Nexus
Memory

Broadcom Corporation
5300 California Avenue
Irvine, California, USA 92677
Phone: 949-926-5000
Fax: 949-926-5203

Web: www.broadcom.com

Revision History

Revision	Date	Change Description
0.1	1/4/11	Initial draft
0.2	10/20/11	Update 7425, remove 7422, update msg_modules=nexus_platform_settings sample
0.3	12/19/11	Clarify fake addressing, add section on debug
1.0	6/15/12	Added Steps to Minimizing Memory Usage
1.1	8/30/12	Added section on Client Heaps, expanded notes on overriding default configurations

Table of Contents

Contents

Introduction

Embedded systems usually run with limited memory size, limited bandwidth and limited memory access. Therefore, the application designer must understand these limitations and design for them.

The document begins by describing Nexus heaps and how they are configured. This includes configuring which heaps are used for certain features based on memory requirements.

Next, the document gives step-by-step directions that each project must follow to determine and configure the minimum memory usage.

Finally, the document covers a variety of topics including memory mapping, typical memory usage and debugging memory problems.

Nexus Heaps

What is a heap?

Broadcom set-tops divide system memory into two groups: memory managed by the operating system (OS) and memory managed by Nexus. Memory managed by the OS is called “system memory”. Memory managed by Nexus is called “device memory”. Nexus manages its memory by means of heaps. A heap is a region of physically contiguous memory from which you can allocate blocks of memory. All heaps provide physical offsets (also called physical addresses) for non-CPU device access. Some heaps are memory mapped for CPU access. The heap allows you to convert between physical offsets and virtual addresses.

System memory is not guaranteed to be physically contiguous; therefore it is not directly usable by the device.[1]

Platform Default Heaps

The Nexus platform code will configure a set of heaps for the typical usage of the reference board[2]. Your application can override these settings after calling NEXUS_Platform_GetDefaultSettings and before calling NEXUS_Platform_Init. However, be aware that many chips have complex memory architectures which are required for even basic features like video decode. You can study the default heap layout to learn some of these requirements.

Nexus will consult the OS to learn what memory is available for its use. Any memory used by the OS is not usable as a Nexus heap. Linux reports these regions as “bmem” regions. Each bmem region is physically contiguous. There may be zero, one or more bmem regions on each MEMC. See later section for information on how to configure and read bmem regions in Linux.

Nexus will automatically bound all heaps to fit within the reported bmem regions. It will not allow a heap to be created outside of the bmem regions.

When writing general purpose code, it’s important to allow flexibility for heap configuration. The heap layout for one system may be quite different from another system. The application is the final arbiter of the system design and should be able to override any default heap configuration.

[1] Physically discontiguous memory results from memory mapping. If a device wants direct memory access apart from the OS, it must be contiguous and locked. Nexus Dma and Graphics2D interfaces have support for offsets, not addresses. So, access to kernel allocated memory is possible, provided your application can convert Linux virtual addresses to offset and can lock the pages.

[2] See nexus_platform_$(NEXUS_PLATFORM).c and NEXUS_Platform_P_GetDefaultSettings.

Configuring Heaps

You will likely need to customize your heap configuration for your application needs. You will also need to understand the meaning and use of each heap.

The following is a snippet from nexus_platform_init.h for customizing heaps.

```
typedef struct NEXUS_PlatformSettings
{
    struct {
        unsigned memcIndex;
        unsigned subIndex; /* aka MEMC region */
        int size;
        unsigned alignment;
        bool guardBanding;
        unsigned memoryType;
        bool optional;
    } heap[NEXUS_MAX_HEAPS];
}
```

After calling NEXUS_Platform_GetDefaultSettings, you can change any of the heap parameters. The heaps will only be created when you call NEXUS_Platform_Init.

Please see the header file for detailed API-level comments.

- The “MEMC region” is defined by memcIndex and subIndex. MEMC0 has memcIndex == 0. The subIndex is the addressing range on that MEMC. For example, on 7452 MEMC has subIndex 0 for 0x0000_0000 through 0x1000_0000, subIndex 1 for 0x2000_0000 to the end. Nexus will automatically match the requested memcIndex/subIndex to the bmem regions. There is currently no way to specify that a heap be created in a specific bmem region.

- If size == -1, the remainder of the memory in the bmem region will be assigned to that heap. You can only have one size == -1 per bmem region.

- alignment is the minimum alignment of allocations in the heap. This can help reduce fragmentation. Nexus will also apply a chip-specific minimum alignment for cache coherency. The max of the user alignment and the internal alignment will be used.

- guardBanding is a debug option for catching memory overrun bugs. It requires memoryType of eDriver or eFull so that the driver has CPU access to writing and reading guard bands.

- memoryType is a bitmask which controls the memory mapping. See nexus_types.h for bitmasks and macros. The following are typical combinations:

NEXUS_MemoryType	Mapping	Usage
NEXUS_MemoryType_eFull	Driver cached Driver uncached Application cached	Default heap Playback
NEXUS_MemoryType_eDriver	Driver cached Driver uncached	Magnum use only VBI, XPT
NEXUS_MemoryType_eApplication	Application cached	Graphics Record
NEXUS_MemoryType_eDeviceOnly	No mapping	Picture buffers No CPU access

- The optional boolean allows NEXUS_Platform_Init to succeed even if the heap cannot be created. By default, Init will fail if all requested heaps cannot be created, which makes system debug much easier.

Using Heaps

Nexus heaps can be specified in two ways: by index or by handle.

When setting NEXUS_PlatformSettings for NEXUS_Platform_Init, no nexus handle exists. Therefore, all heap configuration must be set using heap indices. These indices refer to heaps that will be created. The following are typical:

API	Usage
NEXUS_DisplayModuleSettings. primaryDisplayHeapIndex	Heap used by VDC if per-window and per-source heaps are not specified
NEXUS_DisplayModuleSettings. videoWindowHeapIndex[]	Default per-window heap for VDC
NEXUS_VideoDecoderModuleSettings. avdHeapIndex[]	XVD heap for picture buffers
NEXUS_VideoDecoderModuleSettings. hostAccessibleHeapIndex	XVD heap for FW, userdata

After NEXUS_Platform_Init has completed, a set of NEXUS_HeapHandles are available using NEXUS_Platform_GetConfigurtion. If you want to customize your memory usage, you can pass the heap handle to the Nexus API. The following are typical uses:

API	Usage
NEXUS_SurfaceCreateSettings.heap	Heap to allocate surface from
NEXUS_VideoWindowSettings.heap	Custom per-window heap for VDC
NEXUS_VideoDecoderOpenSettings.heap	Custom XVD heap for picture buffers
NEXUS_PlaypumpOpenSettings.heap	Heap for CDB allocation
NEXUS_PlaypumpOpenSettings.boundsHeap	Optional bounds check for scatter-gather
NEXUS_Graphics2DOpenSettings.heap	Heap for M2MC HW/SW fifo allocation
NEXUS_Graphics2DOpenSettings.boundsHeap	Optional bounds check for blits

Client Heaps

If you are using Nexus in a multiprocess configuration, there is additional heap complexity. The server decides which heaps the client can access. In a secure system, the client should not be given access to any heap that could compromise the server. Therefore, it is recommended that you have a dedicated heap or set of heaps for clients.

Client heap numbering is not necessarily the same as the server heap numbering. Client heap numbering is the index of the NEXUS_ClientConfiguration.heap[] array. Server heap number is the index of the NEXUS_PlatformSettings.heap[] array. When the server assigns heaps to the client, it determines its numbering. It can also be different per client.

If an application uses an interface with a NEXUS_HeapHandle parameter, and if it leaves that parameter as a default NULL value, the nexus module code will select a default heap. If the call came from a client, it will select a default client heap. If the module code requires NEXUS_MemoryType_eFull mapping, it will select the first heap in the client's heap[] array with that mapping. If there is no driver-side mapping requirement, it will select the first heap in the client's heap[] array.

Steps to Minimizing Memory Usage

The following steps should be performed for every Nexus project:

1. RTS Analysis
2. Complete the System Memory Worksheet to determine video decoder and display heap requirements
3. Customize platform code to reduce memory usage
4. Configure Linux bmem regions
5. Write application to configure features with minimal and reliable memory usage
6. Reduce system memory

RTS Analysis

Broadcom Memory Controllers (MEMC's) use a Real-Time Scheduling (RTS) system to allocate guaranteed or round-robin memory bandwidth to the various memory clients in the system. There is a default RTS delivered with every reference platform, but each project should have its own RTS analysis done. RTS analysis may cause a change in the layout of memory allocations in the system.

This document will assume that you are using the default RTS programming for your project.

System Memory Worksheet

Memory usage estimates per feature are provided for each chip in a Microsoft® Excel® worksheet called System_Memory_Worksheet.xls. This memory worksheet can be requested from your FAE.

The System Memory Worksheet allows you to calculate the heap requirements for your features. You should enable and disable a variety of features to learn what each feature costs in terms of memory. If you simply enable every feature, it may require an unexpectedly large amount of memory.

There are two main blocks handled by the worksheet: VideoDecoder and Display. The requirements for each block will be added together into a "picture buffer heap". If your system has two memory controllers, these numbers may be split into two heaps, one on each MEMC.

The results of the spreadsheet must be programmed into NEXUS_PlatformSettings. Each reference platform has a nexus_platform_$(NEXUS_PLATFORM).c file which programs defaults. You can modify these defaults for your platform, or have your application set your own settings before calling NEXUS_Platform_Init.

Because of the complexity of the system, you will need to test the numbers and verify that they are correct for your usage mode. It is possible that adjusts will need to be made.

Customize platform code to reduce memory usage

You should customize your platform code to remove any features that are not needed. This will reduce your memory requirements.

Remove any module you are not using by deleting the "include …/<module>.inc" line from platform_modules.inc

Edit nexus_platform_features.h and reduce the NEXUS_NUM_XXX macros to only the features you are using. The default values of NEXUS_NUM_XXX are set to the chip capabilities, but you can reduce them below those numbers.

There are numerous ways this reduces memory. The following is a partial list:

- Reducing NEXUS_NUM_PARSER_BANDS will reduce the number of RS and XC buffers allocated by the transport core.
- Reducing NEXUS_NUM_RAVE_CONTEXTS to the actual number of decodes and records will reduce the number of XC buffers allocated by transport.
- Reducing NEXUS_NUM_REMUX or NEXUS_NUM_PLAYPUMPS will reduce the number of XC buffers allocated by transport.
- Setting NEXUS_SVC_MVC_SUPPORT to 0 will cause VideoDecoder to default to no SVC/MVC codec support. This reduces buffer requirements greatly. It is also possible to disable that codec support at runtime, but the macro is a simple way to do it for all apps.
- Setting NEXUS_NUM_MOSAIC_DECODES will cause VDC to allocate fewer register update lists (RUL's).
- Setting NEXUS_NUM_656_INPUTS or NEXUS_NUM_HDDVI_INPUTS to 0 will cause VDC to allocate fewer RUL's.

As you make each change, re-run with export msg_modules=BMEM (described below) and observe the differences. This will help you understand the nature of each change.

Configure Linux bmem regions

This section duplicates information provided with Broadcom's Linux software release. The features most often required for integration with Nexus are included here for convenience.

By default, Linux will use a portion of memory and leave the rest for Nexus/Magnum.

You can boot Linux with no boot parameters like this:

```
boot -z -elf flash0.kernel:
```

After Linux starts, you can learn what memory Linux is not using a /sys/devices node. Any memory not listed in /sys/devices must be assumed to be in use by the kernel.

> ***NOTE:*** *Starting with Linux 2.6.37, Linux will report all usable memory regions (even MEMC1) using /sys/devices. For all 65nm chips, Linux does not report memory regions which are not usable by Linux. See nexus_platform_settings.c for special exception code for those chips.*

For example:

```
cat /sys/devices/platform/brcmstb/bmem.*
0x04000000 0x0c000000
0x90000000 0x40000000
```

The first number is a physical base address. The second number is the size. The bmem regions are listed in order. Be aware that bmem.1 does not mean MEMC1. It simply means the second bmem region.

Each bmem region does not necessarily correspond to one Nexus heap. You may have multiple heaps in each bmem region, depending on the memory mapping requirements or fragmentation concerns.

bmem parameter

You can tell Linux how much memory to leave to Nexus/Magnum using the bmem boot parameter. The format of bmem is "bmem=size@physical_address". For example:

```
boot -z -elf flash0.kernel: 'bmem=256M@512M'

boot -z -elf flash0.kernel: 'bmem=256M@512M bmem=128M@128M'
```

The bmem parameters are translated as follows:

bmem	Size	Starting Physical Address	Example Usage
bmem=256M@512M	256MB	0x2000_0000	Reserve all of HIMEM on a 7405 with 512MB on MEMC0.
bmem=768M@512M	768MB	0x2000_0000	Reserve all of HIMEM on a 7422/7425 with 1GB on MEMC0
bmem=128M@128M	128M	0x0800_0000	Reserve 128MB of low memory on MEMC0, leaving 128MB for the kernel.
bmem=192M@64M	192M	0x0400_0000	Reserve 192MB of low memory on MEMC0, leaving 64MB for the kernel. This is the typical default for Linux.

memc1 parameter

For systems where MEMC1 is host-accessible, Linux can be configured to use MEMC1:

```
memc1=256M
```

This will be reflected in bmem output.

Overriding bmem

Nexus will read the bmem configuration and obey it strictly. However, because there can sometimes be kernel bugs or configuration problems, there is an override. You can create your own bmem.X files and place them in another directory. Just export “bmem_override” to that directory. For example:

```
export bmem_override=”/tmp”
```

Then place your own bmem.0, bmem.1, etc. in /tmp.

You must be sure that the memory in your bmem override files is actually available to nexus/magnum and not being used by the kernel. If you are wrong, the system will likely fail.

Write application for minimal memory usage

After configuring Nexus platform code, there is still a lot that your application should do to minimize memory usage.

The following API's allocate the most memory from Nexus heaps:

- NEXUS_Playpump_Open – allocates playback FIFO
- NEXUS_Recpump_Open – allocates CDB/ITB FIFO for record
- NEXUS_VideoDecoder_Open – allocates CDB/ITB FIFO for decode
- NEXUS_AudioDecoder_Open – allocates CDB/ITB FIFO for decode
- NEXUS_Message_Open – allocates message capture buffer
- NEXUS_Surface_Create – allocates graphics memory

Other API's include NEXUS_Dma_Open, NEXUS_PictureDecoder_Open, NEXUS_Graphics2D_Open and more.

For each API, there is a default buffer size. You may want to examine those sizes and determine if they are correct for you. They usually depend on maximum bit rate and system latency.

If you do not use features at the same time, consider using NEXUS_Memory_Allocate, then passing in a user-allocated buffer pointer to each interface when it is used.

Consider using smaller graphics framebuffers and upscaling with the GFD.

Reduce System Memory

Nexus' use of system memory is usually many small allocations; therefore the OS memory manager must implement an algorithm to avoid fragmentation. Nexus does not allocate a large amount of system memory. Therefore this is usually not a large concern, but can be of some benefit.

The method to monitor system allocations varies per OS. For Linux, you can use `cat /proc/meminfo` or run `top`.

Nexus does not call malloc or kmalloc directly. Instead, it calls Magnum's `BKNI_Malloc`. You can add debug code into KNI (refer to magnum/basemodules/kni) to monitor Nexus and Magnum `BKNI_Mallocs`.

One way to reduce system memory use is to reduce Nexus code size. This can be done by limiting features. Some specific ideas are:

- Compile in compact, error-only mode. That is export B_REFSW_DEBUG_LEVEL=err B_REFSW_DEBUG_COMPACT_ERR=y. See bdbg.inc for a description. We do not recommend compiling in release mode (export B_REFSW_DEBUG=n) because you will get no error messages.

- Modify nexus_platform_features.h and reduce the numbers to what you're actually using. Set unused features to 0.

- Remove unused modules from platform_modules.inc.

Monitoring Allocations

After doing the configuration described above, you still need to verify the allocations that are being performed. There are several methods.

You can monitor all heap allocations using the DBG interface. Run with:

```
export msg_modules=BMEM
```

You will see each allocation and free printed on the console, along with a summary of heap usage.

```
--- 00:00:00.131 BMEM: BMEM_Heap_TagAlloc(0x2bee8000 (0x2bee8000), 8192, 8,
magnum/portinginterface/xpt/7425/bxpt_xcbuf.c, 873)
--- 00:00:00.473 BMEM: BMEM_Heap_TagAlloc(0x2bee8000 (0x2bee8000), 8192, 8,
magnum/portinginterface/xpt/7425/bxpt_xcbuf.c, 873)
--- 00:00:00.473 BMEM: BMEM_Heap_TagAlloc(0x2bee8000 (0x2bee8000), 204800,
8, magnum/portinginterface/xpt/7425/bxpt_xcbuf.c, 873)
--- 00:00:00.474 BMEM: BMEM_Heap_TagAlloc(0x2bee8000 (0x2bee8000), 204800,
8, magnum/portinginterface/xpt/7425/bxpt_xcbuf.c, 873)
--- 00:00:00.475 BMEM: BMEM_Heap_TagAlloc(0x2bee8000 (0x2bee8000), 204800,
8, magnum/portinginterface/xpt/7425/bxpt_xcbuf.c, 873)
--- 00:00:00.476 BMEM: BMEM_Heap_TagAlloc(0x2bee8000 (0x2bee8000), 204800,
8, magnum/portinginterface/xpt/7425/bxpt_xcbuf.c, 873)
```

You should copy-and-paste this information to a document, then analyze every large allocation. You may need to go to the file and line number of the code to learn what the meaning of the allocation is. It is important that you be able to understand the meaning of the various allocations and correlate them with your requirements.

Be aware that Magnum sub-heaps are often created within top-level heaps. These are for internal-use only. Take note of the heap pointer to avoid double counting allocations.

You can also see the heaps at init-time with

```
export msg_modules=nexus_platform_settings
```

Output looks like this:

```
nexus_platform_settings: request heap[0]: MEMC0/0, size -1, eFull
nexus_platform_settings: request heap[1]: MEMC1/0, size 67108864, eDeviceOnly
nexus_platform_settings: request heap[2]: MEMC0/1, size -1, eApplication
nexus_platform_settings: creating heap[0]: MEMC0, offset 0x4000000, size 201326592,
eFull
nexus_platform_settings: creating heap[1]: MEMC1, offset 0x60000000, size 67108864,
eDeviceOnly
```

You can also get heap status at runtime with the /proc interface. In user mode, `echo core >/proc/bcmdriver/debug`. In kernel mode, `cat /proc/brcm/core`. Output looks like this:

```
--- 00:00:02.184 nexus_core: Core:
--- 00:00:02.184 nexus_core: heap offset     size       mapping used peak largestavail
--- 00:00:02.185 nexus_core: 0    0x04000000 0x0c000000 eFull    49%  49% 0x060d2cf8
```

Your program can also access the heaps by calling NEXUS_Platform_GetConfiguration and using NEXUS_PlatformConfiguration.heap[] and NEXUS_Heap_GetStatus.

Memory Mapping Limitations

32 Bit Limitations

There are two difficulties created by having a 32 bit system.

First, all HW cores, CPU and non-CPU, can only physically address up to a 4GB maximum. We also have to subtract physical address space for non-DRAM uses like registers and memory-mapped PCI buses. So the actual maximum is slightly less than 4GB.

Second, CPU access requires virtual addresses. We are also limited to a 32 bit virtual address space. However, the MIPS CPU divides this space into two 2GB regions: the lower 2GB for user space, the upper 2GB for kernel space. So the actual limitation is 2GB. If we want both cached and uncached access, that further limits the total addressable memory.

In order to meet all needs, Nexus provides explicit memory mapping control for all heaps. This adds complexity, but allows us to run a system with as close as possible to the theoretical 4GB max.

RTS Limitations for dual MEMC systems

In order to guarantee sufficient memory bandwidth to all hardware clients, some hardware clients are restricted to only using one of two memory controllers. This means that the assignment of heaps to Nexus interfaces is sometimes dependent on this RTS programming. This can be true for a variety of hardware cores, but usually affects only graphics and video cores because they are memory bandwidth intensive.

When programming the System Memory Worksheet, you will need to configure picture buffer heaps on both MEMC0 and MEMC1. Then, you will need to program which video path (display 0 or 1, window 0 or 1) is assigned to which heap. This assignment is based on the RTS configuration.

The graphics feeder (GFD) for a display may also be restricted to only one MEMC. Each display's GFD may have different MEMC restrictions. To facilitate this, Nexus has a function which returns a heap which is accessible by the GFD for a particular display.

```
NEXUS_Platform_Init(NULL);
NEXUS_Platform_GetConfiguration(&platformConfig);
NEXUS_Surface_GetDefaultCreateSettings(&createSettings);
createSettings.pixelFormat = NEXUS_PixelFormat_eA8_R8_G8_B8;
createSettings.width = 720;
createSettings.height = 480;
/* get a heap accessible by GFD0 for HD display */
createSettings.heap = NEXUS_Platform_GetFramebufferHeap(0);
```

```
surface = NEXUS_Surface_Create(&createSettings);
```

For MPEG feeder (MFD) access, each platform has a platform-specific file (for example, nexus/platforms/97425/src/nexus_platform_97425.c) which sets heaps for the decoder and display based on RTS. You can customize the heaps for your platform by modifying this file, and you may have a new RTS plan which changes the requirements, but the reference release should have default heaps which maps to the default RTS requirements. So, please use this file as a guide for how that heap mapping should be done.

NEXUS_Platform_GetFramebufferHeap is not available for clients. Typically, clients don't have access to the framebuffer.

3D Graphics Performance Issues

We've extend the NEXUS_Platform_GetFramebufferHeap function to also supply a heap which has best performance for the 3D core. It can be obtained as follows:

```
NEXUS_Surface_GetDefaultCreateSettings(&createSettings);
createSettings.heap =
NEXUS_Platform_GetFramebufferHeap(NEXUS_OFFSCREEN_SURFACE);
surface = NEXUS_Surface_Create(&createSettings);
```

The heap provided by NEXUS_OFFSCREEN_SURFACE must meet the following requirements for optimal performance of the VC4 3D graphics core:

- Must be HD displayable (GFD0) for 3D apps which program the framebuffer directly
- Must not cross a 256MB memory boundary (for VC4 binner addressing limit)
- Must not have BMEM guard banding turned on (which can slow down a system with lots of allocations)
- Must have application cached memory mapping (memoryType & NEXUS_MEMORY_TYPE_APPLICATION_CACHED == true)
- For V3D variants prior to 7425 B2, must not cross a 1GB memory boundary
- Must be at least 64MB in size

Debugging Memory Management Failures

Your first encounter with Nexus memory configuration may be trying to figure out what just went wrong. The following are typical problems and recommended actions.

Out of Memory

Out of system memory – a malloc or kmalloc from the kernel has failed. Nexus will try to recover, but the system is likely going to fail. The following actions are recommended:

- If the failure occurring with BKNI_Malloc, the BKNI_TRACK_MALLOCS feature will help pinpoint memory usage. If you are not using BKNI_Malloc, consider using it.
- Find and fix a memory leak, either in your application or in nexus/magnum.
- If there is no leak, you must reduce memory usage or give the kernel more memory.

Out of device memory – a heap allocation has failed. The heap may have run out of space or you may be allocating from the wrong heap.

- BMEM will print the size of the failed allocation and the line number of code. Inspect the code and the size and see if it makes sense.
- Recreate the failure using export msg_modules=BMEM. Evaluate each heap allocation and determine if it is correct.
- Inspect your heap layout in nexus_platform_$(NEXUS_PLATFORM).c. For instance, if your allocation failed in the picture buffer heap, you may need to consult the XVD/VDC allocation spreadsheet.
- Inspect your use of the nexus API. Have you set a heap parameter incorrectly? Or are you using a default heap parameter and must set one?
- See next section on memory fragmentation.

Memory Fragmentation

If a heap memory allocation fails, you should compare NEXUS_MemoryStatus.free and NEXUS_MemoryStatus.largestFreeBlock. If you have a large amount free, but the largestFreeBlock is too small, you have memory fragmentation.

Memory fragmentation is a difficult problem because it may only be detected after long use and there are few good solutions once it occurs.

To avoid fragmentation we recommend:

- Avoid calling Close functions after system init time. Instead, Open interfaces at system init time and leave them open.

- If you must close interfaces at run time, consider using NEXUS_Memory_Allocate at init time, then passing in a user-allocated buffer at run time. For example, see NEXUS_SurfaceCreateSettings.pMemory.

- Writing application code to divide the default heaps created by your platform into smaller heaps. Spread your allocations into different heaps based on size, or reallocation characteristics, or whatever else helps you avoid fragmentation.

- If you do run out of memory due to fragmentation, write your application to do a graceful shutdown and restart. This will naturally defragment the memory, but there will be an interruption to the user experience. You may want to monitor your helps and do a pre-emptive restart to avoid running out of memory in an inopportune context that can't easily recover.

Inaccessible Memory

Some failures result from the CPU trying to access memory that has no memory mapping.

Client-side invalid memory – the heap does not have NEXUS_MemoryType_eApplication mapping for client-side CPU access.

- If Nexus detected the error, you will get a clean error message with the line number of code. Inspect the code and see if it makes sense. If Nexus does not, you may just get a segmentation fault. Use a core dump and stack trace to get the same information.
- Recreate the failure using export msg_modules=nexus_platform_settings. Determine if the heap you are using has eApplication memory mapping.
- You may need to modify nexus_platform_$(NEXUS_PLATFORM).c, or you may need to use a different heap.

Server-side invalid memory – the heap does not have NEXUS_MemoryType_eDriver mapping for driver-side CPU access.

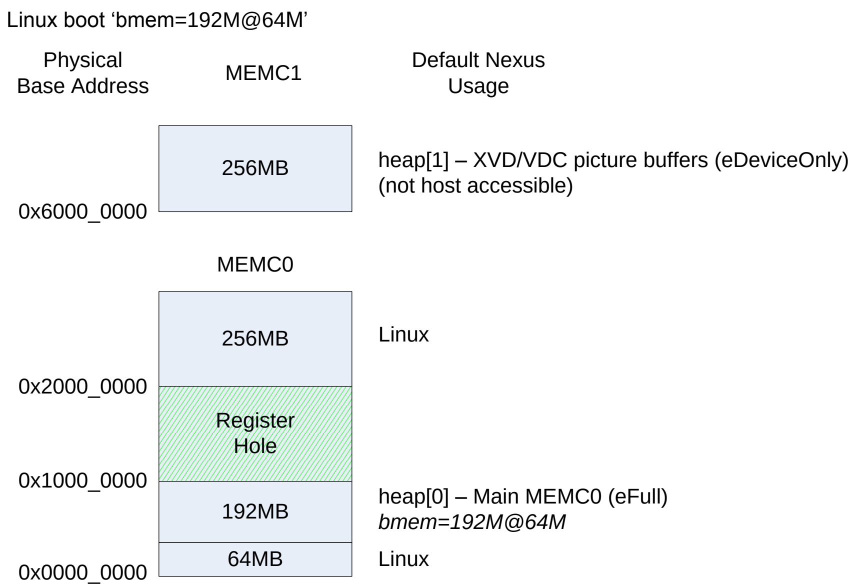
- Recreate the failure using export msg_modules=nexus_platform_settings. Determine if the heap you are using has eApplication memory mapping.
- You may need to modify nexus_platform_$(NEXUS_PLATFORM).c, or you may need to use a different heap.

Default Memory Mapping per Platform

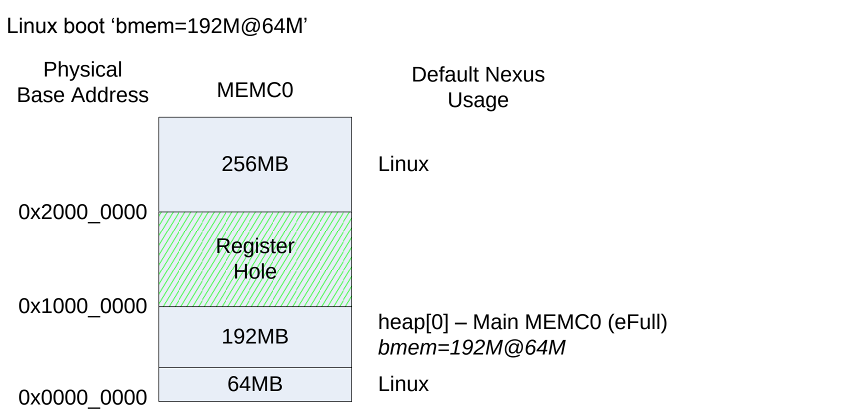
The following diagrams show the Nexus heap layout, physical base address, and typical usage for heap memory. Customers are free to modify this mapping, but will need to understand substantially more about Nexus and Magnum memory management.

> ***NOTE:*** *The following mapping diagrams are for example only. The actual numbers and layout may change over time. Your software may not actually map memory this way. Please run with export msg_modules=nexus_platform_settings to see your own mapping.*

97405 Non-UMA with 512MB MEMC0/256MB on MEMC1

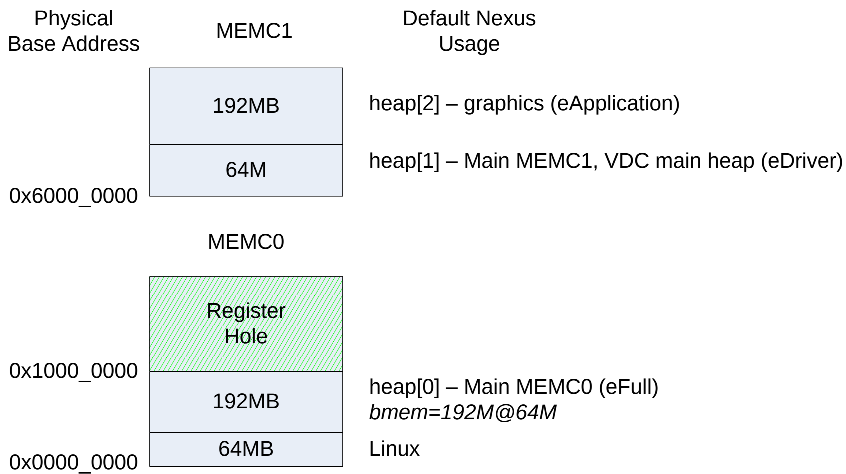


97405 UMA with 512MB MEMC0



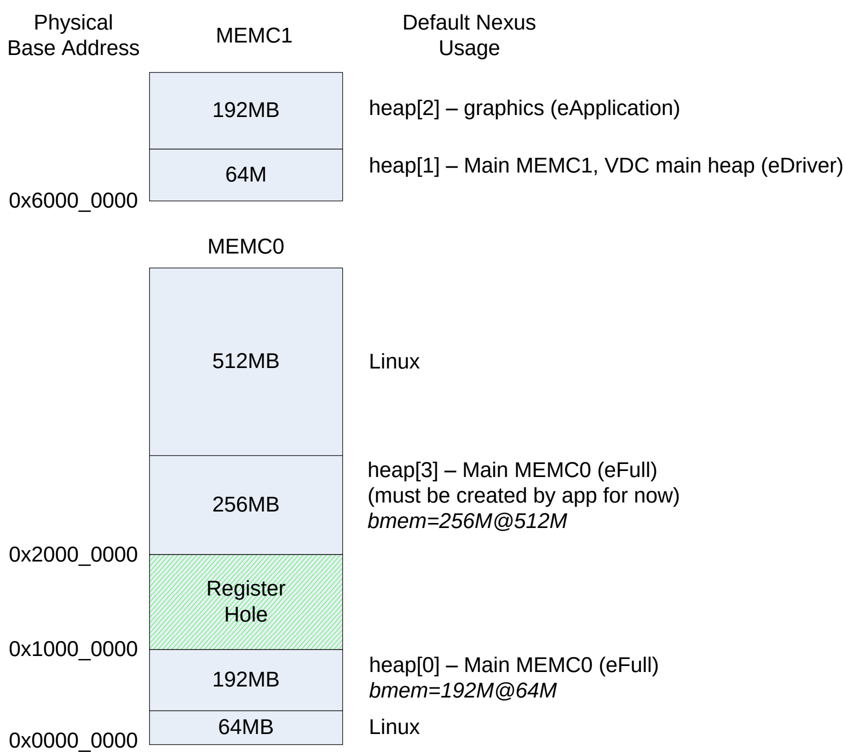
97420 with 256MB on MEMC0/256MB on MEMC1

Linux boot 'bmem=192M@64M'



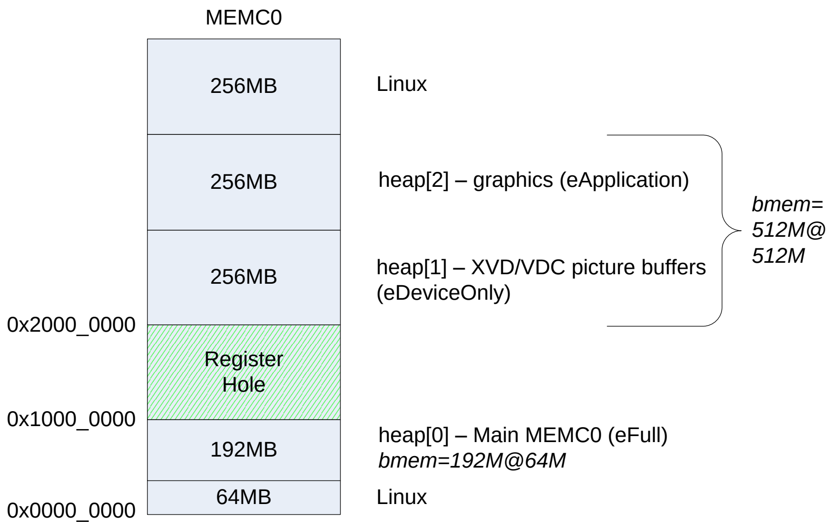
97420 with 1GB on MEMC0/256MB on MEMC1

Linux boot 'bmem=192M@64M bmem=256M@512M'

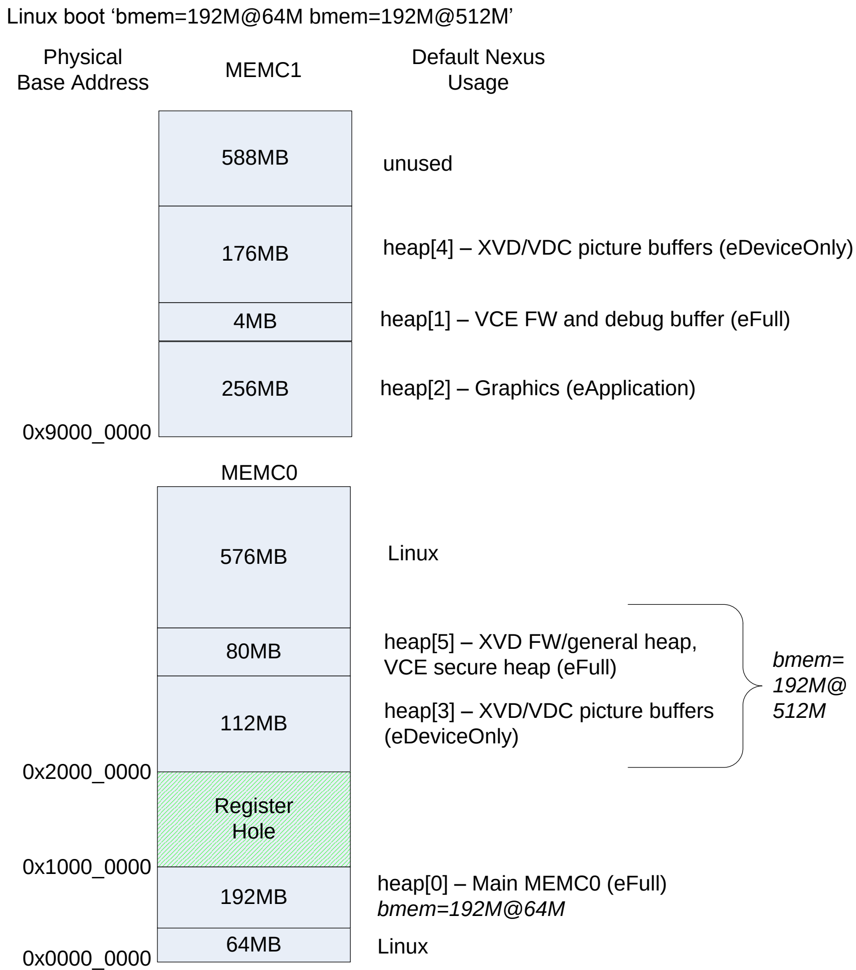


97231/97344/97346 with 1GB on MEMC0

Linux boot 'bmem=192M@64M bmem=512M@512M'



97425 B1 with 1GB on MEMC0/1GB on MEMC1



Appendix: Terminology

The following terms are used within this document and in Broadcom source code that may not be widely known. Many terms listed here may not be necessary to understand in order to use Nexus, but are included for completeness.

Term	Definition
Device Memory	Memory allocated from Nexus/Magnum heap. Contrast with system memory.
Dynamic Mapping	A dynamic TLB entry must be created for the CPU to access memory. This is the only way to access memory in user mode (mmap system call). It is required for access to high memory in kernel mode (ioremap kernel function). Contrast with fixed mapping.
Fixed mapping	Linux kernel has pre-assigned addresses for cached and/or uncached access to memory. Contrast with dynamic mapping. Same as “zone normal” and “low mem”
High Mem	Memory that requires dynamic mapping in the kernel (ioremap kernel function) New for 7420 and following.
Low Mem	Memory with fixed mapping in the kernel. Same as “zone normal”. **CAUTION:** Low Mem memory can still be **above** the register hole.
MEMC	Memory Controller. In Nexus/Magnum, DDR blocks are typically referred to by the MEMC that controls access to them. For example, MEMC0 is synonymous with DDR0.
MEMC Region	A contiguous physical addressing range within a single MEMC. Nexus refers to MEMC regions has NEXUS_PlatformSettings.heap[].subIndex. The register hole creates two regions on MEMC0.
Offset	In general, any offset from a base address. However, “offset” is often used synonymously with physical address. This is a valid use only if the “base” is understood to be physical address 0x0. Otherwise the term “physical address” is preferred.
Physical Address	Address used on memory bus to access memory. Not directly usable by the CPU. Contrast with virtual address.
Register hole	A region of physical addresses reserved for register access, not memory access. For many chips, this is physical address 0x1000_0000 – 0x2000_0000 which creates two “MEMC regions” on MEMC0.
Strapping Option	The machine-readable board configuration which tells the OS and Nexus how much memory is available on each MEMC. See nexus_platform_$(NEXUS_PLATFORM).c for code.
System Memory	Memory allocated from operating system. Contrast with device memory.
TLB	Translation Lookaside Buffer. An entry in the page table using for dynamic mapping of virtual address to physical address.
Upper memory	“Zone normal” memory above the register hole **CAUTION:** There is some “Upper Memory” which is “Low Mem”
Virtual Address	An address usable by the CPU to access memory. May be fixed or dynamic mapping. Contrast with physical address.

XKS01	Feature on 40nm silicon which allow kernel to access a full 1GB without HIGHMEM. See 2.6.37 memory app note.
Zone HighMem	Linux term for memory without fixed mapping. Requires dynamic mapping.
Zone Normal	Linux term for memory with fixed mapping