Implementing S0ix (Modern Standby) Support in FreeBSD

Aymeric Wibo

The FreeBSD Project
obiwac@FreeBSD.org

Abstract

Modern laptops have largely abandoned the legacy ACPI S3 sleep state in favor of low-power idle states collectively known as S0ix. As a result, operating systems that rely on S3 for suspend-to-RAM functionality are increasingly unable to provide effective sleep support on contemporary hardware.

This paper presents ongoing work to implement suspend-to-idle (s2idle) and S0ix (specifically S0i3) support in FreeBSD. We describe the architectural differences between S3 and S0ix, the necessary kernel infrastructure for suspend-to-idle, platform coordination via ACPI System Power Management Controllers, and vendor-specific requirements, with a focus on AMD platforms.

1 Introduction

A significant barrier to using FreeBSD on modern laptops is the inability to suspend the system. While legacy systems utilized the ACPI S3 state, modern hardware increasingly relies on S0ix. S3 is a global sleep state defined by ACPI. In contrast, S0ix keeps the system in the global S0 (working) state, relying on firmware to enter low-power states only when CPUs are idle and device power constraints are met. The target for maximum power savings is the S0i3 state.

1.1 Prior Art

Early efforts toward suspend-to-idle and S0ix support in FreeBSD date back to 2018, when Ben Widawsky (Intel) proposed initial implementations for these mechanisms. The first patch introduced basic suspend-to-idle infrastructure [1], while a follow-up revision attempted to emulate the legacy ACPI S3 sleep state on top of S0ix-capable platforms [2].

Although these revisions established important groundwork, neither was completed nor integrated into the FreeBSD source tree, and significant work still remained, motivating the effort described in this paper.

1.2 Hardware Identification

Identifying whether a machine uses S3 or S0ix is the first step in enablement. In FreeBSD, supported sleep states are queried via the `hw.acpi.supported_sleep_state` sysctl. The absence of `S3` usually indicates a reliance on S0ix. Definitive confirmation requires checking the FADT flags, specifically `(AcpiGbl_FADT.Flags & ACPI_FADT_LOW_POWER_S0)`.

2 Suspend-to-idle (s2idle)

Before the platform can be instructed to enter the S0i3 low-power state, the operating system must first provide a mechanism for quiescing execution while remaining within the S0 (working) power state. Suspend-to-idle (s2idle) is this mechanism, allowing the system to reach a fully idle configuration without invoking a firmware-managed sleep transition.

At a high level, s2idle closely resembles existing ACPI sleep paths, and much of the required infrastructure is already present in the `acpi_EnterSleepState()` implementation[1]. The procedure largely mirrors that of traditional sleep states and consists of the following steps:

- Binding the current thread to the bootstrap processor (BSP) using `sched_bind()`.
- Suspending userspace execution and mounted filesystems via `stop_all_proc()` and `suspend_all_fs()`.
- Suspending the device hierarchy by invoking `DEVICE_SUSPEND(root_bus)`.
- Halting scheduler clock interrupts on all CPUs using `suspendclock()`. Although invoked on the BSP, this operation is propagated to all other CPUs through inter-processor interrupts (IPIs).

[1] In the future, we will want a lot of this infrastructure to be divorced from ACPI in order to support s2idle (and perhaps even hibernate) on non-ACPI platforms.

- Saving the interrupt flag (IF) and disabling maskable interrupts using `intr_disable()` (i.e. `cli` on x86).

Resumption from s2idle is achieved by executing essentially the same sequence of operations, inverted, and in reverse order. This initial s2idle framework was implemented in D48734 [3].

Once the system has been prepared for s2idle, it is necessary to ensure that all non-BSP CPUs (a.k.a. *application processors*, or APs) cease normal scheduling activity and remain idle for the duration of the suspend interval.

In the current prototype, this is accomplished via explicit coordination with the scheduler by introducing a `tdq_do_idle` flag within `struct tdq`, which represents per-CPU scheduler state. When entering s2idle, this flag is set on all CPUs, causing the scheduler's thread selection routine, `tdq_choose()`, to unconditionally return the idle thread. The flag is cleared during resume. This mechanism is implemented in D54407 [4].

While effective, this approach is heavy-handed and bypasses normal scheduler behaviour. A more refined solution would allow the scheduler to converge naturally to the idle thread, for example through the introduction of a high-priority idle task. Nevertheless, the current approach is sufficient to enable early experimentation and validation.

The APs are transitioned into this forced idle state using an SMP rendezvous:

```
static void
set_cpu_idle(void *data)
{
    bool idle = *(bool*) data;

    sched_do_idle(curthread, idle);
}

other_cpus = all_cpus;
CPU_CLR(curcpu, &other_cpus); // curcpu is
    the BSP, as it was previously bound.

bool idle = true;
smp_rendezvous_cpus(other_cpus, NULL,
    set_cpu_idle, NULL, &idle);
```

Since the BSP is what's actually running our sleep entry/s2idle routine, all we have to do is explicitly enter the idle state:

```
cpu_idle(0);
```

The argument to `cpu_idle()` specifies the `busy` flag. When set, this flag indicates that the scheduler considers the CPU to be under load, prompting the processor to select a shallower idle state (typically C1) with reduced exit latency. For s2idle, the flag is unset, allowing deeper architecture-specific idle states to be selected. CPU idle states (C-states) and their role in platform power management are discussed in a later section.

2.1 Interrupts and GPEs

Entering the idle loop on the BSP using `cpu_idle()` causes execution to resume upon receipt of any interrupt. While a variety of interrupt sources may be active, only a restricted subset should be permitted to wake the system from s2idle, such as user-initiated wake events (e.g. power button presses or lid open events).

When the platform needs to notify the operating system power management (OSPM) layer of an ACPI-related event, it delivers an ACPI System Control Interrupt (SCI) to the BSP. The SCI interrupt vector is specified by `AcpiGbl_FADT.SciInterrupt` and is commonly mapped to interrupt 9 on x86 systems. To ensure that only SCIs can cause the BSP to exit its idle state, all other interrupt sources are masked prior to entering `cpu_idle()`.

The resulting interrupt masking sequence is shown below:

```
register_t rflags = intr_disable(); // Save
    previous IF, disable interrupts.
intr_suspend();                     // Mask
    all PIC interrupt sources.
intr_enable_src(AcpiGbl_FADT.SciInterrupt);
    // Enable SCI (typically IRQ 9).

cpu_idle(0); // Enter BSP idle state.

intr_resume(false); // Re-enable PIC
    interrupt sources.
intr_restore(rflags); // Restore interrupt
    flag.
```

Because `acpi_EnterSleepState()` binds the executing thread to the BSP and because SCIs are delivered exclusively to the BSP, any SCI received during s2idle is guaranteed to cause the BSP to exit `cpu_idle()` rather than waking application processors (APs).

However, not all SCIs correspond to wake-worthy events. After receiving an SCI, OSPM must determine the underlying cause by inspecting the ACPI General Purpose Event (GPE) registers. Each SCI is associated with one or more GPEs, which represent device- or platform-specific events.

Importantly, not all GPEs should result in a system wake. For example, some embedded controllers generate periodic GPEs to report battery status updates. Waking the entire system in response to such events would be undesirable.

ACPI provides a mechanism for masking device wake capability through the `_DSW` method (or `_PSW` for legacy devices), as specified in the ACPI standard [5]. In practice, however, many platforms multiplex wake-relevant and noisy events onto the same GPE number, preventing selective masking.

Listing 1 shows a simplified ASL excerpt from a Framework AMD 7040 series laptop in which both lid events and battery status updates share the same GPE:

```
Device (EC0) { // Embedded controller.
    Name (_GPE, 0x0B)
    Device (LID0) { /* ... */ }

    Method (_Q01, 0, NotSerialized) { //
    Lid event.
        P80H = 0x01
        Notify (LID0, 0x80)
    }

    Method (_Q3C, 0, NotSerialized) { //
    Battery status update (high frequency).
        P80H = 0x3C
        Notify (BAT1, 0x80)
    }
}
Device (BAT1) { /* ... */ }
```

Listing 1: Simplified ASL illustrating shared GPEs

In this configuration, masking battery-related GPEs would also suppress lid open events, rendering the system unable to wake in response to legitimate user interaction. As a result, the BSP must periodically exit its idle state, inspect the GPE that triggered the SCI, and determine whether the event warrants a full system resume.

This behavior is implemented using an explicit s2idle polling loop [6]:

```
wake_event = false;

while (!wake_event) {
    cpu_idle(0); // Enter BSP idle state.
    taskqueue_quiesce(acpi_taskq); //
    Process GPEs; sets wake_event if
    appropriate.
}
```

It is critical that the ACPI taskqueue be bound exclusively to the BSP. During s2idle, the schedulers on all APs are forced into idle loops and are therefore unable to execute queued tasks [7].

As noted previously, APs are not expected to exit their idle states during s2idle, as SCIs are delivered solely to the BSP. Even if an AP were to receive an interrupt, the idle thread contains its own loop that immediately re-enters the idle state [8].

This design is not ideal, as repeatedly exiting and re-entering idle states incurs non-trivial overhead, particularly when deeper CPU sleep states are involved. Nevertheless, SPMC hints significantly reduce the frequency of spurious wakeups. On the evaluated system, battery-related GPEs are reduced from approximately once per second to once per minute during suspend.

This behavior can be observed in the embedded controller firmware, shown in simplified form below [9]:

```
if (!curr.ac && (curr.state == ST_IDLE ||
    curr.state == ST_DISCHARGE)) {
    if (chipset_in_state(
    CHIPSET_STATE_ANY_OFF |
    CHIPSET_STATE_ANY_SUSPEND) && curr.
    output_current == 0)
        sleep_usec =
    CHARGE_POLL_PERIOD_VERY_LONG; // 1 min
    else
        sleep_usec =
    CHARGE_POLL_PERIOD_LONG; // 500 ms
} else {
    sleep_usec = CHARGE_POLL_PERIOD_CHARGE;
     // 250 ms
}
```

Although the impact of this behavior on overall power consumption has not yet been quantitatively profiled, the reduced wake frequency suggests that it is limited, particularly given that only a single CPU is involved.

As future work, exposing the event responsible for exiting the s2idle loop via a `sysctl` interface would aid in diagnosing unexpected wakeups and improve system observability.

3 Platform Coordination for S0ix Entry

Suspend-to-idle is, by design, a purely operating-system–managed mechanism. From the platform's perspective, the system remains in the S0 (working) power state, and the CPUs merely happen to be idle. As a result, platform components such as the embedded controller (EC) are unable to distinguish between normal idle behavior and a deliberate attempt to enter a low-power idle state.

Consequently, the operating system must explicitly notify the platform when entering and exiting an S0ix state. On Framework laptops, successful coordination is observable through platform-controlled indicators, such as the power button LED slowly fading in and out during suspend.

3.1 System Power Management Controller (SPMC)

Platform notification is performed through the System Power Management Controller (SPMC), also referred to as the Power Engine Plug-in (PEP). The SPMC is identified via the ACPI hardware ID `PNP0D80` ("Windows-compatible System Power Management Controller").

To support this functionality on FreeBSD, a new `acpi_spmc` driver was implemented in D48387 [10].

The SPMC serves two primary purposes:

- It provides device power constraints (D-states) required for entry into specific low-power idle states.
- It allows the operating system to notify the platform of suspend and resume checkpoints, such as display power transitions and entry into low-power idle.

Both functions are implemented through ACPI Device Specific Methods (DSMs).

3.2 Device Specific Methods (DSMs)

In ACPI terminology, a DSM is a multiplexed method exposed via the `_DSM` object that enables vendor-specific functionality. A DSM invocation consists of four arguments: a vendor UUID, a revision number, a function index, and an optional argument package. On FreeBSD, DSM evaluation is performed using `acpi_EvaluateDSMTyped()`.

Although an early Intel specification for Low Power S0 Idle defined a DSM interface [11], this interface (UUID `c4eb40a0-6cd2-11e2-bcfd-0800200c9a66`) does not appear to be implemented on AMD platforms, and isn't sufficient on contemporary Intel platforms either. Instead, AMD systems rely on vendor-specific DSMs, defined by Microsoft and AMD.

3.2.1 Microsoft Modern Standby DSM

Microsoft defines a DSM interface for Modern Standby firmware notifications [12], identified by UUID `11e00d56-ce64-47ce-837b-1f898f9aa461`. This interface closely resembles the original Intel design, with additional Modern Standby–specific functions and the omission of others.

Table 1 summarizes the Microsoft DSM function indices.

Table 1: Microsoft/Intel Modern Standby DSM functions

Index	Description	Notes
0	Enumerate functions	
1	Get device constraints	Intel spec only
2	Get crash dump device	Intel spec only
3	Display off notification	
4	Display on notification	
5	Entry notification	
6	Exit notification	
7	Modern Standby entry notification	
8	Modern Standby exit notification	

3.2.2 AMD SPMC DSM

AMD platforms expose an additional DSM interface identified by UUID `e3f32452-febc-43ce-9039-932122d37721`. Public documentation for this interface is scarce, and its behavior is primarily inferred from the existing Linux implementation [13].

The AMD DSM functions are summarized in Table 2.

Table 2: AMD SPMC DSM functions

Index	Description	Notes
0	Enumerate functions	
1	Get device constraints	
2	Entry notification	Fades power button LED
3	Exit notification	
4	Display off notification	
5	Display on notification	

A simplified pseudo-code example illustrating invocation of the AMD *Get Device Constraints* function is shown below:

```
Arg0 = "e3f32452-febc-43ce-9039-932122
    d37721"; // AMD SPMC DSM UUID
Arg1 = 0; // Revision
Arg2 = 1; // Get device constraints
Arg3 = Package(); // No arguments
call_dsm(spmc_device, Arg0, Arg1, Arg2,
    Arg3);
```

On AMD platforms, the AMD DSM UUID must be used to retrieve device power constraints, as the Microsoft interface does not provide this functionality. Contrary to the equivalent Intel DSM, AMD's returned device constraint package follows a vendor-specific format for which no public specification could be identified.

In contrast, platform notifications—including both legacy and Modern Standby entry and exit notifications—appear to require evaluation of DSMs from both the Microsoft and AMD UUID sets. Furthermore, the ordering of these notifications may be significant, as suggested by the prior Linux implementation [14].

The precise behavior of modern Intel platforms with respect to SPMC DSMs requires further investigation for FreeBSD.

4 Achieving S0i3

While suspend-to-idle provides a functional suspend and resume mechanism, it does not by itself result in meaningful power savings. At this stage, the system remains fully within the S0 power state, with CPUs idling and the platform notified of suspend entry, but without transitioning into a deeper low-power configuration. As a result, overall power consumption remains comparable to that of an idle but otherwise fully operational system.

Substantial power savings require entry into the S0i3 low-power idle state. Reaching S0i3 necessitates coordinated action across the operating system, device drivers, CPU power management, and platform firmware. Moreover, unlike traditional sleep states, entry into S0i3 is not explicitly

commanded by the OS; rather, it it initiated by firmware once all hardware and software preconditions are satisfied.

4.1 Debugging S0i3 Entry: Residency Counters and Platform Telemetry

The first challenge in adding S0i3 support is observability, i.e. making sure we've entered it in the first place.

Determining whether the platform has successfully entered the S0i3 low-power idle state is nontrivial, as no externally visible indicator reliably distinguishes S0i3 from a shallow idle configuration. Instead, verification relies on *residency counters*, which report the cumulative time spent by the system or individual CPUs in specific low-power states.

Early ACPI-based implementations expose residency information via the Low Power Idle Table (LPIT), originally defined in Intel's Low Power S0 Idle specification [11]. The LPIT describes supported low-power idle states and may include per-state residency counters.

Beginning with ACPI 6.0, however, the LPIT mechanism has fallen out of favor, particularly on non-x86 platforms [15]. Instead, ACPI introduces the `_LPI` object, which describes low-power idle states and optionally exposes residency information [16]. Notably, support for residency counters within `_LPI` objects is optional.

In practice, platform implementations are inconsistent. Recent Intel systems frequently expose LPIT tables but omit `_LPI` objects, whereas contemporary AMD laptops typically provide `_LPI` objects without an accompanying LPIT. Consequently, both mechanisms must be supported. Unfortunately, on the evaluated AMD Framework laptop, residency counters are absent from the `_LPI` objects, rendering ACPI-based residency tracking unavailable.

4.1.1 AMD System Management Unit (SMU)

On AMD platforms, residency information can instead be obtained from the System Management Unit (SMU), also referred to as MP1. The SMU is an on-die microcontroller responsible for power management decisions, including final arbitration of S0i3 entry. Internally, it executes proprietary power management firmware (PMFW) and operates independently of the main CPU cores.

Initial FreeBSD support for interfacing with the AMD SMU was introduced via the `amdsmu` driver [17]. Subsequent work exposed SMU-provided residency counters through `sysctl` interfaces [18]. To correctly measure S0i3 residency, the SMU must be explicitly notified of S0i3 entry and exit boundaries, as it reports residency accumulated between these two points.

The internal structure of modern AMD processors, which contain multiple embedded "Matroshka" microcontrollers operating alongside the main cores, has been explored in prior work [19].

4.1.2 Practical Debugging Support

On Linux, the `amd_s2idle.py` script [20] provides a comprehensive diagnostic tool for identifying reasons why a system fails to enter S0i3, including firmware-level vetoes and some more granular common vendor-specific blockers. An equivalent tool does not yet exist on FreeBSD, though developing similar instrumentation would significantly improve debuggability and user-facing diagnostics.

Overall, reliable S0i3 debugging requires combining ACPI-provided information, platform-specific telemetry, and firmware cooperation. The absence of standardized residency reporting across vendors remains a significant challenge.

4.2 Putting Devices to Sleep

As outlined previously in the suspend-to-idle flow, all devices must be transitioned into appropriate low-power states before the platform firmware will permit entry into the S0i3 state.

At a minimum, entry into any Low Power Idle (LPI) state requires that the device power constraints communicated by the System Power Management Controller (SPMC) are satisfied. In practice, however, once a system-wide suspend is initiated, it is desirable to minimize overall power consumption by transitioning all devices into their deepest supported low-power states. On FreeBSD, this is initiated via a recursive suspension of the device tree using `DEVICE_SUSPEND(root_bus)`.

ACPI defines a set of device power states, known as D-states, which describe progressively lower-power operating modes. These include D0 (fully operational), D1 and D2 (intermediate low-power states), and two variants of D3: D3hot and D3cold. While D3hot represents a powered-off device that retains auxiliary power, D3cold corresponds to a state in which all power is removed. The distinction between D3hot and D3cold was introduced relatively late in the ACPI specification, and historical references to "D3" are often ambiguous with respect to which variant is intended.

Recent work has clarified this distinction within ACPICA, following discussion in an upstream pull request [21].

On FreeBSD, transitions between device power states are mediated by the `acpi_pwr_switch_consumer()` function. To

transition a device into a target D-state, the operating system must first evaluate the corresponding power resource object _PRx, where x denotes the desired D-state. All power resources listed in _PRx must be enabled, while power resources associated with higher-power states (i.e., lower-numbered D-states) must be disabled. Once the power resources are configured, the device power state is changed by evaluating the _PSx control method, as specified in the ACPI standard [22].

A device supports the D3cold state only if it explicitly exposes a _PR3 object. In this case, keeping the _PR3 power resources enabled results in a D3hot transition, whereas disabling them fully removes power and places the device into D3cold.

An initial attempt to correct the transition of device power states in FreeBSD was implemented in revision [23]. However, this change revealed latent issues in the existing D-state handling logic and caused regressions on systems relying on legacy S3 suspend. As a result, the patch was reverted, and further work is required to reconcile proper D-state transition existing suspend mechanisms.

4.3 Checking for Device Power Constraint Violations

To verify that no device violates its SPMC-communicated power constraints, the OS must first determine the current D-state of each ACPI power consumer. Support for querying a device's effective power state was added to FreeBSD via the `acpi_pwr_get_consumer()` function [24].

ACPI defines multiple mechanisms for determining a device's current power state. The most direct method is evaluation of the _PSC (Power State Current) control method, which returns the device's active D-state when present [25]. However, _PSC is optional and is not implemented by all devices, and the ACPI specification explicitly permits omission of _PSC when the device state can be inferred from its power resources.

In the absence of _PSC, the device's D-state can be inferred by inspecting the power resource objects associated with each D-state. These power resources are described by the _PRx objects, where x denotes the target D-state. The effective D-state is determined using the following procedure:

1. Iterate over the supported D-states in ascending order, starting from D0.

2. For each D-state, evaluate the corresponding _PRx object.

3. For each power resource returned, evaluate its _STA method to determine whether the resource is enabled.

4. If all power resources for a given D-state are enabled, the device is inferred to be in that D-state.

5. If no D-state satisfies this condition, and all power resources associated with _PR3 are disabled, the device is inferred to be in the D3cold state.

Once the current D-state of a device has been determined, validating compliance with SPMC constraints is straightforward. For each device, the inferred D-state must be numerically greater than or equal to the minimum D-state specified in the corresponding device power constraint package. Any device found to violate this condition may prevent successful entry into the S0i3 state and must be addressed before suspend can proceed.

4.4 CPU Idling and the _CST Object

For successful entry into the S0i3 state, it is essential that all CPUs reside in their deepest available C-state for the duration of suspend. On contemporary x86 platforms, this is typically the C3 state, though the exact set of supported C-states is platform-dependent.

As discussed previously, entry into suspend-to-idle results in calls to the `cpu_idle()` routine. This occurs both on the bootstrap processor (BSP), where idling is performed explicitly by the suspend thread, and on application processors (APs), which are forced into their idle threads for the duration of s2idle. When the scheduler indicates that the CPU is not busy (i.e., `busy == 0`), `cpu_idle()` may enter a deeper idle state.

On ACPI-based systems, this ultimately results in a call to `acpi_cpu_idle()`, which is implemented by the `acpi_cpu` driver. This driver determines the deepest supported C-state and its associated entry method by parsing the _CST object exposed by firmware.

A representative _CST object on an AMD-based laptop is shown below. It is parsed in FreeBSD by `acpi_cpu_cx_cst()`.

```
Name (_CST Package (0x04) {
    0x03, // Number of C-states
    Package (0x04) { // C1
        ResourceTemplate () {
            Register (FFixedHW, 0x02, 0x02,
     0x0000000000000000)
        },
        0x01,         // C-state type: C1
        0x0001,       // Entry/exit latency
    (us)
        0x00000000,   // Power consumption (
    mW)
    },
    // ...
    Package (0x04) { // C3
        ResourceTemplate () {
```

```
            Register (SystemIO, 0x08, 0x00,
     0x0000000000000414, 0x01)
        },
        0x03,         // C-state type: C3
        0x015E,       // Entry/exit latency
    (us)
        0x00000000,   // Power consumption (
    mW)
    }
})
```

Listing 2: Example _CST object

In this example, the shallowest supported C-state is C1, while the deepest supported C-state is C3. Each C-state entry in the _CST package specifies an entry method via a ResourceTemplate, the C-state type, its entry/exit latency, and an estimated power consumption.

On platforms where the _CST object is implemented as a complex method rather than a static package, the parsed results can be inspected at runtime via the dev.cpu.0.cx_method sysctl, with additional C-state information available under the dev.cpu.0 sysctl subtree.

4.4.1 MWAIT Entry Method

In the example above, the entry method for the C1 state is specified as FFixedHW. This causes FreeBSD to follow the acpi_PkgFFH_IntelCpu() path, where the bit offset field of the register descriptor is interpreted as a C-state class identifier.

In this case, the class corresponds to CST_FFH_INTEL_CL_MWAIT, indicating that the processor should enter the specified C-state using the x86 MWAIT instruction [26]. (An alternative class, CST_FFH_INTEL_CL_C1IO, indicates a legacy C1 entry path using an I/O instruction followed by HLT.)

The MWAIT instruction is typically used in conjunction with MONITOR to place the processor into an implementation-defined optimized state until a write occurs to a monitored memory range. However, when CPUID indicates support for interrupt break events, MWAIT can be used independently of MONITOR.

Specifically, in that case, setting bit 0 of ECX prior to executing MWAIT allows interrupts (including SCIs) to break out of MWAIT and thus wake the processor. Furthermore, if CPUID indicates support for power-managed MWAIT, bits 7–4 of EAX can encode the desired C-state, while the lower four bits select sub-states.

On the system shown above, entry into the C1 state using MWAIT would be performed as follows:

```
mov eax, 0x00000000 ; C1 hint (MWAIT_C1)
mov ecx, 1          ; Break on interrupt (
    MWAIT_INTRBREAK)
mwait
```

Listing 3: MWAIT entry into C1

The C-state hint placed in EAX is derived from the address field returned by acpi_PkgFFH_IntelCpu(), as implemented in acpi_cpu_cx_cst_mwait().

On Intel platforms, it is common for deeper C-states, such as C3, to use the same MWAIT-based entry mechanism, with a different hint value:

```
mov eax, 0x00000020 ; C3 hint (MWAIT_C3)
mov ecx, 1          ; Break on interrupt (
    MWAIT_INTRBREAK)
mwait
```

Listing 4: MWAIT entry into C3

4.4.2 I/O Entry Method

Returning to the _CST object shown previously, the entry method for the deepest supported C-state (C3 in this example) is specified as SystemIO. This indicates that the processor should enter the target C-state by performing an I/O read from a platform-defined register.

During _CST parsing, this register is extracted from the resource descriptor and stored in the p_lvlx field of the corresponding struct acpi_cx instance. Entering the C-state then reduces to issuing a read from this resource:

```
struct acpi_cx *cx_next = /* C3 C-state
    descriptor */;
CPU_GET_REG(cx_next->p_lvlx, 1);
```

The argument 1 denotes a single-byte read and ultimately resolves to a bus_space_read_1() operation. As with the interrupt-breakable MWAIT entry method, the processor remains in this C-state until an external interrupt—such as an ACPI System Control Interrupt (SCI)—causes it to resume execution.

Historically, these I/O-based C-state entry registers were described directly in the Fixed ACPI Description Table (FADT) as the P_LVL2 and P_LVL3 registers. Prior to the introduction of the _CST object, operating systems interacted with these registers explicitly. This legacy probing path remains visible in the FreeBSD implementation for backwards compatibility.

4.4.3 The _LPI Object Revisited

While previously discussed in the context of residency counters, the _LPI object serves a broader purpose. It is intended as a functional superset of the _CST object, providing a unified description of hierarchical processor idle states, entry mechanisms, and optional residency counting [16]. As such, it is expected to eventually supersede _CST on contemporary platforms.

At present, FreeBSD does not implement support for _LPI-based processor idle state management.

5 Vendor-Specific Considerations: AMD

On AMD platforms, additional platform-specific conditions must be satisfied before the PMFW running on the SMU will permit entry into the S0i3 state. Many of these conditions can be diagnosed on Linux systems using the `amd_s2idle.py` debugging script [20].

At present, the following requirements must be met on AMD-based systems:

- The operating system must explicitly notify the SMU when entering and exiting low-power idle. [27].
- The USB4 host connection manager (HCM) must place the USB4 controller into a low-power state [28]. The controller is initially put into a high-power configuration by the pre-boot connection manager, preventing S0i3 entry if we do not explicitly do this.
- On some systems, all GPIO interrupts must be serviced prior to suspend. Preliminary AMD GPIO support is provided by the `amdgpio` driver [29], with interrupt servicing later added [30]. A suspend routine for this driver is under development [31].
- The `amdgpu` Direct Rendering Manager (DRM) driver must be loaded and correctly transitioned into its low-power state. This required a change to the LinuxKPI infrastructure to distinguish between S3 and S0ix suspend paths [32].

If any of these conditions are not satisfied, the PMFW will refuse to transition the system into S0i3. To aid in figuring out what exactly would block S0i3 entry on AMD with only access to incomplete documentation, a minimal Linux kernel configuration was constructed starting from `make tinyconfig`, only iteratively enabling components until successful S0i3 entry was acquired [33].

6 Hibernation (S4)

Hibernation (ACPI S4) is largely orthogonal to the S0ix power model. Rather than maintaining system state in RAM, hibernation writes the contents of memory to persistent storage and powers the system off entirely, making it more closely related to the S5 state.

While S4 offers substantially lower power consumption than S0i3—effectively eliminating battery drain—the trade-off is significantly increased suspend and resume latency. To balance responsiveness and energy efficiency, operating systems commonly employ a hybrid strategy: the system initially enters a low-power idle state such as S0i3, and a real-time clock (RTC) alarm is programmed to wake the system after a fixed interval to initiate hibernation. This approach allows for fast wake-up during short suspend intervals while still guaranteeing minimal power consumption during extended inactivity.

More advanced policies may dynamically adjust the hibernation timeout based on battery state, allowing more aggressive power savings when charge levels are low and favoring responsiveness when the battery is near full. Hybrid suspend schemes further extend this idea by writing the hibernation image to disk before entering S0i3 or S3, preserving fast wake-up while protecting against unexpected power loss.

FreeBSD includes legacy support for S4BIOS, an early firmware-assisted hibernation mechanism intended to ease S4 adoption. This facility is not present on modern systems but can be queried via the `hw.acpi.s4bios` sysctl.

From an implementation perspective, S4 support is arguably simpler than S0i3, as it involves fewer platform dependencies. As of 2025, the FreeBSD Foundation has initiated a focused effort on implementing native S4 hibernation support [34].

7 Future Work

While the current implementation enables functional S0i3 suspend on select AMD platforms, several areas remain for further development:

- **Intel platform validation.** Testing on modern Intel systems is currently limited by hardware availability. Community testing and feedback on Intel laptops will be critical to validating portability.
- **Power profiling tools.** A FreeBSD equivalent to Linux `powertop` would greatly assist in diagnosing power consumption issues and could serve as a suitable Google Summer of Code student project.
- **RTC alarm wake support.** RTC-based wakeups enable deterministic suspend durations and are a key building block for automatic transition from S0i3 to S4.
- **Battery-level wake events.** ACPI battery devices expose programmable thresholds via the `_BLT` control method. Leveraging these thresholds to trigger wake events would allow graceful handling of critically low battery conditions.
- **Idleness-based power management.** Device idleness determination, as employed by

macOS IOKit, could enable intelligent runtime power gating of buses and devices.

- **NVIDIA driver support.** Enabling S0ix support in the FreeBSD NVIDIA driver stack remains ongoing work.

8 Acknowledgments

The author would like to thank Mario Limonciello (AMD) for assistance in identifying and resolving AMD-related platform-specific requirements.

Much of this work was sponsored by The FreeBSD Foundation, under their "Laptop Support and Usability Project".

References

[1] B. Widawsky, "Suspend to idle support," 2018, https://reviews.freebsd.org/D17675.

[2] ——, "Emulated s3 with s0ix," 2018, https://reviews.freebsd.org/D17676.

[3] A. Wibo, "acpi: Suspend-to-idle support (s2idle)," 2025, https://reviews.freebsd.org/D48734.

[4] ——, "sched: Flag to force scheduler to always choose idle thread," 2025, https://reviews.freebsd.org/D54407.

[5] UEFI Forum, "Advanced Configuration and Power Interface (ACPI) Specification, Device Sleep Wake (_DSW)," ACPI Specification Section 7.3.1, 2021, version 6.4.

[6] A. Wibo, "acpi: Implement s2idle loop," 2025, https://reviews.freebsd.org/D54410.

[7] ——, "acpi: Make taskqueue only run on BSP," 2025, https://cgit.freebsd.org/src/commit/?id=c0df8f6.

[8] J. Roberson, "ULE scheduler idle loop implementation," 2007, https://cgit.freebsd.org/src/commit/?id=c0df8f6.

[9] The Chromium OS Authors, "Framework laptop embedded controller firmware," 2014, https://github.com/FrameworkComputer/EmbeddedController/blob/f6620a8/common/charge_state_v2.c#L2324.

[10] A. Wibo, "acpi_spmc: Add SPMC (system power management controller) driver," 2025, https://reviews.freebsd.org/D48387.

[11] Intel Corporation, "Intel Low Power S0 Idle," 2017, https://uefi.org/sites/default/files/resources/Intel_ACPI_Low_Power_S0_Idle.pdf.

[12] Microsoft, "Modern standby firmware notifications," 2024, https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/modern-standby-firmware-notifications.

[13] S. Sundar, "ACPI: PM: s2idle: Add AMD support to handle _DSM," 2020, https://github.com/torvalds/linux/commit/146f1ed.

[14] M. Limonciello, "ACPI: SPMC DSM ordering fixes," 2023, https://github.com/torvalds/linux/commit/f198478c.

[15] A. Stone, "ACPI Tables," 2016, https://www.kernel.org/doc/html/v6.4/arm64/acpi_object_usage.html.

[16] UEFI Forum, "Advanced Configuration and Power Interface (ACPI) Specification, Low Power Idle States (_LPI)," ACPI Specification Section 8.4.4, 2021, version 6.4.

[17] A. Wibo, "amdsmu: Initial work on a driver for the AMD SMU," 2025, https://cgit.freebsd.org/src/commit/?id=f261b63.

[18] ——, "amdsmu: Expose sysctls for metrics about last sleep," 2025, https://cgit.freebsd.org/src/commit/?id=e4e44f6.

[19] R. Marek, "AMD x86 SMU Firmware Analysis," 31C3 Conference Talk, 2014, Chaos Communication Congress.

[20] M. Limonciello, "Debug tools for AMD zen systems," 2021, https://git.kernel.org/pub/scm/linux/kernel/git/superm1/amd-debug-tools.git/about/.

[21] A. Wibo, "Clarify D3hot vs D3cold handling in ACPICA," 2024, https://github.com/acpica/acpica/pull/993.

[22] UEFI Forum, "Advanced Configuration and Power Interface (ACPI) Specification, Device Power Management," ACPI Specification Sections 7.3.8–7.3.11, 2021, version 6.4.

[23] A. Wibo, "acpi_powerres: Fix turning off power resources on first D-state switch," 2025, https://reviews.freebsd.org/D48385.

[24] ——, "acpi_powerres: acpi_pwr_get_state and getting initial D-state for device," 2025, https://reviews.freebsd.org/D48386.

[25] UEFI Forum, "Advanced Configuration and Power Interface (ACPI) Specification, Power State Current (_PSC)," ACPI Specification Section 7.3.6, 2021, version 6.4.

[26] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual, Instruction Set Reference: MWAIT," 2025.

[27] A. Wibo, "amdsmu: Sleep entry/exit hints for PMFW," 2025, https://reviews.freebsd.org/D48721.

[28] ——, "USB4 initial work on suspend routine," 2025, https://reviews.freebsd.org/D49453.

[29] R. Kumar, "Add amdgpio, driver for GPIO controller on AMD-based x86_64 platforms," 2018, https://cgit.freebsd.org/src/commit/?id=8ce574d.

[30] A. Wibo, "amdgpio: Mask and service interrupts," 2025, https://cgit.freebsd.org/src/commit/?id=a4d738d.

[31] ——, "amdgpio: Suspend routine," 2025, https://reviews.freebsd.org/D51589.

[32] ——, "linuxkpi: Support s2idle in pm_suspend_target_state," 2025, https://cgit.freebsd.org/src/commit/?id=a25cfca.

[33] ——, "Minimal linux kernel config to successfully enter s0i3," 2025, https://obiw.ac/public/blog/s0i3-linux-config.

[34] The FreeBSD Foundation, "FreeBSD Closes the Laptop Gap: Year One Project Update," 2025, https://freebsdfoundation.org/blog/freebsd-closes-the-laptop-gap-year-one-project-update/.