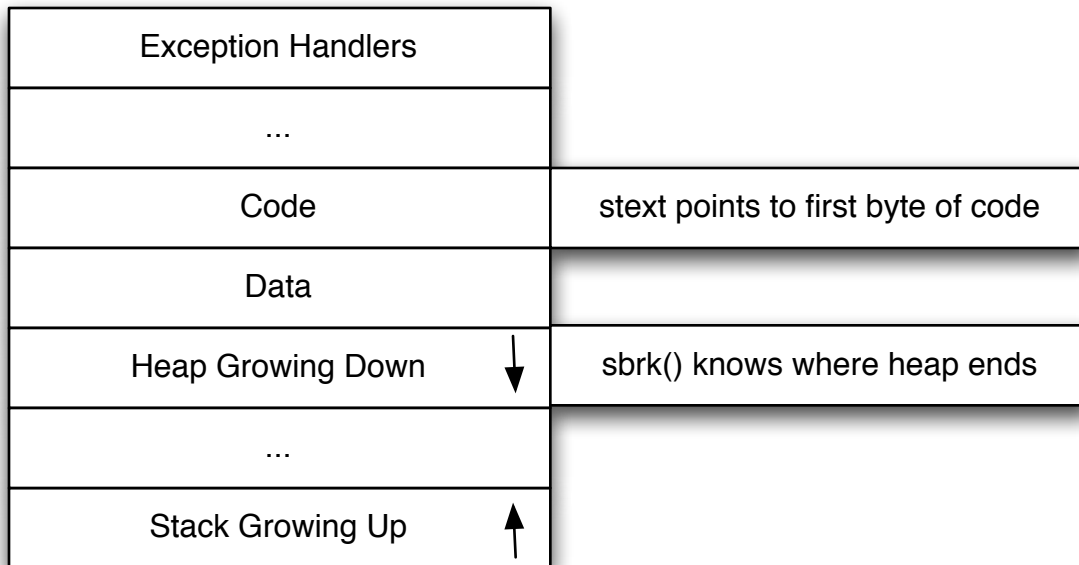


**Figure 2.** Cold boot / hibernate suspend sequence.

When the device's state is stored to flash, the framework takes care of saving off both CPU registers and regions of RAM occupied by the application, such as:

- code
- global variables
- heap
- stack
- exception handlers

The sections listed above could reside in any available memory defined by the linker script; however, typical applications use the same memory region where all sections are organized as shown in **Figure 3**.



**Figure 3.** Typical organization of code sections.

The sections in **Figure 3** can be defined by the following array in **Listing 6**:

```
const void* mem_regions[][2] =
{
  { &_amp_ram_exceptions_start, &_amp_ram_exceptions_end },
  { &stext, sbrk(0) },
  { alt_stack_pointer(), &_amp_alt_stack_pointer }
};
```

**Listing 6.** C array to house image sections.

The array definition above is used to enumerate all sections and to store the structured data to flash. Since this structure is identical to the format used by the IPMF boot loader, it's possible to load the application code back into flash and resume operation from the last execution state. The use of a special IPMF signature, stored in the first few bytes of flash by the suspend operation, makes it possible to determine if the device should boot normally or resume from a previous hibernation.

The code snippet in **Listing 7** shows how the IPMF boot loader utilizes the special signature to either perform a normal boot after copying the image into RAM (i.e., crt0.S) or jump into the IPMF's restore point after the image is reloaded.

```

// r_signature = PM_SIGNATURE
movhi r_signature, %hi(PM_SIGNATURE)
ori r_signature, r_signature, %lo(PM_SIGNATURE)

// r_flash_ptr = PM_SIGNATURE_OFFSET
movhi r_flash_ptr, %hi(PM_SIGNATURE_OFFSET)
ori r_flash_ptr, r_flash_ptr, %lo(PM_SIGNATURE_OFFSET)

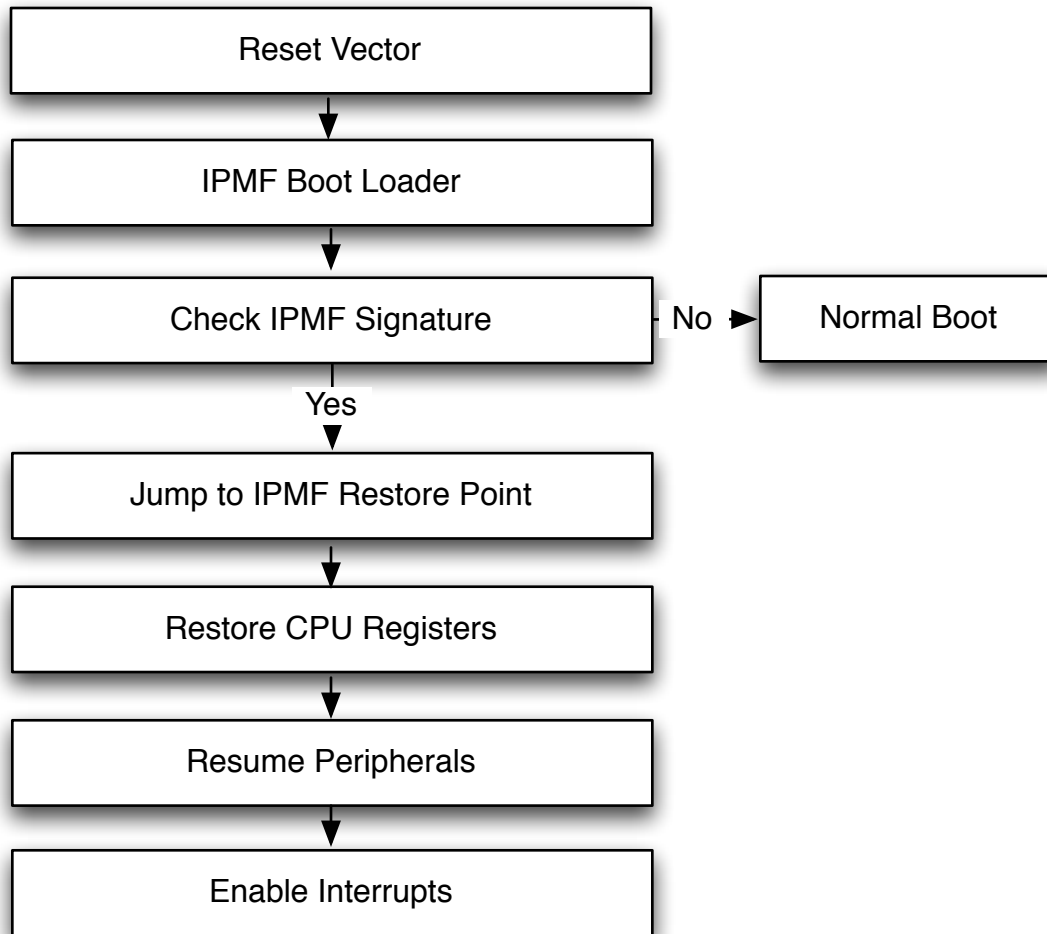
// r_flash_ptr = PM_FLASH_SIGNATURE_OFFSET
movhi r_flash_ptr, %hi(PM_FLASH_SIGNATURE_OFFSET)
ori r_flash_ptr, r_flash_ptr, %lo(PM_FLASH_SIGNATURE_OFFSET)

// rf_temp = READ_INT(r_flash_ptr++)
nextpc return_address_less_4
br READ_INT
mov rf_temp, r_read_int_return_value
beq r_signature, rf_temp, copy_job

```

**Listing 7.** Code snippet of IPMF boot loader.

If it is determined that the device was previously hibernated, a jump into the IPMF framework is made which restores / resumes all registered devices (**Figure 4**). Once all restoration is completed, control is returned to the caller of the `pm_cold_boot` routine, and the application continues normal execution with all registers and context restored to exactly the same state as before the hibernation. Once all sections are restored from flash, the IPMF invalidates the flash block where the IPMF special signature is located so that it will no longer not be recognized by the boot loader anymore.



**Figure 4.** Cold boot / hibernate resume sequence.

## 7. Conclusion

The framework was used in Arrow's Low Power Reference Platform (LPRP) and demonstrated a considerable power savings as shown in **Table 3**.

Mode	Freq.	Power	Recovery Time	Battery Life
Normal	80 MHz	613 mW	NA	13 hours
Reduced	65 MHz	492 mW	< 100 ns	16 hours
Hibernate	0 Hz	7.5 mW	20 ms	months
Off	NA	< 500 $\mu$ W	< 70 ms	forever

**Table 3.** Comparison of different power modes.

The IPMF is a light-weight library which integrates with Altera's Nios core / HAL and offers the ability to manage power consumption on an FPGA dynamically. The framework's API is simple to use in application development and is highly expandable.