USER'S MANUAL

Nema Pico Platform Drivers

A Comprehensive Overview

A-SOCAPG-UMGA07EN v1.0

Legal Information and Disclaimers

AMBIQ MICRO INTENDS FOR THE CONTENT CONTAINED IN THE DOCUMENT TO BE ACCURATE AND RELIABLE. THIS CONTENT MAY, HOWEVER, CONTAIN TECHNICAL INACCURACIES, TYPOGRAPHICAL ERRORS OR OTHER MISTAKES. AMBIQ MICRO MAY MAKE CORRECTIONS OR OTHER CHANGES TO THIS CONTENT AT ANY TIME. AMBIQ MICRO AND ITS SUPPLIERS RESERVE THE RIGHT TO MAKE CORRECTIONS, MODIFICATIONS, ENHANCEMENTS, IMPROVEMENTS AND OTHER CHANGES TO ITS PRODUCTS, PROGRAMS AND SERVICES AT ANY TIME OR TO DISCONTINUE ANY PRODUCTS, PROGRAMS, OR SERVICES WITHOUT NOTICE.

THE CONTENT IN THIS DOCUMENT IS PROVIDED "AS IS". AMBIQ MICRO AND ITS RESPECTIVE SUPPLIERS MAKE NO REPRESENTATIONS ABOUT THE SUITABILITY OF THIS CONTENT FOR ANY PURPOSE AND DISCLAIM ALL WARRANTIES AND CONDITIONS WITH REGARD TO THIS CONTENT, INCLUDING BUT NOT LIMITED TO, ALL IMPLIED WARRANTIES AND CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHT.

AMBIQ MICRO DOES NOT WARRANT OR REPRESENT THAT ANY LICENSE, EITHER EXPRESS OR IMPLIED, IS GRANTED UNDER ANY PATENT RIGHT, COPYRIGHT, MASK WORK RIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT OF AMBIQ MICRO COVERING OR RELATING TO THIS CONTENT OR ANY COMBINATION, MACHINE, OR PROCESS TO WHICH THIS CONTENT RELATE OR WITH WHICH THIS CONTENT MAY BE USED.

USE OF THE INFORMATION IN THIS DOCUMENT MAY REQUIRE A LICENSE FROM A THIRD PARTY UNDER THE PATENTS OR OTHER INTELLECTUAL PROPERTY OF THAT THIRD PARTY, OR A LICENSE FROM AMBIQ MICRO UNDER THE PATENTS OR OTHER INTELLECTUAL PROPERTY OF AMBIQ MICRO.

INFORMATION IN THIS DOCUMENT IS PROVIDED SOLELY TO ENABLE SYSTEM AND SOFTWARE IMPLEMENTERS TO USE AMBIQ MICRO PRODUCTS. THERE ARE NO EXPRESS OR IMPLIED COPYRIGHT LICENSES GRANTED HEREUNDER TO DESIGN OR FABRICATE ANY INTEGRATED CIRCUITS OR INTEGRATED CIRCUITS BASED ON THE INFORMATION IN THIS DOCUMENT. AMBIQ MICRO RESERVES THE RIGHT TO MAKE CHANGES WITHOUT FURTHER NOTICE TO ANY PRODUCTS HEREIN. AMBIQ MICRO MAKES NO WARRANTY, REPRESENTATION OR GUARANTEE REGARDING THE SUITABILITY OF ITS PRODUCTS FOR ANY PARTICULAR PURPOSE, NOR DOES AMBIQ MICRO ASSUME ANY LIABILITY ARISING OUT OF THE APPLICATION OR USE OF ANY PRODUCT OR CIRCUIT, AND SPECIFICALLY DISCLAIMS ANY AND ALL LIABILITY, INCLUDING WITHOUT LIMITATION CONSEQUENTIAL OR INCIDENTAL DAMAGES. "TYPICAL" PARAMETERS WHICH MAY BE PROVIDED IN AMBIQ MICRO DATA SHEETS AND/OR SPECIFICATIONS CAN AND DO VARY IN DIFFERENT APPLICATIONS AND ACTUAL PERFORMANCE MAY VARY OVER TIME. ALL OPERATING PARAMETERS, INCLUDING "TYPICALS" MUST BE VALIDATED FOR EACH CUSTOMER APPLICATION BY CUSTOMER'S TECHNICAL EXPERTS. AMBIQ MICRO DOES NOT CONVEY ANY LICENSE UNDER NEITHER ITS PATENT RIGHTS NOR THE RIGHTS OF OTHERS. AMBIQ MICRO PRODUCTS ARE NOT DESIGNED, INTENDED, OR AUTHORIZED FOR USE AS COMPONENTS IN SYSTEMS INTENDED FOR SURGICAL IMPLANT INTO THE BODY, OR OTHER APPLICATIONS INTENDED TO SUPPORT OR SUSTAIN LIFE, OR FOR ANY OTHER APPLICATION IN WHICH THE FAILURE OF THE AMBIQ MICRO PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR. SHOULD BUYER PURCHASE OR USE AMBIQ MICRO PRODUCTS FOR ANY SUCH UNINTENDED OR UNAUTHORIZED APPLICATION, BUYER SHALL INDEMNIFY AND HOLD AMBIQ MICRO AND ITS OFFICERS, EMPLOYEES, SUBSIDIARIES, AFFILIATES, AND DISTRIBUTORS HARMLESS AGAINST ALL CLAIMS, COSTS, DAMAGES, AND EXPENSES, AND REASONABLE ATTORNEY FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PERSONAL INJURY OR DEATH ASSOCIATED WITH SUCH UNINTENDED OR UNAUTHORIZED USE, EVEN IF SUCH CLAIM ALLEGES THAT AMBIQ MICRO WAS NEGLIGENT REGARDING THE DESIGN OR MANUFACTURE OF THE PART.

Revision History

Rev	Date	Description
1.0	December 2025	Initial Release

Reference Documents

Document ID	Description
A-SOCAPG-UMGA01EN	Nema DC API Library User's Manual
A-SOCAPG-UMGA02EN	Nema GFX API Library User's Manual
A-SOCAPG-ANGA01EN	Nema DC MiP Panels Configuration Application Note
A-SOCAPG-ANGA02EN	Nema GFX Extensions TSVG Supported Elements List Application Note
A-SOCAPG-SGGA01EN	Nema Pixpresso Starting Guide
A-SOCAPG-ANGA03EN	Nema GFX API Debugging Application Note
A-SOCAPG-UMGA04EN	Nema GUI Builder User's Manual
A-SOCAPG-UMGA05EN	Nema GFX Benchmark Suite User's Manual
A-SOCAPG-UMGA06EN	Nema Pico Graphics Processing Unit User's Manual
A-SOCAPG-UMGA03EN	Nema GFX Extension Vector Graphics User's Manual

Table of Contents

SECTION

1

Preface

1.1 Introduction

This manual describes the parts of low level SW which are necessary to drive Nema Pico GPU. More precisely, this document describes the lower-end of Nema GFX as well as the Nema Pico kernel drivers where an Operating System is available.

1.2 Purpose

This document is intended to be used as a reference manual of the available platform drivers of Nema Pico GPU.

1.3 Scope

This manuals contains a comprehensive overview of Nema GFX lower-end (such as HAL, Platform Definition, Interrupt Handling) as well as a detailed description of the Nema kernel driver, necessary in systems with Linux OS.

1.4 Audience

This manual is intended for engineers who want to port Nema GFX and other SW layers on top of Nema Pico in a particular platform (either Bare Metal or with OS).

SECTION

2

Nema GFX Platform Porting

Nema GFX Library has been designed to be easily portable in a variety of different platforms. This includes systems with or without an operating system. In order to port Nema GFX successfully, one must take into account the target platform and adapt the HAL (Hardware Abstraction Layer) accordingly.

The HAL is a thin layer of the library that:

- Communicates directly with the system hardware and the GPU drivers (when drivers are available)
- Performs the communication between the host and the GPU (access to the GPU registers)
- Handles interrupts
- Implements the memory management scheme (memory allocation, deallocation, mapping and unmapping).

2.1 Platform Specific HAL

Each target platform has:

- A **nema_sys_defs.h** header file located in **NemaGFX_SDK/platforms/PLATFORM/common/** folder
- A separate **nema_hal.c** file, located in **NemaGFX_SDK/platforms/PLATFORM/NemaGFX** folder
- In order to port Nema GFX to a new platform, it is advised that the corresponding source files of an already ported platform are used as templates.

2.2 nema_sys_defs.h

nema_sys_defs.h contains all the global definitions and inclusions that are platform specific. For example, Nema GFX uses a set of integer types defined inside the C standard **stdint.h** header. If the platform's compiler supports the **stdint.h**, then it should be included in the **nema_sys_defs.h** header file. If the compiler does not support the stdint.h, then the following types should be defined:

- int8_t, uint8_t,
- int16_t, uint16_t,
- int32_t, uint32_t,
- int64_t, uint64_t

If Nema GFX is compiled for a platform that runs multiple processes and/or multiple threads, the following definitions should be added respectively:

```
#define NEMA_MULTI_PROCESS
#define NEMA_MULTI_THREAD
```

2.3 nema_hal.c

nema_hal.c contains all the platform specific functions that implement hardware register read/write operations, interrupt handling, memory management and mutex support. It acts as a thin layer which incorporates all the platform dependent portions of a Nema GFX implementation.

2.3.1 System Initialization

The nema_init() function initializes the Nema GFX and calls the **nema_sys_init()** function which is responsible for the system initialization. The system initialization includes:

- GPU register memory mapping
- Graphics Memory mapping
- Mutex initialization
- Ring Buffer allocation and initialization

2.3.2 Registers Read/Write

The host CPU writes to and reads from the GPU's configuration registers. The functions **nema_reg_read()** and **nema_reg_write()** are used for the communication of the CPU with the GPU.

When the target platform does not need memory virtualization (e.g., BareMetal systems), the access to the GPU's registers is straightforward by using the register's physical memory address. The only prerequisite in this case, is the appropriate memory mapping of the GPU registers to the system's main memory.

The following examples illustrate the register read and write operations for BareMetal systems.

```
uint32_t  nema_reg_read(uint32_t reg) {
    uint32_t *ptr = (uint32_t *)(nema_regs + reg);
    return *ptr;
}
void nema_reg_write(uint32_t reg, uint32_t value) {
    uint32_t *ptr = (uint32_t *)(nema_registers_base_addr + reg);
    *ptr = value;
}
```

On platforms that support virtual memory (e.g., Linux, Android), the GPU registers' physical addresses must be mapped to the virtual memory addresses before an access is attempted. This can be performed in three ways.

The first one is to utilize the GPU driver (if applicable), and perform the memory mapping using the **mmap** system call.

```
nema_regs_base_virt = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE,
MAP_SHARED, nema_fd,  0);
```

The second way in Linux systems, is to use the **/dev/mem** instead of the **uNema** driver as shown in the following example.

```
 // Map and initialize Graphics Memory
    nema_devmem_mmap(&gmem_base_virt, gmem_base_phys, gmem_size,
   "VideoMemory");
    // Memory map nema registers
    nema_devmem_mmap((void **)&nema_regs, nema_regs_base_phys,
   0x1000, "NemaRegisters");
```

The third way in Linux systems, is to perform the read/write operations to the registers via respective IOCTL calls to the **uNema** driver.

```
uint32_t  nema_reg_read(uint32_t reg) {
unema_ioctl_readwrite_t ioctl_rw;
    ioctl_rw.reg = reg;
    ioctl(nema_fd, UNEMA_IOCTL_REG_READ, &ioctl_rw);
    return ioctl_rw.value;
}
void nema_reg_write(uint32_t reg, uint32_t value) {
    unema_ioctl_readwrite_t ioctl_rw;
    ioctl_rw.reg   = reg;
    ioctl_rw.value = value;
    ioctl(nema_fd, UNEMA_IOCTL_REG_WRITE, &ioctl_rw);
}
```

2.3.3 Interrupt Handling

The interrupt handler is executed when an interrupt is triggered by the GPU. Its purpose is to awaken suspended processes and clear the interrupt.

When the GPU kernel driver is available (Linux and Android systems), the interrupt handler is defined in the **uNema** driver. When the GPU kernel driver is not available (BareMetal and RTOS based systems), the interrupt handler has to be defined in the **nema_hal.c** file.

A typical interrupt handler for BareMetal systems is the following:

```
void irq_handler (void)
{
      //Clear the interrupt    nema_reg_write(NEMA_INTERRUPT, 0);
}
```

The HAL should implement the **nema_wait_irq** function for interrupt handling. Its purpose is to suspend the process (put to sleep) if the GPU is idle or until the GPU signals an interrupt. Its implementation is platform (CPU) dependent. If the kernel driver is available, then IOCTL calls are used, otherwise it is manually defined according to the CPU platform.

```
void nema_wait_irq(void)
{
    ioctl(nema_fd, UNEMA_IOCTL_WAIT_IDLE);
}
```

Typically, the GPU will raise an interrupt when it has finished executing a Command List. The ID of the last Command List that has been executed can be read from the Configuration Register **NEMA_CLID**. The function **nema_wait_irq_cl(int cl_id)** should wait until the content of the **NEMA_CLID** register is greater or equal than the **cl_id** argument.

```
void nema_wait_irq_cl(int cl_id)
{
    while ( nema_reg_read(NEMA_CLID) < cl_id) {
        nema_wait_irq();
    }
}
```

2.3.4 Memory Management

At this stage, the memory management scheme should be implemented. Memory can be considered to consist of two parts: host memory (memory available only to the CPU) and graphics memory (memory available both to the GPU and the CPU).

Host memory can be allocated by using systems' default malloc method:

```
//System malloc
void  nema_host_free     (void *ptr ) {
    free(ptr);
}
void *nema_host_malloc   (unsigned size) {
    return malloc(size);
}
```

If such a method is not available, the graphics memory allocator can be used:

```
//Think Silicon's graphics memory allocator
void  nema_host_free     (void *ptr ) {
    tsi_free(ptr);
}
void *nema_host_malloc   (unsigned size) {
    return tsi_malloc(size);
}
```

Graphics memory is the portion of system memory that the GPU is allowed to have access. In this case, it is necessary that the graphics memory occupies a contiguous physical memory space. On systems that do not support virtual memory (e.g., BareMetal systems), graphics memory can be allocated in the same way as host memory (using malloc, **tsi_malloc** or other custom memory allocator).

Nema GFX includes the following API calls for memory allocation, deallocation and mapping:

- **nema_buffer_create()** - Allocate memory
- **nema_buffer_create_pool()** - Allocate memory from specific memory pool
- **nema_buffer_map()** - Map allocated memory space for CPU access
- **nema_buffer_unmap()** - Unmap previously mapped memory space
- **nema_buffer_destroy()** - Deallocate memory space

A reference example that illustrates this memory allocation scheme can be found in: **Software/NemaGFX_SDK/platforms/lattice_mico32_no_OS/NemaGFX/nema_hal.c**. In this file, functions **nema_buffer_create** and **nema_buffer_destroy** adopt this specific scheme.

When virtual memory is used (e.g., Linux or Android systems), graphics memory should be allocated by the system's contiguous memory allocator (e.g., ION for Android, CMA for Linux).

Linux based system: The uNema kernel driver pre-allocates a contiguous physical memory space using Linux CMA. When Nema GFX is initialized, this memory space is mapped to the process' virtual space. Graphics memory is then managed by Think Silicon's custom memory allocator (**tsi_malloc**/**tsi_free**). The code that

implements this specific memory management scheme can be found in: **Software/NemaGFX_SDK/platforms/zc70x_linux_ioctl/NemaGFX/nema_hal.c**.

Android based system: When Nema GFX is initialized, the ION kernel module is opened. On each subsequent graphics memory allocation or deallocation (**nema_buffer_create** or **nema_buffer_destroy**), the corresponding IOCTL to the ION module is called. This implementation can be found in: **Software/NemaGFX_SDK/platforms/NemaGFX/zc70x_android**.

Nema GFX supports multiple memory pools. This can be useful for systems with nonuniform memory hierarchy that have different characteristics (e.g. latency, throughput, etc). If such a feature is not needed, **nema_buffer_create_pool()** can be set to redirect to **nema_buffer_create()**:

```
nema_buffer_t nema_buffer_create_pool(int pool, unsigned size) {
#ifdef NEMA_MULTI_MEM_POOLS //defined in nema_sys_defs.h
nema_mutex_lock(MUTEX_MALLOC);
    nema_buffer_t bo;
    bo.base_virt = tsi_malloc_pool(pool, size);
    bo.base_phys = (uint32_t) tsi_virt2phys(bo.base_virt);
    bo.size      = size;
    bo.fd        = 0;
    nema_mutex_unlock(MUTEX_MALLOC);
    return bo;
#else
    return nema_buffer_create(size);
#endif
}
```

2.3.5 Support for Multi-Process Multi-Threaded Systems

Nema GFX is designed to support a wide variety of systems, from BareMetal to Linux platforms. These systems might also support multiple processes and/or multiple threads within a process. Nema GFX is a graphics API that can manage resource sharing among multiple processes/threads if needed, using mutices and thread local storage (TLS). To support multiple processes, only mutices are necessary. For multiple threads, both mutices and TSL are needed.

The most obvious shared resource is the GPU itself. Multiple processes/threads send work to the GPU using Command Lists (CL). When a CL is executed by the GPU, it is guaranteed that it will not be interrupted by another CL. Each CL also needs to set the entire state of the GPU and not rely on previous CLs. So the only thing that needs to be taken care of, when multiple processes/threads are running, is the submission of a CL for execution. In this case, a simple mutex is used by the library.

The same applies for memory management. When a buffer is created or destroyed, a mutex ensures that no allocation or deallocation is performed by two processes or threads concurrently.

During system initialization, nema_init() should call **nema_sys_init()** only once for each process. New threads within the process must not call **nema_init()** again. The **nema_sys_init()** should not reinitialize the memory allocator nor the Ring Buffer, unless no other running process performed the aforementioned initializations.

For multi-threaded environments, **TLS_VAR** should be defined inside **nema_sys_defs.h**. Nema GFX uses **TLS_VAR** as a prefix to declare thread local variables. For example, when using GCC the following definition should be added:

```
#ifdef NEMA_MULTI_THREAD
#define TLS_VAR __thread
#else
#define TLS_VAR
#endif
```

SECTION

3

Linux Kernal Driver

Think Silicon provides the GPU device driver for the Linux operating system, which enables user processes to use Nema Pico. The driver configures hardware parameters and allows communication between the host CPU and Nema Pico. The main responsibility of the driver is to properly initialize the kernel module, detect the GPU, perform the necessary setup, and handle hardware interrupts.

3.1 Requirements

- Nema Pico driver has been tested and optimized for Linux Kernel 4.19.
- Nema Pico driver has been tested on Ubuntu 16.04 OS running on top of 32-bit Arm processor

3.2 Folder Structure

The driver is located in the following folder: **Software/NemaDriver**

Nema Pico driver folder structure and a brief description are shown below:

- include
  - **nema_incl.h**: Nema Pico driver public header file.
- src
  - **build_nema_driver.sh**: Nema Pico driver building script
  - **Makefile**:
    - **nema.c**: Nema Pico driver implementation file
    - **nema.h**: Nema Pico driver internal header file
    - **README**: Nema Pico driver building instructions

3.3 Building

The driver can be built by running make, in the following path: **Software/NemaDriver/src**. The corresponding Makefile is located in the same directory. When building the driver, one can configure its platform dependent parameters by using them as CFLAGS in the Makefile. The system dependent parameters are:

- **NEMA_BASE** - defines the base physical address of the memory space used by Nema Pico (default value 0x43d00000).
- **NEMA_IRQ** - defines Nema Pico interrupt line number (default value 87). This must be unique for all the devices that use interrupts.
- **NEMA_MAJOR** - denotes the major number of the device (default value 40).
- **DONT_ALLOC_SHMEM** - if this parameter is defined, then no shared memory will be allocated (default not defined).
- **NEMA_SHMEM_SIZE** - defines the size of the shared memory (default value 32 MB).

Building the driver without any arguments will make use of its default parameters. In order to set different parameters, one has to build the driver with proper CFLAGS (declared in the Makefile), or run make with the designated CFLAGS. For instance, when the desired parameters have to be: **NEMA_BASE** value 0x68dd0000, **NEMA_IRQ** value 18, **NEMA_MAJOR** value 24 and **DONT_ALLOC_SHMEM** defined, then the driver must be built by running:

```
make CFLAGS='-DNEMA_BASE=0x68dd0000 -DNEMA_IRQ=18 -DNEMA_MAJOR=24
-DDONT_ALLOC_SHMEM'
```

3.4 Memory Allocation

Memory accesses made by Nema Pico must be done in unpaged, physical contiguous memory. When the user wishes the GPU to access graphics data (mostly texture data) this must be allocated in contiguous physical memory. However, user processes cannot allocate such memory. Calls to system's malloc will return virtual paged memory, which does not correspond to contiguous physical memory.

The driver overcomes this problem, by allocating automatically at system boot time the required memory using Linux Contiguous Memory Allocator (CMA); the size and base physical address are defined when building the driver. This is performed only when the **DONT_ALLOC_SHMEM** parameter is not defined. Consequently, the graphics memory is reserved at boot time so that user applications can access it afterwards, during their runtime.

After allocating memory, it has to be properly mapped to virtual memory space. This is performed using the mmap system call as illustrated in the following example.

```
graphics_mem_base_virt = mmap((void *)NULL, NEMA_SHMEM_SIZE,
PROT_READ|PROT_WRITE,  MAP_SHARED, nema_fd, 0);
```

3.5 Input Output Controls (IOCTLs)

Nema Pico Linux driver offers five main IOCTLs with which a user process can communicate with the Linux Driver and the GPU.

1. **NEMA_IOCTL_REG_READ**: User process requests to read a specific GPU register. Bellow follows an example about the usage of this IOCTL.

```
#include <stdint.h>
#include <stdio.h>
#include <nema_incl.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
int main(int argc, char* argv[])
{
    // Open driver
    int nema_fd = -1;
    nema_fd = open("/dev/nema", O_RDWR);
    if(nema_fd < 0) {
        printf("No Nema driver found\n");
        return -1;
}
    // declare and initialize ioctl read - write struct
    uint32_t reg = 0x118;
    nema_ioctl_readwrite_t ioctl_rw;
    ioctl_rw.reg = reg;
    // pass the initialized struct as argument to ioctl
    int ret = ioctl(nema_fd, NEMA_IOCTL_REG_READ, &ioctl_rw);
    // print the register value or inform the user for error
    if(ret >= 0)
       printf("Register 0x118 value is: %x\n", ioctl_rw.value);
    else
        printf("An error has occured\n");
    // close file descriptor
    close(nema_fd);
    return 0;
}
```

In order to access a register and its value inside the Nema GPU, a special struct is used: **nema_ioctl_readwrite_t**. This struct is composed by two fields. The first one is reg and denotes the register from which the user needs to read data and the second one is value.

The reg field is initialized (in the example register 0x118 is used) and then the struct is passed along with the respective request (**NEMA_IOCTL_REG_READ**) and the file descriptor of the Nema GPU to the ioctl function (a system call that

is handled by the uNema kernel driver). After the execution of the function, the value field has been updated with the actual value of register 0x118 and is displayed via the **printf** function.

2. **NEMA_IOCTL_REG_WRITE**: User process requests to write a specific GPU register with a value. This case is handled similarly to the previous one, with the only difference that except the reg field, the value is also set by the user.

   The respective ioctl call is then executed and value is written to the corresponding register. The result of this action can be verified using the previous case (**NEMA_IOCTL_REG_READ**) as explained earlier.

```
#include <stdint.h>
#include <stdio.h>
#include <nema_incl.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

int main(int argc, char* argv[])
{
    // Open driver
    int nema_fd = -1;
    nema_fd = open("/dev/nema", O_RDWR);
    if(nema_fd < 0) {
        printf("No Nema driver found\n");
        return -1;
}
    // declare and initialize ioctl read - write struct
    nema_ioctl_readwrite_t ioctl_rw;
    ioctl_rw.reg = 0x01c; // NEMA_TEX_COLOR
    ioctl_rw.value = 0xff00aaff;

    // write value to register
    int ret = ioctl(nema_fd, NEMA_IOCTL_REG_WRITE, &ioctl_rw);
    if(ret >= 0)
        printf("Value 0xff00aaff has been written in register
        0x01c\n");
    else
        printf("An error has occured\n");
    // close file descriptor
    close(nema_fd);
return 0;
}
```

3. **NEMA_IOCTL_GET_SHMEM_PHYS_ADDR**: Returns to the user process the base pointer of the contiguous physical memory. If **DONT_ALLOC_SHMEM** is defined, then calling this callback will return -1, otherwise it should return 0. In case that it returns a negative number, it indicates that an internal error (inside the calling function) has been occurred.

In the following example, the virtual address of the contiguous memory is returned by the mmap function. In case that the return value is invalid, an error message is displayed and the process terminates its execution.

The respective ioctl call is then executed and translates the virtual address (**vmem_base_virt**) in the corresponding physical address (**vmem_base_-phys**).

The example then finishes by printing both addresses (virtual and physical) or by informing the user that the address could not be recovered due to some error.

```
#include <stdint.h>
#include <stdio.h>
#include <nema_incl.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

int main(int argc, char* argv[])
{
    void *vmem_base_virt;
uint32_t vmem_base_phys;

    // Open driver
int nema_fd = -1;
    nema_fd = open("/dev/nema", O_RDWR);
    if(nema_fd < 0) {
        printf("No Nema driver found\n");
        return -1;
}
    // Map and initialize Graphics Memory
    vmem_base_virt = mmap((void *)NULL, NEMA_SHMEM_SIZE,
PROT_READ|PROT_WRITE,  MAP_SHARED, nema_fd, 0);

    if((int)vmem_base_virt == -1) {
        printf("Memory mapping failed, please check the platform
dependent parameters\n");
        close(nema_fd);
return -1;
}
    //get the physical address at vmem_base_phys
    int ret = ioctl(nema_fd, NEMA_IOCTL_GET_SHMEM_PHYS_ADDR,
&vmem_base_phys);

    if(ret >= 0)
        printf("Graphics memory base address: Virtual: 0x%08x,
Physical: 0x%08x
\n", (unsigned int)vmem_base_virt, vmem_base_phys);
    else
        printf("An error has occured\n");

    //close file descriptor
    close(nema_fd);
return 0;
}
```

It must be noted that the contiguous memory is allocated by the kernel driver. Its size is user defined and can be configured when building the driver.

4. **NEMA_IOCTL_WAIT_IDLE**: When a user process signals this IOCTL, puts the process to sleep until an interrupt is triggered by the GPU or the GPU is idle.

```
#include <stdint.h>
#include <stdio.h>
#include <nema_incl.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

int main(int argc, char* argv[])
{
    // Open driver
int nema_fd = -1;
    nema_fd = open("/dev/nema", O_RDWR);
    if(nema_fd < 0) {
        printf("No Nema driver found\n");
        return -1;
}
    // sleep until GPU triggeres an interrupt or GPU is idle
int ret = ioctl(nema_fd, NEMA_IOCTL_WAIT_IDLE);
    if(ret >= 0)
        printf("Process continues execution\n");
    else
        printf("An error has occured\n");
    //close file descriptor
    close(nema_fd);
 return 0;
}
```

5. **NEMA_IOCTL_PROCESS_ATTACH**: Attaches the current process to the kernel driver. This is necessary in order to ensure inter-process communication in cases whereas there are multiple processes that use the GPU.

```
#include <stdint.h>
#include <stdio.h>
#include <nema_incl.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

int main(int argc, char* argv[])
{
    // Open driver
int nema_fd = -1;
    nema_fd = open("/dev/nema", O_RDWR);
    if (nema_fd < 0) {
        printf("No Nema driver found\n");
        return -1;     }

int attached_processes;
```

```
    // attach current process to driver
    int ret = ioctl(nema_fd, NEMA_IOCTL_PROCESS_ATTACH,
&attached_processes);
    if(ret >= 0){
        if(attached_processes == 0)
            printf("This is the first process attached to the
driver\n");
        else
            printf("%d processes attached to the driver\n",
attached_processes +
 1); //+1 because numbering starts from 0
    }

    //close file descriptor
    close(nema_fd);
return 0;
}
```

An **ioctl** call, always returns a value. When this value is greater than or equal to '0', it means that the **ioctl** was executed successfully (its input parameters were consistent with the driver's parameters).

In this example, if the **ioctl** was executed successfully, the program then, prints the number of attached processes. If the returned value is negative, it means that an error has occurred during its execution.

6500 River Place Boulevard, Building 7, Suite 200, Austin, TX 78730
www.ambiq.com
sales@ambiq.com
+1 512. 879.2850

A-SOCAPG-UMGA07EN v1.0
December 2025