

---

# Analysis of the PASR Standard and its usability in Handheld Operating Systems such as Linux

---

**Todd Brandt**  
**Tonia Morris**  
**Khosro Darroudi**



Handheld platforms such as cell phones and PDAs are continually growing in complexity, as each new product generation requires wider application support, better graphics, and faster connectivity. The embedded processor market is meeting this demand with faster and more complex chipsets, with support for multiple power modes, and the capability of using larger amounts of memory. The modes supported typically include:

- an active, or run state, which consumes the most power, but provides the highest performance and the most functionality
- one or more low power states, which incrementally reduce performance and functionality while increasing battery life.

Each handheld platform is different, but the most ubiquitous component is memory. In active mode, the highest consumers of power are typically the display, backlight, and CPU, so memory power represents a small percentage of the total. However, in low power modes, such as standby and sleep, the display, CPU, and most of the system may be shut down because the device is not in use. In these modes, memory power consumption becomes a greater factor in the battery life of the device.

Volatile memory is typically comprised of DRAM, which uses more power but has better density per die size than SRAM because it uses memory cells that drain energy over time. Thus, DRAM must be refreshed periodically, by the CPU, to retain its data. In low power mode, the CPU is off, and thus DRAM can enter a mode where it refreshes itself with just enough power so that its contents are retained, but cannot be accessed or modified. This mode is aptly called self-refresh.

Handheld devices, in general, have no real standard for inactive mode power consumption (less is obviously better), but a good estimate is in the single digit Watt range for active mode, and anywhere from **4mW-**

**20mW** in sleep mode<sup>1</sup>. As for Low Power DRAMs, the manufacturers typically provide detailed power numbers. The table below shows our test data using a Mainstone II board. It indicates that self-refreshed DRAM is responsible for a significant portion of sleep mode handheld power; anywhere from **5%** (0.977mW/20mW) - **25%** (0.977mW/4mW).

With companies now producing Low Power DRAM for handhelds, a new feature was developed called Partial Array Self Refresh (PASR). This feature enables the DRAM to retain state in only part of memory, thus further reducing the self-refresh power. If handheld operating systems could use this feature, they could potentially add hours of battery life. This has not happened because the PASR footprint provided cannot be easily integrated into established OS memory management schemes.

The PASR power savings below assumes that the handheld device consumes a total of 4mW in sleep mode with all the DRAM self-refreshed. Thus, the rest of the device consumes 3.023 mW. This value, added to the PASR-enabled self-refresh power for DRAM is used to determine the savings PASR can provide if a portion of memory is unused.

## Power Saving in Relation to PASR

PASR Ratio	Memory Retained	Self-Refresh power	PASR Power Savings
Full SR	<b>64 MB</b>	0.977mW	<b>0%</b>
1/2 PASR	<b>32 MB</b>	0.670mW	<b>8%</b>
1/4 PASR	<b>16 MB</b>	0.516mW	<b>12%</b>
1/8 PASR	<b>8 MB</b>	0.424mW	<b>14%</b>
1/16 PASR	<b>4 MB</b>	0.374mW	<b>15%</b>

Handheld w/64MB DRAM using ~4mW total in sleep mode<sup>2</sup>

In this paper, we review the current PASR feature as it has been specified in the LP-DRAM standard. We also describe the limitations of this approach, within the context of the embedded Linux operating system. We further propose additional PASR variants, and compare the effectiveness of these proposals to the original LP-DRAM PASR specification. We conclude with a recommendation of PASR for future DRAM technologies, and a proposal for how embedded operating systems can be modified to use existing PASR.

## PASR Methodologies

In the following sections, three PASR techniques are described that reduce the power consumption in dynamic random access memory (DRAM) devices during self-refresh mode:

- Single Ended
- Dual Ended PASR
- Bank Selective PASR

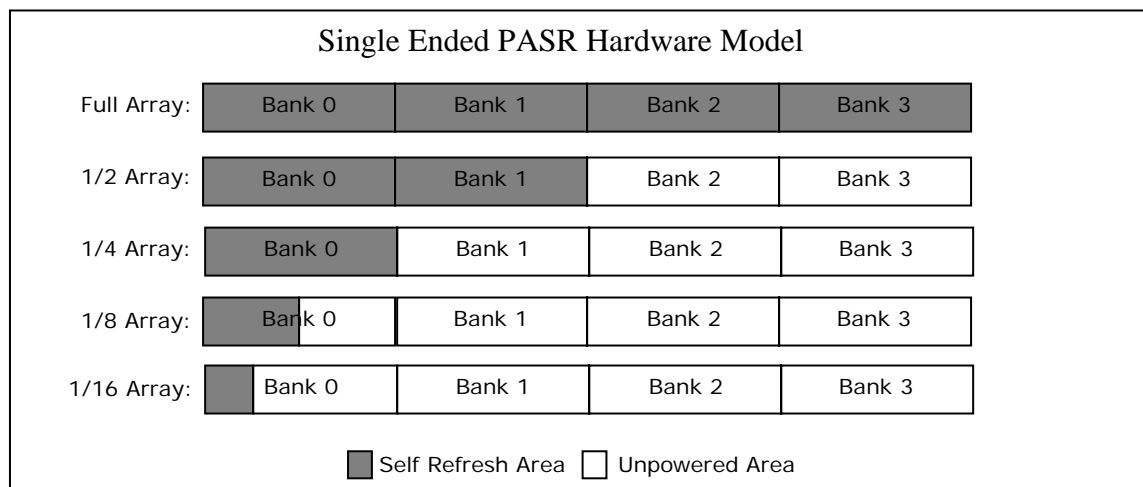
These techniques reduce power by refreshing only a portion of memory, where the remaining memory is powered off and the contents lost. To software, they differ only in how the PASR footprint is enabled (that is, which parts of memory are shut down to save power). The existing standard is Single Ended Partial Array Self Refresh (PASR), and we are proposing two new standards called Dual Ended PASR and Bank Selective PASR.

### Single Ended PASR (Existing Standard)

Currently there is only one JEDEC-supported Partial Array Self Refresh (PASR) standard available. The PASR feature is supported in many of the low power SDR and DDR SDRAMs on the market and employs the simplest method of selectively refreshing a portion of the memory that is used. Thus, with PASR, the self refresh may be restricted to a variable portion of the total DRAM space. The JEDEC PASR standard currently states that compliant chipsets must support self-refresh in the following 3 proportions, as shown in the figure below:

- all of memory (full array)
- the first 1/2
- the first 1/4

Vendors can support optional proportions of the first 1/8 and first 1/16. Any data outside the selected Self Refresh area in the DRAM will be lost.

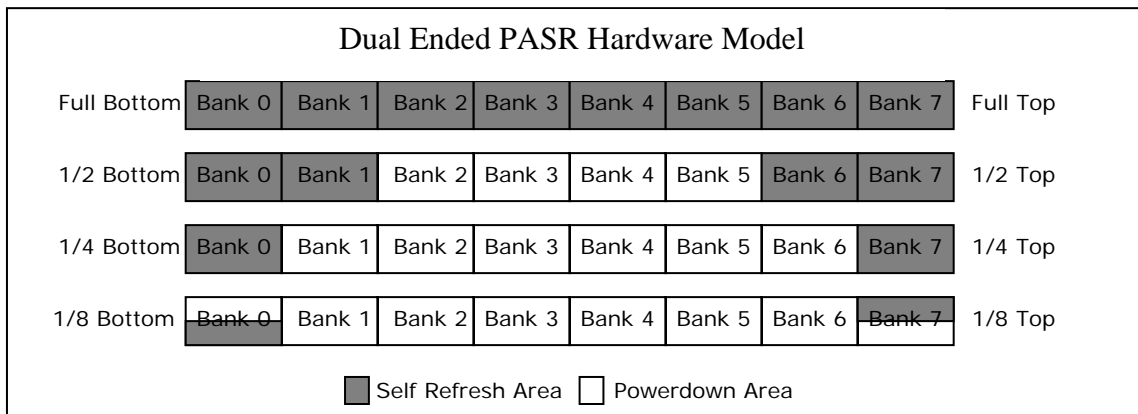


Normally, DRAM is configured by sending MRS (Mode Register Set) commands through the controlling processor's Dynamic Memory Controller register file. This allows changes to things like CAS latency, refresh time, etc. However PASR is a special feature that is not a part of the MRS command set, but rather the EMRS (Extended Mode Register Set) command set. Newer mobile processors have special register sets or configurations which allow EMRS commands to be sent by software, for example Intel's PXA270 has two registers, MDMRS and MDMRSLP, which are written by software to send out MRS and EMRS commands respectively. The PASR commands will have no effect on memory while the system is in run mode and are enabled only when the system is placed in a low power state (that is, standby/sleep).

List of EMRS commands for Mainstone III	
PASR	EMRS for PXA270 CS0/1, CS2/3
1 / 1	MDMRSLP = 0xC000C000
1 / 2	MDMRSLP = 0xC001C001
1 / 4	MDMRSLP = 0xC002C002
1 / 8	MDMRSLP = 0xC005C005
1 / 16	MDMRSLP = 0xC006C006

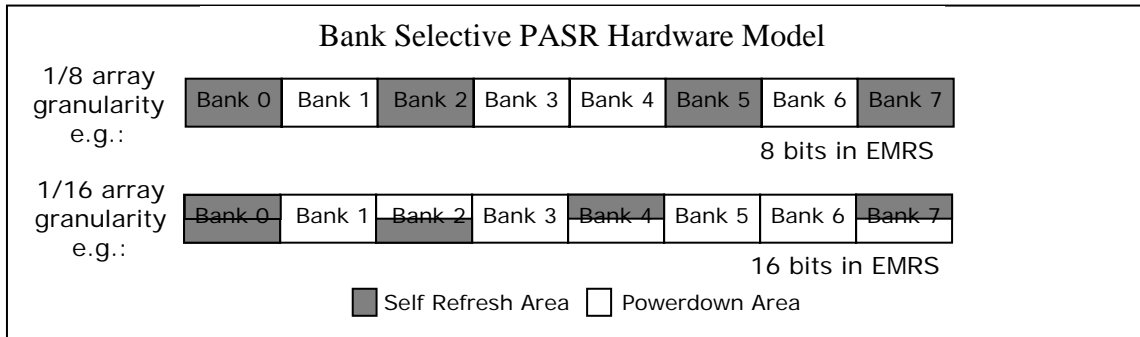
### Dual Ended PASR (Proposed)

In the Dual Ended PASR method, the self-refresh regions are selected at the top and bottom of the memory, as shown in the figure below (the figure represents an 8 bank memory space, which is what we propose to allow for more PASR granularity). This method is motivated by embedded operating systems that use mostly the top and bottom of the memory (that is, Linux OS). Linux kernel physical memory grows up from the bottom and user (application) memory grows down from the top. Any residual data in the middle portion of the DRAM array, must be moved to either end of the array by the OS before the PASR commands are issued to the DRAM device.



## Bank Selective PASR (Proposed)

In the Bank Selective PASR technique, each bank is independently marked to be self refreshed. Thus, any combination of memory banks can be self refreshed. This technique can be extended to cover smaller granularity, that is, any combination of half banks can be self refreshed. The figure below shows 2 examples of this Bank Selective PASR: in the top version, banks 0, 2, 5 and 7 are refreshed. In the bottom version, the shaded 1/2-bank areas are self refreshed.



The benefits/drawbacks of these methods of PASR are shown in the following section, as an embedded operating system is analyzed using each of them.

## Embedded Operating System Support Linux Memory Usage

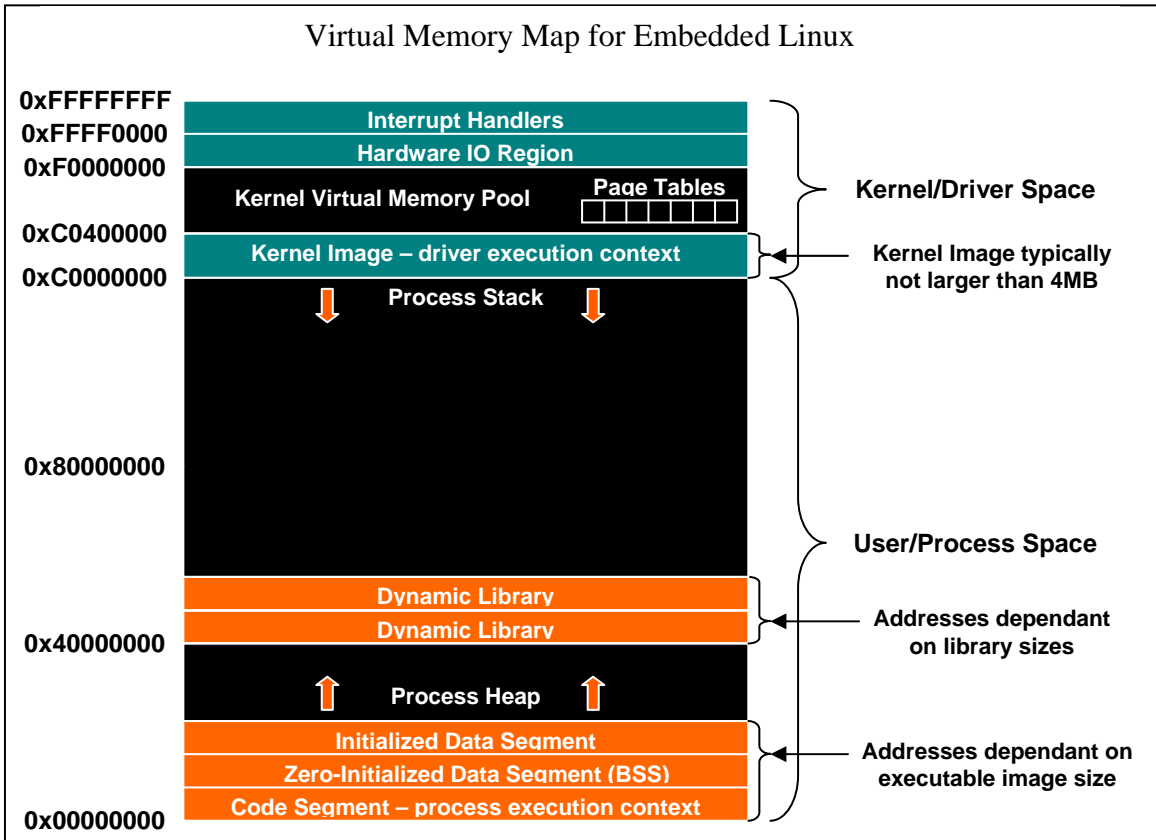
Embedded Linux can be ported to just about any mobile processor on the market, including ones both with and without a Memory Management Unit. The MMU is responsible for maintaining two distinct memory address spaces:

- virtual (or linear) address space is what application and device programmers see. It is as large as the processor's addressing permits, for instance for 32 bit addressing the virtual address space is  $2^{32}$  bytes wide, or 4GB.
- physical address space represents the range of addresses actually used by the DRAM chip(s) on the system. This space is only as large as the available DRAM and doesn't necessarily have to be anywhere near as large as the virtual address space.

On processors with MMU functionality, Linux is capable of maintaining a constant virtual space for applications and devices, while varying the amount and location of physical DRAM used underneath. This paper focuses on embedded linux implementations for processor's with MMUs because this is the most common case. The virtual memory map (that is, linear address space) is very different from the physical address space, so we'll explore both in detail.

# Virtual Memory Footprint

The 32-bit virtual memory space is broken into two regions: kernel and user space. Kernel/Driver code, executing in kernel space, does so in supervisor mode, where application code, running in user space, does so in user mode. This way kernel space is protected by the CPU permissions only allowing trusted code to access it (such as, drivers). Kernel space is typically located in the upper ¼ of memory (0xC0000000 and above), and user space is located in the bottom ¾. The diagram below illustrates how virtual memory is laid out for handheld linux and what each specific region is used for.



## Linux Execution and Virtual Memory Utilization

When Linux boots, it starts with the MMU disabled, so initially it deals only with physical memory. The kernel image is copied to physical address 0x8000 in DRAM and executed. First a master page table is created and placed just before the kernel, which describes all available DRAM. The MMU is then switched on, mapping all of DRAM to virtual address 0xC0000000. The kernel reserves 64K of virtual memory for the interrupt handler code (usually at the very top of virtual memory), and sets up all the mappings for the hardware registers (UART, USB, LCD, GPIO, etc). Once kernel space is established and all drivers are initialized, the linux kernel moves on to establishing user space. This involves reading in the file system and actually executing processes.

Each process that runs on the system does so in its own memory “context”. Each context has its virtual memory space established and maintained by the kernel using a separate page table. So each process can “see” the entire user space, but its underlying physical memory is different from the other processes. Multiple processes and their contexts are run in time slices. This means that each process executes for a time, then is halted, and execution is passed to the next process in a rotating queue. The act of passing execution is called “context switching” and its frequency varies, depending on the kernel configuration and the current system load.

Each context includes the mappings for both user and kernel space because each new page table is initialized with the master page table contents. So, code executing in kernel space has full access to the entire memory map for the current process. Switching back and forth between kernel and user space execution does not require a page table swap.

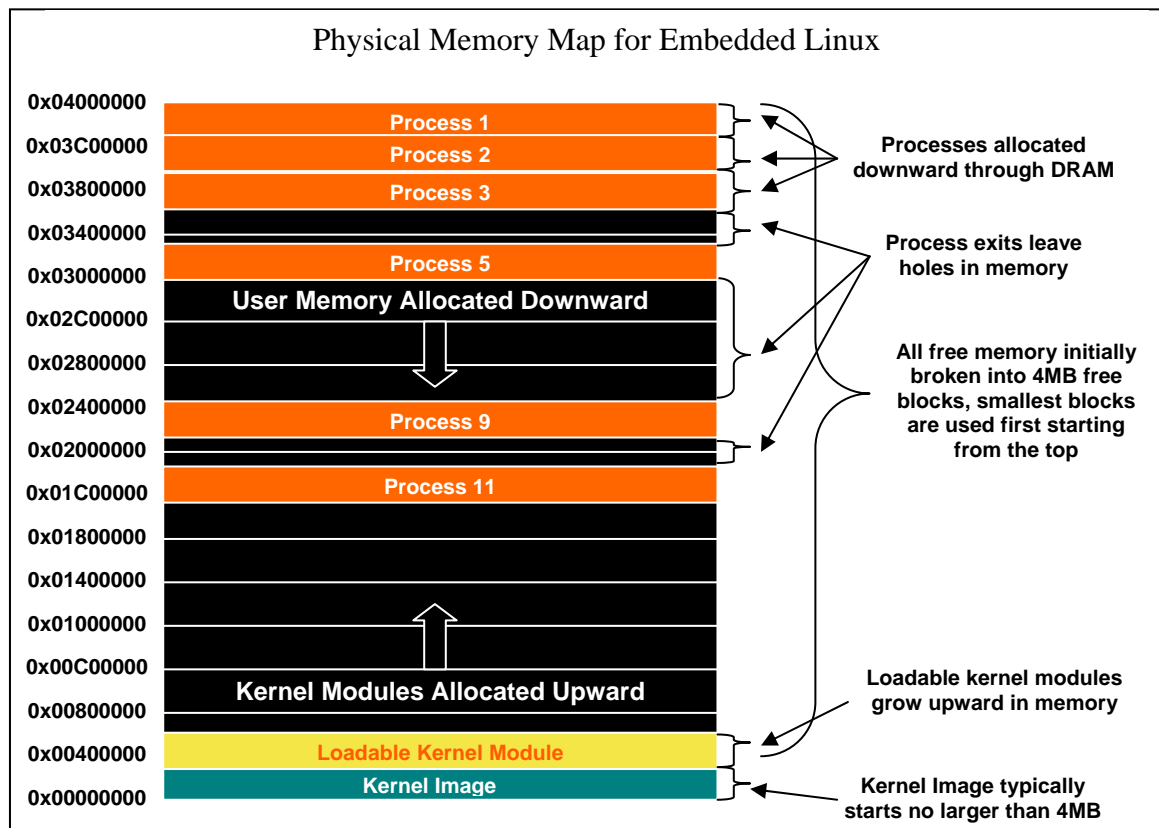
When a process is executed, the kernel creates a new context, and maps just enough physical memory to copy the executable image into DRAM (starting at least a page above virtual address 0, because 0 is reserved for faults). The executable is broken into the same sections as the kernel: code and data (uninitialized global data, zero-initialized global data). Once the kernel schedules the process for execution, it sets the stack to grow down from the top of user space, and the heap to grow up from the end of the executable image. This way the chances of the two spaces colliding is minimized. Any dynamic libraries the process pulls in are placed at virtual address 0x40000000.

The kernel reserves virtual memory whenever a process or driver requests it, but it doesn't actually map in the underlying physical memory until the memory is accessed (copy-on-write). So, the physical DRAM is used only when it absolutely has to be, and the kernel controls the allocation through a global memory management scheme. Ideally, the kernel tries to allocate contiguous blocks of physical memory as contiguous memory is most likely to evenly fill physically tagged caches. Thus, the kernel organizes memory to grow inward from the ends of physical memory, maximizing contiguous memory in the middle. The physical memory map is described in detail in the next section.

## Physical Memory Footprint

The physical memory map for Linux is completely independent from the virtual map and is designed to maximize contiguous space. Given that the kernel image will always be at the start of DRAM, the Linux kernel maximizes contiguous space by allocating runtime memory from the end of physical DRAM moving downward.

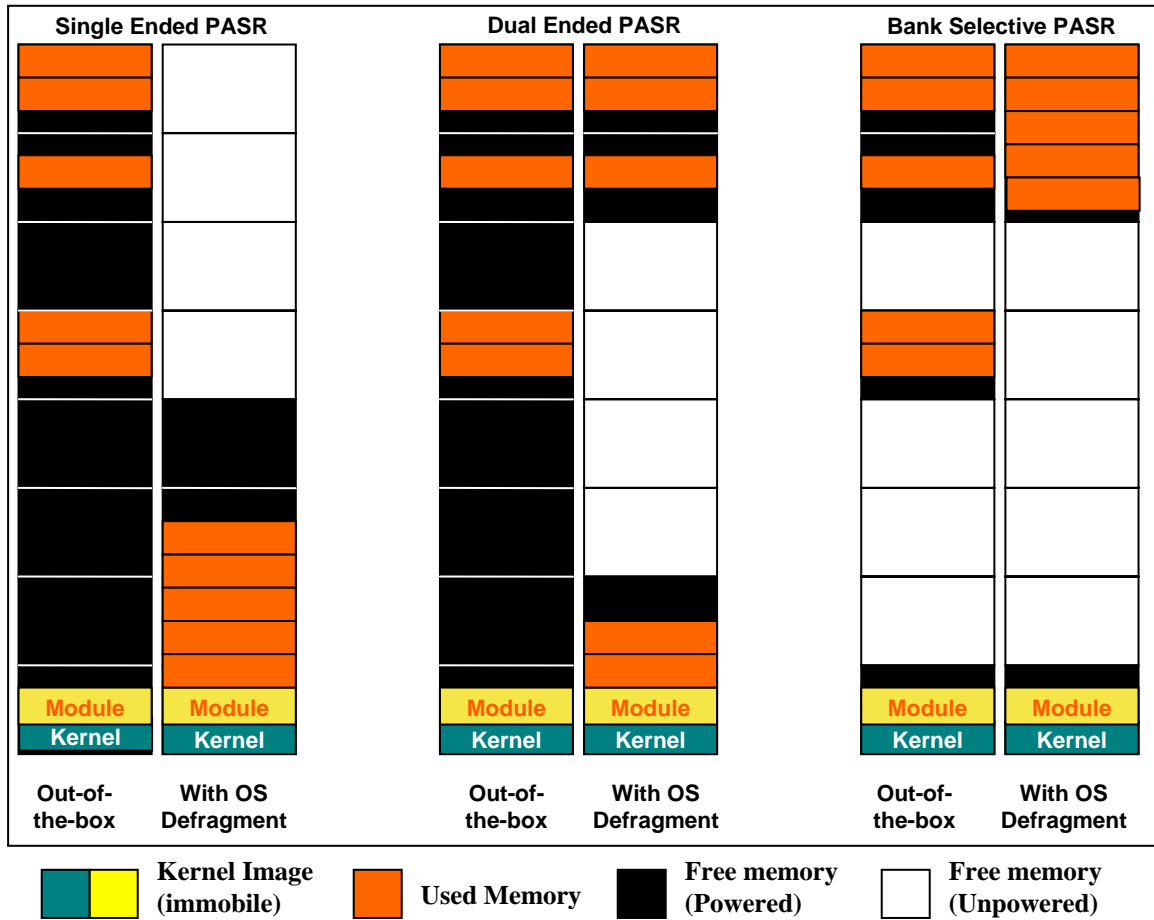
The kernel starts by breaking available memory out into large, contiguous blocks (typically 4MB or more). It then maintains memory using the buddy system, where physical memory is always allocated in combinations of blocks of  $2^n$  pages (where  $n$  is the order, that is, 4K is a 0<sup>th</sup> order block, 8K is a 1<sup>st</sup> order block, 16K is a 2<sup>nd</sup> order block, etc).



When physical memory is allocated, that is, on process start, or when malloc'ed memory is written to (copy-on-write), the kernel scans for the smallest order block that will fit starting from the top of DRAM. As the number of running processes increases, or new allocations are spawned from drivers or processes, the physical memory used grows downward.

When a process exits or a large enough memory block is freed, its DRAM space is unmapped and becomes a gap in memory. This process is called memory fragmentation and becomes more and more prevalent the longer a device is used and the more frequently the use case changes (such as checking email, playing an mp3, watching a video, etc). As new processes and allocations are created, the gaps are filled whenever possible, but fragmentation is still an inevitable part of any OS memory map.

# OS Memory Compatibility with PASR Standards



The figure above shows a typical physical memory map for embedded Linux and its effect on PASR for the 3 methods described in section 2. Each of the methods can be rated by its probable effectiveness with no OS modifications, and the level of OS modification needed to achieve maximum effectiveness. The first column in each of the 3 groupings represents what level of power savings Linux will achieve “out-of-the-box”, in other words, without any modification to the OS or its existing memory allocation infrastructure. The column on the right represents how much OS modification is needed to achieve maximum effectiveness. The results can be summarized as follows:

	Existing Standard	Proposed Future Standards	
	Single Ended PASR	Dual Ended PASR	Bank Selective PASR
Probable effectiveness with no OS modification	<b>Zero:</b> Memory will always be allocated at the top of DRAM	<b>Medium:</b> Light usage will not touch middle memory	<b>High:</b> All but the highest usage level will have some benefit
OS modification needed to maximize effectiveness	<b>High:</b> all of memory needs to be compacted toward the start of DRAM	<b>Medium:</b> as usage gets heavier middle memory can be compacted to either end	<b>Low:</b> Memory gaps can aid PASR anywhere in the map

Conclusion: Ultimately Bank Selective PASR is the most attractive methodology. It provides the highest level of power savings out of the box, and the OS can be enhanced much more easily to maximize PASR effectiveness.

## Modifying an OS to support Existing PASR

Now that we know how the PASR standard works and how Linux's memory map functions within it, the next step is determining how to maximize it. For either the existing or proposed PASR standards, there are really only two ways that an OS can be modified to enhance PASR effectiveness:

- The first is to change the global allocation scheme to favor free memories that fall within the smallest possible PASR boundary. This method is ultimately doomed to fail however, since gaps are inevitable over time as process memory maps change and the use case is altered.
- The only sure-fire way to ensure PASR is fully optimized is through a second method: defragmentation or "compression" of memory into the PASR boundary prior to entry into low power modes. This method is more extensible to other operating systems in that all one needs is a method of detecting and moving used memory. The following sections describe what must be done to enable PASR compression in Linux.

This would need to be done the instant before the system enters low power mode when no memory can be mapped or unmapped. To do this, we need to add two pieces of functionality:

- a method for determining the physical memory usage map
- a method of compressing memory before low power mode entry and decompressing memory after low power mode resume.

## Taking a snapshot of the physical memory footprint

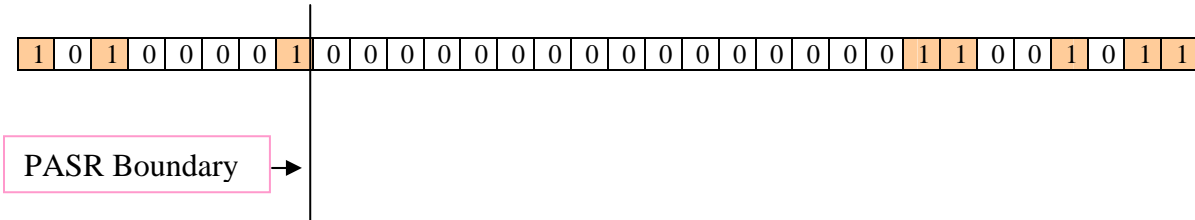
Determining the physical memory map in linux is very simple. The hard part is timing this determination so that it occurs only after there is no possibility of new memory being allocated. Linux stores all the information about memory usage in the "zone list". There are three zones: DMA, Normal, and Highmem, all of which contain some portion of the physical memory map resident on the DRAM chips. Each zone has a list of free memory regions called a "free list". If we loop through each zone and each of their free lists, we can compile a complete list of what physical memory is designated free by the Linux kernel. So, we can assume that all memory falling outside of these regions is used.

## "Compressing/Decompressing" Memory

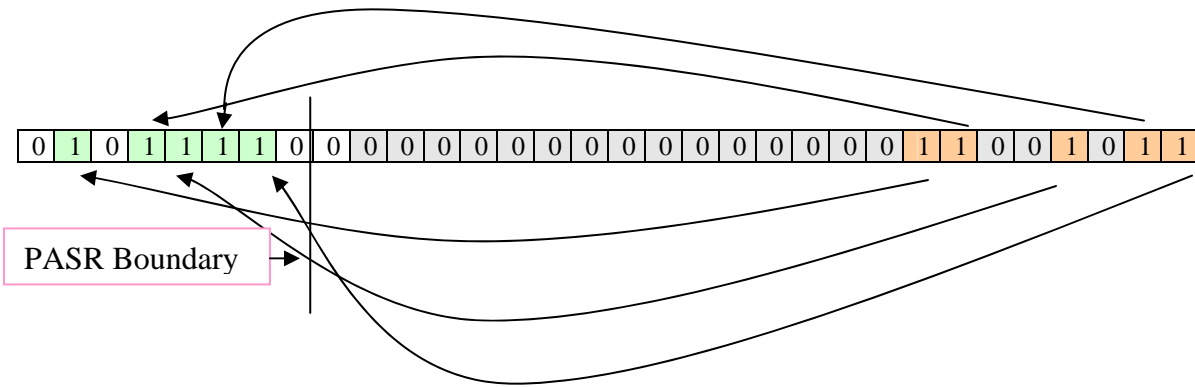
From the total number of used pages, we now know what the PASR boundary *can* be after compression. For existing PASR, this is a simple calculation, we simply take the used page count, divide by the total page count, and round up to the smallest PASR boundary that will fit (0.0625, 0.125, 0.25, 0.5, 1). For example if we discover 2233 4K pages are used in 64MB of DRAM (16,384 pages total), the used fraction is about 0.14, so the smallest PASR boundary that will work is 0.25, or ¼.

Once we have the used pages identified, and the PASR boundary is known, all we need to do is relocate all the used memory that resides outside the PASR boundary into "PASR-safe" memory. The simplest and fastest way to do this, is to simply move used pages from the dead side of memory (the part that will be lost during PASR) to unused pages on the live side (the part that will be state-retained during PASR).

To do this, we first need to create a simple map, which can be read to determine which pages of memory to move on compression, and which to move back on decompression. The most efficient method is with a simple bitmap, where there is 1 bit for every page of DRAM, (0 = free and 1 = used). We would initialize all the bits to 1, denoting “used”. Then we’d run through the zone lists and set any free page bits to 0. As an example, let’s assume that memory consists of 32 4K pages and we discover 24 pages are free, the PASR boundary will then be ¼. The following could be the bitmap describing the memory layout.



Now we know that the 5 pages on the right of the PASR boundary are going to have to be moved into the gaps on the left. For simplicity, we can now invert all the bits on the PASR-safe side of the bit map. This way the bit map now simply denotes source and destination pages. The following shows the finalized compression bitmap, with arrows showing page memory moves.



The bitmap and boundary should be static globals that are compiled directly into the kernel space, thus they will be guaranteed to reside in PASR-safe memory. The bitmap can be just an array of unsigned ints using the suite of bit functions, defined in bitopts.h, to make usage easy. Each bit, at an index less than pasr\_boundary, represents a 4K page of memory that will be retained in low power mode. Each bit, at an index equal to or greater than pasr\_boundary, represents a 4K page that will be destroyed in low power mode.

The bitmap is read left to right with two incrementors, one for the live side (below pasr\_boundary), and one for the dead side (above pasr\_boundary). Each ‘1’ found on the live side represents a destination page, and each ‘1’ on the dead side represents a source page (the swapping is simply reversed on decompression but the bitmap is read the same way). From the figure above, the compression algorithms would read the following:

<b>Pasr_memory_compress</b>	<b>Pasr_memory_decompress</b>
Copy page 24 to page 1	Copy page 1 to page 24
Copy page 25 to page 3	Copy page 3 to page 25
Copy page 28 to page 4	Copy page 4 to page 28
Copy page 30 to page 5	Copy page 5 to page 30
Copy page 31 to page 6	Copy page 6 to page 31

So basically, when the system is about to enter a low power state, a “pasr\_calculate” function is called which creates the mappings and determines whether PASR is worthwhile (if more than half of memory is used, PASR should be skipped). Then a “pasr\_memory\_compress” function reads the bit mapping, carries

out the compression, and the system suspends. On resume, a “pasr\_memory\_decompress” function is called to restore memory and the system is returned to a working state.

The following rules *must* hold true if this methodology is going to work safely:

#### **Cache coherency must be maintained**

Both L1 and L2 cache must be completely invalidated and flushed immediately after memory compression and decompression. This is because low power modes are not guaranteed to retain cache contents, and in some cases completely disable the MMU.

#### **Kernel memory cannot be moved**

No static kernel memory can be moved. This means anything before the physical address of the symbol “\_end”. So, if any physical memory page resides below “(unsigned long)virt\_to\_phys(&\_end)”, it *must* be considered used.

#### **A PASR-safe page table must be used**

The MMU must switch to a page table located in static kernel memory before any memory is compressed. This is because the currently active page table can reside in physical memory outside the PASR boundary, which is about to be destroyed. The switch to the static table must occur immediately before compression, and the old table must be restored immediately after decompression. This rule can be satisfied by using the kernel page table, which is always located 4K from the start of physical DRAM. This table contains all the necessary mappings for DRAM, IO, interrupt vectors, cache flush memory, etc., because it is used as the template for process page tables. “Pasr\_calculate” can be called with the current page table since all tables map kernel memory. But “pasr\_memory\_compress” must first switch to the kernel table before it moves any memory. “Pasr\_memory\_decompress” switches the page table back when it’s done.

#### **A PASR-safe stack space must be used**

The kernel stack pointer must be redirected to a preallocated stack space located in static kernel memory. This is because the active kernel stack memory may be located outside the PASR boundary, which is about to be destroyed. The size of the PASR-safe stack needs to be large enough to accommodate any function calls used for compression, decompression, or any other activity occurring between the entry to or resume from standby/sleep.

#### **Execution flow must be tightly controlled on lp mode entry/resume**

On low power mode entry, no code, executing between the call to “pasr\_calculate” and the entry to standby/sleep, can create or access memory that is outside the PASR boundary. On low power mode resume, no code, executing between resume from standby/sleep and the completion of “pasr\_memory\_decompress”, can create or access memory that is outside the PASR boundary. This rule requires that the execution distance between the calculate/compress calls and the system suspend instruction be as small as possible. The same applies to the distance between the system resume point and the decompress call.

#### **DRAM is untouchable in low power mode**

DRAM memory, particularly the memory outside the PASR boundary, must not be accessed or modified by any entity (the core, a peripheral, a comm subsystem, etc) during the period where linux is in standby or sleep. This would both nullify the benefits of PASR, and may crash the system if the entity uses something compressed.

For all of this to function, there is a penalty in terms of time to entry and exit to/from low power mode. This has to be taken into account when implementing PASR usage within existing PASR or even when maximizing its effectiveness in our proposals for future PASR. For most platforms, this technique will only be usable for deep sleep mode, because it’s the only time a user will tolerate the fraction of a second it takes to go online.

---

---

## References

- 1 – Marc A. Viredaz and Deborah A. Wa, “Power Evaluation of a Handheld Computer,” Published by the IEEE Computer Society, 2003, <http://www.hpl.hp.com/research/papers/2003/handheld.pdf>.
- 2 – Our test setup was a Linux 2.4.21 build for the Mainstone III platform which uses PXA270 and Infineon Mobile DRAM (HYB25L256160AC-7.5). Power was measured on the VCC\_2P5V tap which supplies the DRAMs. The assumption is that the end device will draw ~4mW total power in sleep mode<sup>1</sup>. So the PASR power savings is calculated by comparing 4mW to the reduced DRAM self-refresh power.
- 3 – Joint Electron Device Engineering Council (JEDEC) – [www.jedec.org](http://www.jedec.org)