CTR
System Programming Guide
OS and Debug Libraries (os/dbg)

2012/06/22

Version 1.4

The content of this document is highly confidential and should be handled accordingly.

CTR-06-0034-002-H
Released: January 15, 2013

Table of Contents

Code

Figures

Tables

1 Introduction

This document describes the following, with a focus on the os and dbg libraries.

- Starting an Application
- Memory Management
- Threading
- Synchronization Mechanisms
- Time
- Debugging Output
- Exception Handlers

2 Terminology

This chapter defines the terminology used in the CTR-SDK. These definitions are specific to the CTR-SDK and may differ from the standard usage of the terms.

- os Library

  The portions of the API provided by the CTR-SDK that are included in the `nn::os` namespace.

- fnd Library

  The portions of the API provided by the CTR-SDK that are included in the `nn::fnd` namespace.

- application

  General term for software that runs on CTR.

- user application

  An application that can be created using the CTR-SDK. The term “application program” may sometimes be used to emphasize the “program” aspect of the application.

- system application

  An application that makes up the environment in which applications run and that supplements the operation of the applications.

- device

  This term refers to all of the hardware included in the CTR system, with the exception of the CPU. The GPU, DSP, microphone, accelerometer, and other such components are all considered to be devices.

- heap

  One of the two memory regions that programs can allocate dynamically.
  Memory used by system applications must use this region.

- device memory

  One of the two memory regions that programs can allocate dynamically.
  Memory used by devices must use this region.

- fnd heap

  Class for managing memory at the byte level. Included in the `fnd` library. This term also refers to the memory regions that are managed by this class.

- weak function

  A function that is marked with weak symbols.

Weakly defined functions are a compiler/linker mechanism for allowing function definitions to be overridden during linking. The CTR-SDK defines certain functions with weak symbols in order to allow them to be overridden by applications.

- A weakly defined function behaves like a normally defined function unless a non-weakly defined function of the same name is linked into the same image. However, if both a normal (non-weakly defined) function and a weakly defined function of the same name exist in the same image, all calls to the function resolve to the non-weak function. (No linker errors will occur.) If the default implementation of a function is defined with weak symbols, you can therefore override the default implementation by defining a normal (non-weak) version of that function.

- C++ static initializer

  The process defined in the C++ language standard for performing initialization prior to executing the `main` function. Its tasks include the initialization of static variables. It includes both static and dynamic initialization.

- C static initializer

  This is an independent extension of the CTR-SDK. It is a process coded in the C language that performs the equivalent operations of the C++ static initializer.

- `nnMain` function

  This function has the role of the C/C++ `main` function in the CTR-SDK.
  Although the execution of a normal C/C++ program starts from the `main` function, a CTR-SDK program starts from the `nnMain` function.

- `Start*` functions

These refer to the following four member functions of the `Thread` class: `Start`, `TryStart`, `StartUsingAutoStack`, and `TryStartUsingAutoStack`.
These member functions create new threads.

- thread function

  A function that indicates the process to execute on a newly created thread.
  Specify a thread function as an argument to a Start function to execute that function in the newly created thread.

- main thread

  This is first thread that the program creates. It is executed by the `nnMain` function.

- system core

  Of the two CPUs on the CTR system, this is the one mainly used by the system. It is core number one.

3 Startup and Initialization

3.1 Starting Applications

Application processes begin with the `_ctr_start` function.

The `_ctr_start` function is defined in `sources/libraries/crt0/MPCore/crt0.cpp`. User applications cannot change this operation. The implementation provided by the CTR-SDK must not be modified.

The `_ctr_start` function performs the following operations in the following order.

1. Clear the `.bss` section to zero.
2. Initialize the C-language floating-point runtime library.
3. Initialize the C-language locale.
4. Initialize the `os` library.
5. Call the `nninitStartUp` function.
6. Call the C++ static initializer.
7. Call the C static initializer.
8. Initialize the user application environment.
9. Call `nnMain`.

Of these operations, user applications can customize the implementations of steps (5), (6), (7), and (9). The following sections explain how to customize these steps and what types of operations are required.

In the text below, the term "static initializers" is used to refer collectively to the C++ static initializer and the C static initializer.

3.2 (Step 5) `nninitStartUp`

The `nninitStartUp` function is provided to initialize any user-application-specific memory-management systems before the static initializer is run.

The `nninitStartUp` function is not intended to be called from user applications. Rather, user applications can define their own specific `nninitStartUp` function to run any proprietary code before the static initializer is run.

3.2.1 Default `nninitStartUp`

The CTR-SDK defines the default `nninitStartUp` as a weakly defined function. You can therefore use the language's standard memory-allocation functions (like `malloc` and `new`) within the default static initializers without doing anything special. That said, the memory-management system provided

by the default nninitStartUp function is only intended for use in the initial application development phase. It will not stand up to the rigors of use by a fully developed application.

Note: Nintendo strongly recommends that application developers override the default implementation with one that is customized for the requirements of the application.

The default nninitStartUp function performs the following operations.

- Allocates 32 MB of device memory.
- Allocates all of the remaining memory that can be used by the user application as a heap.
- Enables the use of the MemoryBlock class.
- Configures the default AutoStackManager.
- Allocates an 8MB MemoryBlock, and creates an instance of fnd::ExpHeap that is made thread-safe through the use of the CriticalSection class.
- Configures subroutines such as malloc, free, new, and delete to use the aforementioned fnd heap.
- Enables the use of ManagedThread.
- Registers dbg::CTR::ExceptionScreeen as a user exception handler.
- Sets the handler that uses dbg::CTR::ExceptionScreeen as the user handler when an ASSERT fails or a PANIC occurs.

3.2.2 Overriding nninitStartUp

The default nninitStartUp function is defined with weak symbols. If you define a non-weakly defined function of the same name in your application, your custom implementation will automatically be called instead of the default nninitStartUp function.

Note that when coded in C++, the nninitStartUp function must be declared with C linkage (Code 3-1). Also note that this code will run before the static initializer is called. Keep in mind that constructors for static objects will not run until after nninitStartUp has run. To avoid risk, no operations other than the initialization of the memory-management system should be run within your nninitStartUp implementation.

Code 3-1 C Linkage Is Required in C++

```
extern "C" void nninitStartUp()
{
// Application-specific operations
}
```

If you override the default `nninitStartUp` function, you must also override the `malloc`, `free`, `new`, and `delete` subroutines. (You don't need to override these functions if your application doesn't use them.)

In the same way, you will not be able to use `Thread::StartUsingAutoStack` member function if you override the default `nninitStartUp` implementation. See 5.4 The Stack for details on `AutoStack`.

3.3 (Step 6) C++ Static Initializer

The C++ static initializer performs a series of initializations defined by the C++ language specification. By writing C++ code to run the static initializer, the operations within the initializer will run automatically. One example of a usage case is a static object that has a user-defined constructor. For details, refer to textbooks or other resources about C++.

When using a C++ static initializer with the CTR-SDK, pay special attention to the initialization of your memory-management system. Although some C++ static initializers make explicit calls to subroutines like `new` to allocate memory, others may allocate memory implicitly. Your application's memory-management system must therefore be enabled for use before the C++ static initializer is called.

The `nninitStartUp` function is provided to initialize your application's memory-management system before the C++ static initializer is called. See section 3.2 (Step 5) `nninitStartUp` for details.

3.4 (Step 7) C Static Initializer

The C static initializer is a proprietary specification of the CTR-SDK that allows static initializers to be used with the C language. This feature is similar to the one that was provided with the TWL-SDK.

You can define the C static initializer for each source code file, as shown in the sample below. The C static initializer will be called automatically after the C++ static initializer but before `nnMain`.

Although a variety of operations can be performed in the C static initializer, some of the operations will be delayed because step (8)—initialization of the user application environment—has not yet occurred. Nintendo recommends limiting the operations that are performed within the static initializer to the bare minimum required.

Code 3-2 `nninitStaticInt` Usage

```
#include <nn/sinit.h>
static void nninitStaticInit(void)
{
   // User-application-specific operations
}
```

3.5 (Step 9) nnMain

The nnMain function is the main program for CTR applications.

As it is the main function, this function must be present and contains application-specific code. If your application does not define the nnMain function, a linker error will occur.

When coding in C++, you must declare the nnMain function with C linkage. To accomplish this, you can either specify the C linkage explicitly (as shown in **Error! Reference source not found.**) or include nn.h (as shown in **Error! Reference source not found.**). Including nn.h will cause the nnMain function to be declared with C linkage.

Code 3-3 Explicitly Declaring the C Linkage

```
extern "C" void nnMain()
{
}
```

Code 3-4 Including the nn.h Header

```
#include <nn.h>
void nnMain()
{
}
```

3.6 Memory Allocation by STL

If you use STL, memory is sometimes allocated when you run new in the C++ static initializer, as mentioned in Step 6. This allocation of memory takes place in the following two cases:

- If you include <iostream>, then about 1.5KB of memory will be allocated.

  You can avoid this by including <istream> or <ostream> instead of <iostream>.

- 8 bytes of memory are allocated for initializing the runtime library

  You cannot avoid this because it occurs when various headers are included.

If you are using STL you will need to support this allocation of memory.

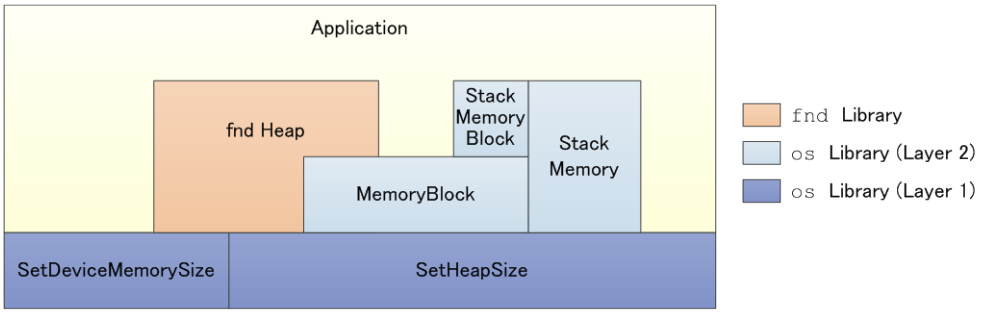
4 Memory Management

4.1 Memory-Management Systems of the CTR-SDK

Figure 4-1 shows the overall structure of the CTR-SDK's memory-management APIs.

The os library has a two-layer structure, and the fnd library sits on top of the os library's layers. Everything but the first layer of the os library allows application-specific code to be used exclusively, without using the APIs provided by the CTR-SDK.

Figure 4-1 Memory-Management API Structure



4.2 Heap and Device Memory

User applications must divide their memory into two separate regions that can be allocated dynamically: the heap and the device memory.

The main difference between the heap memory and the device memory lies in their accessibility from outside of the application. If you are using these regions solely for application-specific code, there is no significant difference between the heap and the device memory.

Use of the heap is required if memory for a system application must be allocated from within your user application.

Use of the device memory is required if memory for a device must be allocated from within your user application.

4.3 os Library

The memory-management system of the os library is implemented as a two-layer hierarchy.

4.3.1 Required Functions and Classes (Layer 1)

The first layer consists of functions and classes that must always be used. (There are no such classes at the moment.)

The main functions and classes in layer 1 are listed below.

- `SetHeapSize`
- `SetDeviceMemorySize`

The `SetHeapSize` and `SetDeviceMemorySize` functions form the core of the first layer. These functions are essential if your application allocates memory dynamically at runtime.

The `SetHeapSize` and `SetDeviceMemorySize` functions both assign contiguous blocks of memory to user applications.

In order to prevent unintentional corruption of memory managed by the default memory management system, when `SetHeapSize` and related functions are called they are stopped with an assert when using the default memory management system.

4.3.2 Optional Functions and Classes (Layer 2)

The second layer consists of functions and classes whose use is not required. They allow you to manage the heap directly within your user application's own memory-management code. The heap can also be managed directly in the same way using the `fnd` library described later in this document.

The main functions and classes in layer 2 are listed below.

- `InitializeMemoryBlock`
- `MemoryBlock`
- `StackMemoryBlock`
- `StackMemory`

The `MemoryBlock` class provides the ability to split up a heap allocated using `SetHeapSize` into 4KB chunks for use. Before using the `MemoryBlock` class, you must first call `InitializeMemoryBlock` and specify the heap region that the `MemoryBlock` instance will control.

The `StackMemoryBlock` class is a specialized version of the `MemoryBlock` class for use with stacks. It can be passed directly to the `Thread::Start` member function as a stack. Before using the `StackMemoryBlock` class, you must first call `InitializeMemoryBlock`, just as you must with the `MemoryBlock` class. The `MemoryBlock` and `StackMemoryBlock` classes are used to divide a single region of memory in the heap into smaller parts for later use.

Use of the `MemoryBlock` and `StackMemoryBlock` classes is not recommended.

The `StackMemory` class isolates the region specified within the heap to a different address. It intentionally uses memory as a stack and makes it possible to detect stack underflows and overflows as data-abort exceptions.

4.4 fnd Library

The fnd library provides several classes for managing memory at the byte level. These classes allow you to provide the starting address and size of the desired memory region. Because the os library does not provide a byte-level memory management feature, you must use the fnd library to implement memory-management systems that can be used in the same way as standard memory-allocation subroutines like malloc and new.

These classes can be used to manage the device memory and heap directly. They can also be used to manage memory regions that were allocated using the MemoryBlock class.

4.5 Default Memory-Management System

If your user application does not override the default nninitStartUp function, the default memory-management system will be set up.

See section 3.2 (Step 5) nninitStartUp for more information about the memory-management system that is set up by default.

Nintendo strongly recommends that application developers create their own user-application-specific memory-management system instead of using the default one.

5 Threading

Threading is a mechanism for creating different processing flows. The information in this chapter assumes a basic understanding of threading in general.

5.1 Creating a Thread

Threads are managed by the `Thread` class. You can create an instance of the `Thread` class, then call the `Start`, `TryStart`, `StartUsingAutoStack`, or `TryStartUsingAutoStack` function to create a thread and begin execution. These four member functions are collectively called "`Start*`" member functions.

When calling a `Start*` member function, specify a pointer to the thread function, the stack to use in the thread's process, the thread priority, and the number of the core to run it on. You can specify the stack using one of two ways: specifying the end of the memory area to use as the stack, or specifying only the size of the stack, and allocating the memory internally. See section 5.4 The Stack for details about stacks. See section 5.6 Using the System Core for information about using the system core.

You can also specify the options to pass to the thread function in the call to the `Start*` member functions. These arguments are copied to the thread's stack, and passed to the thread function as arguments. Note that the available stack size will be reduced by the size of the arguments. The caller of the `Start*` function does not need to maintain the original arguments, because they are copied.

5.2 Destroying a Thread

Threads are destroyed automatically when the thread function returns. However, the thread's system resources are not freed at the time the thread is destroyed. You must call the `Thread::Join` or `Thread::Detach` function to free these system resources.

5.3 Scheduling

The CTR system bases scheduling on priority only. At any given time, the thread capable of running that has the highest priority is the one that is run.

Although the CTR system does not perform preemption using time slices, it will preempt the currently running thread and switch to a different thread if an interrupt, timer, or other mechanism makes a high-priority waiting thread available to be run.

The system core performs a special kind of scheduling. See section 5.6 Using the System Core for details about the system core.

5.4 The Stack

5.4.1 Managing the Stack

The CTR-SDK requires the application developer to manage the thread's stack. The `Thread` class just takes the specified address as the bottom of the stack; it does not check how the stack is being managed. The application developer is responsible for ensuring that enough memory is allocated for the stack area, and for guaranteeing that this memory area will not be used for other purposes.

However, the system manages the stack of the main thread. A mechanism called "AutoStack" is also provided for stack management. These are each described in the following sections.

5.4.2 The Main Thread's Stack

The system allocates the stack for the main thread, because it must be allocated before the application's process begins. The application cannot free the memory for this stack while it is running.

You can configure the size of the main thread's stack in the RSF settings. Specify the size of the main thread's stack with the `StackSize` variable under `SystemControlInfo` of the RSF settings.

5.4.3 AutoStack

AutoStack is a mechanism for isolating the stack-management process. Specifying an instance of a class derived from `AutoStackManager` in `Thread::SetAutoStackManager` will make the `Thread::StartUsingAutoStack` and `Thread::TryStartUsingAutoStack` functions available. These functions take a stack size as an argument rather than the address of the bottom of the stack, and leave the deallocation of memory for the stack to the class derived from `AutoStackManager`, which was set in the call to `SetAutoStackManager`.

If you set an instance of a properly implemented class derived from `AutoStackManager` in `SetAutoStackManager`, you will not need to allocate or free the memory for the stack used when calling the `Thread::StartUsingAutoStack` and `Thread::TryStartUsingAutoStack` functions.

5.4.4 AutoStackManager

`AutoStackManager` is an interface class that contains two pure virtual functions and nothing else. You use it by creating a derived class that implements these pure virtual functions. You cannot create and use instances of `AutoStackManager` directly.

The CTR-SDK provides two derived classes of `AutoStackManager`:

The `SimpleAutoStackManager` class has a pointer to an instance of a derived class of `fnd::IAllocator`, and leaves actual memory management to this derived class. It is a wrapper class for using `IAllocator` as an `AutoStackManager`.

`StackMemoryAutoStackManager` is similar to `SimpleAutoStackManager`, but differs in that it uses `StackMemory` to isolate the allocated memory area. The use of `StackMemory` makes it more secure against stack overflows and underflows.

5.4.5 Implementing AutoStackManager

You can also implement your own `AutoStackManager`. Although you need only to derive a class from `AutoStackManager` and implement the pure virtual functions, there are a few potential issues you need to be aware of. In particular, great care is required for the implementation of the `Destruct` function.

5.4.5.1 The `Construct` Function

This function allocates memory for the stack, and returns the address to it.

- The function must return the address to the end of the memory region, and not to the start.
- The address returned by the function (i.e., the end of the memory region) must be eight-byte aligned. Although you can specify the alignment of the start of the memory region to functions such as `fnd::ExpHeap`, note that you cannot specify the alignment of the end of the region.

5.4.5.2 The `Destruct` Function

This function takes the address to the memory `region` allocated by `Construct`, and frees this memory area.

- It takes the address that is the value returned by the `Construct` function; this is the address to the end of the memory region. You must devise some means of obtaining the address of the start of the memory region from the end address, because allocators ordinarily require the start address of a memory region to free it.

The `Destruct` function is called when **the memory region to be freed is in use as the stack**. If you free this memory region yourself, the application will end up using a freed memory region, which can cause unexpected bugs. Before freeing the memory region, you must ensure that it is not being used as a stack.

Taking these two points into account when implementing the `Destruct` function may require knowledge of assemblers.

5.5 Thread-Local Storage

Each thread uniquely owns 16 four-byte areas of storage, which is called “thread-local storage.” Thread-local storage lets you write streamlined code to get and set different values for each thread efficiently. It is well suited to storing values that you want to change automatically each time the thread changes.

For example, you could create a different allocator for each thread, and store pointers to these allocators in thread-local storage. You could then eliminate the need for locks on your allocators by using the allocators stored in thread-local storage by calling their `new` and `malloc` functions.

Thread-local storage cannot be accessed by other threads, and each thread can only access its own thread-local storage.

Use the `ThreadLocalStorage` class to access thread-local storage.

5.6 Using the System Core

5.6.1 The System Core

The CTR system has two built-in CPU cores. The “system core” is the core actually used for processing of the functions that the CTR-SDK provides. Since CTR-SDK 3.0, a portion of the system core is available to user applications. This section describes the usage of the system core by user applications.

5.6.2 Assigning CPU Time

To use the system core, the application must first set the amount of the system core’s CPU time to assign. This value is expressed as a percentage and indicates the percentage of the system core’s CPU that will be available to the user application. The larger the percentage you set, the greater the proportion of the system core that will be available to the user application.

Note: Setting a larger percentage means that less of the system core will be available to the system. As a result, API functions will take longer to complete.

Use the `nn::os::CTR::SetApplicationCpuTimeLimit(s32)` function to assign CPU time. Passing a value of 20 to this function makes 20% of the system core available to the user application, and 80% available to the system.

Note: This means that even if your user application is not using the system core at all, it will only allow 80% to be available to the system.

When a user application first starts, 0% of CPU time is assigned to it. The user application will not be able to perform processing on the system core unless it assigns itself CPU time by calling the `SetApplicationCpuTimeLimit` function.

You can currently set a value between 5 and 30 (inclusive) in `SetApplicationCpuTimeLimit`. Once you set a non-zero value, it will no longer be possible to return the value to 0.

5.6.3 Creating a Thread

To actually perform processing on the system core, the user application must create a thread to run on it. To create a thread that will run on the system core, specify a core number of 1 when calling a `Start*` member function of the `Thread` class. You can do processing on the system core by running this thread.

You must assign CPU time to your application before creating a thread to run on the system core. The `Start*` member functions will fail if you try to create a thread that will run on the system core when 0% of CPU time is assigned to the user application.

5.6.4 Scheduling in the System Core

CTR scheduling is described in section 5.3 Scheduling, but scheduling rules have been added to implement the abovementioned assignment of CPU time from the system core.

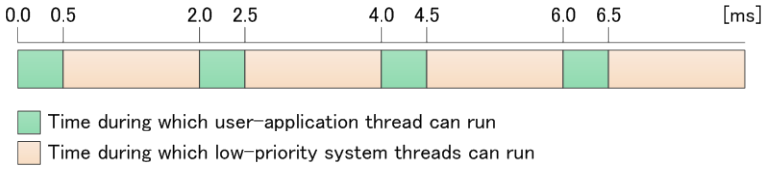
Both threads created by the user application and threads created by system applications run on the system core. Threads created by the system are further divided into two categories: high-priority system threads and low-priority system threads. The following rules apply to these categories. Assuming the percentage of CPU time assigned to the user application is A:

1. The user-application thread is run for the first (A/100) * 2.00 ms of each 2.00 ms. Low-priority system threads are not run at all.
2. Low-priority system threads are run during the remainder of each 2.00 ms from step 1, above. The user-application thread is not run at all.
3. High-priority system threads run by interrupting other threads. However, if a high-priority system thread runs during the period in step 1, above, then the period in step 1 will be extended by the time that thread is run.

High-priority system threads are mainly used to control the GPU/DSP, and sample the microphone. Other processes belong to low-priority system threads.

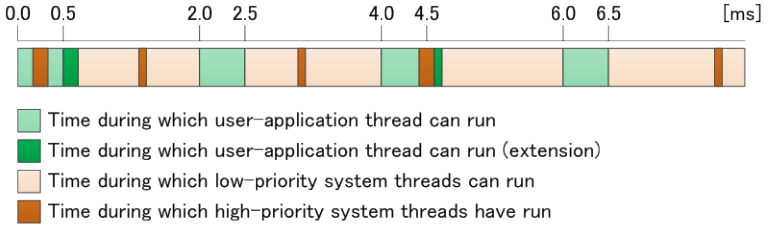
If no high-priority system threads are running, then if A is equal to 25%, the CPU assignments for the user application and low-priority system threads per 2-ms cycle will be as shown in Figure 5-1 System Core Assignments (A=25%).

Figure 5-1 System Core Assignments (A=25%)



If a high-priority system thread runs under the same conditions, then the assignments will be as in Figure 5-2 Running a High-Priority System Thread.

Figure 5-2 Running a High-Priority System Thread



6 Synchronization Mechanisms

The CTR-SDK provides two basic types of synchronization mechanisms: “standard” objects and “lightweight” objects.

6.1 Standard and Lightweight Objects

The standard and lightweight objects provide the same types of synchronization. The differences between the standard and lightweight objects are as follows.

Table 6-1 Differences Between Standard and Lightweight Objects

	Standard	Lightweight
Numeric restrictions	Yes	No
Common base class	`WaitObject`	None
Amount of work when there is no contention	Large	Small
Instance size	Small	Large
Simultaneous waiting for multiple instances	Yes	No

6.1.1 Numeric Restrictions

The system limits the number of standard synchronization objects of each type that can be used at one time. The object’s `Initialize` member function will fail if this limit is exceeded.

To read about the upper limits of the various synchronization objects, see 8.1 Classes With Upper LimitsCommon Base Class and Simultaneous Waits

All standard synchronization objects are derived from `nn::os::WaitObject`.

The `WaitObject` class defines an array of pointers to `WaitObject` objects, a `WaitAny` static member function, and a `WaitAll` static member function. You can use these functions to wait for one of multiple instances of a class derived from `WaitObject`, or for all instances.

6.1.2 Choosing Which to Use

In general, it is better to use the lightweight synchronization objects if the application does not need to wait for multiple instances simultaneously. However, the synchronization objects have differences apart from those listed in the table, so in some cases you will need to take those differences into account.

For example, although the standard `Mutex` object supports inheritance of priority, the lightweight `CriticalSection` object does not. See the following section for details.

6.2 `Mutex` and `CriticalSection` objects

`Mutex` and `CriticalSection` objects are mutual exclusion mechanisms for synchronization. `Mutex` is a standard object, and `CriticalSection` is a lightweight object.

6.2.1 Mutual Exclusion

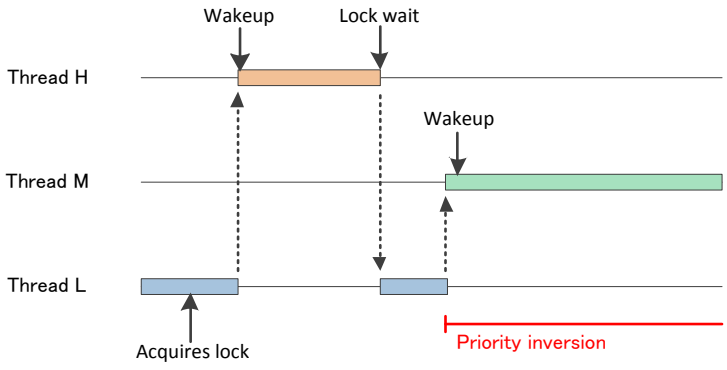
Mutual exclusion “locks” a specified section of code so that it can only be executed by one thread at a time. You call the object’s member functions before entering a specified section of code and after leaving it to ensure exclusive access to that section by one thread at a time. If a thread attempts to enter the specified section while another thread is already executing it, then the thread will be made to wait until the thread currently executing it leaves the specified section.

6.2.2 Priority Inheritance

An issue called “priority inversion” can sometimes occur when using exclusive control (locking).

Priority inversion happens when three threads of differing priorities are involved. For convenience, we will call these three threads H (“high”), M (“medium”), and L (“low”), according to their priorities. Consider that thread L is running and acquires a lock. Now, consider that H starts running and attempts to acquire the same lock. Now H has no choice but to wait until L frees the lock. However, if M becomes runnable, L will be suspended because it has a lower priority, and M will start running. In this case, L is blocked from running until M goes into a waiting state for some reason, and meanwhile, H is not runnable until M goes into a waiting state (because H is waiting for L, which is waiting for M). Priority inversion is the situation in which M, which is not directly contenting for the lock, blocks the execution of H.

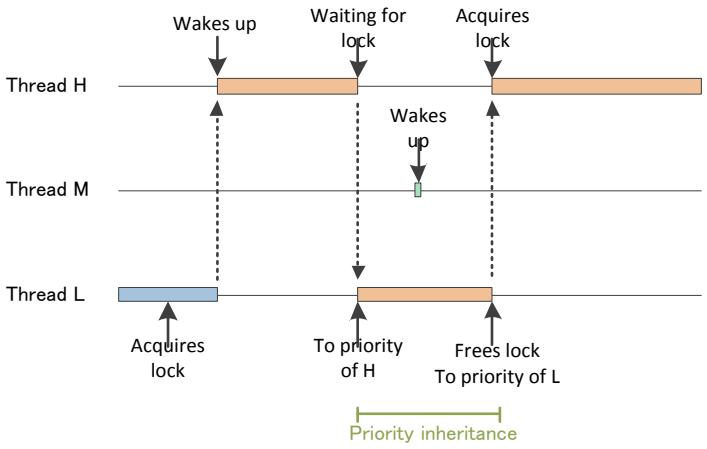
Figure 6-1 Priority Inversion



Although there are a number of ways to remedy priority inversion, a `Mutex` object uses the method called “priority inheritance.” Priority inheritance automatically elevates the priority of a thread that has acquired a lock to the priority of the highest-priority thread waiting for that lock. In the earlier example,

while thread H is waiting to acquire the lock, the priority of L will be elevated to that of H. This will enable L to continue running during this interval without interruption, even if M becomes runnable.

Figure 6-2 Priority Inheritance



The `CriticalSection` object does not have priority inheritance. There is thus a potential for priority inversion.

6.2.3 Automatic Deallocation

When a thread acquires a lock using a `Mutex` or `CriticalSection` object, the system records the thread that acquired the lock, and only allows that thread to release it. Thus if a thread that acquired a lock using a `CriticalSection` is destroyed, then that lock can never be released.

In contrast, if a thread that acquired a lock using a `Mutex` is destroyed, the lock is released automatically.

6.2.4 Comparison with Semaphore

The `Mutex` and `CriticalSection` objects resemble the `Semaphore` and `LightSemaphore` objects in that a maximum of one instance of each is permitted, but they differ in the following ways due to their specialization for exclusive control.

- Nested locks allowed

  A thread that acquires a lock using `Mutex` or `CriticalSection` can acquire a lock again. The system records the number of times the lock has been acquired, and releases the lock when it has been released that many times.

  If you use a semaphore for exclusive control, a thread cannot acquire a lock again until it has released the first lock.

- Only the thread that acquired the lock can release it

  Only the thread that acquired a lock using `Mutex` or `CriticalSection` can release it. If a thread other than the one that acquired the lock tries to release it, it will result in an error.

  If you use a semaphore for exclusive control, threads other than the one that acquired a lock can release it.

6.3 `Semaphore` and `LightSemaphore`

`Semaphore` and `LightSemaphore` are synchronization mechanisms for counter-based control. `Semaphore` is a standard mechanism, and `LightSemaphore` is a lightweight mechanism.

This is used when multiple threads share N resources, to manage the remaining number of available resources.

To use these as synchronization mechanisms, first specify default values for the remaining count and synchronize by decrementing the counter by one, or incrementing it by one or more. Each thread decrements the counter by one before using a resource. The counter is incremented when the thread stops using the resource, or when a new resource is generated. Synchronization works as follows: if the remaining count reaches zero, then the thread will wait until it reaches one or greater. Thus a thread will not continue processing while the remaining count is zero.

6.4 `Event` and `LightEvent`

`Event` and `LightEvent` are synchronization objects that work by way of event notifications. `Event` is a standard object, and `LightEvent` is a lightweight object.

These objects are used to make a thread wait until another thread meets certain conditions. Although this is similar to providing a flag variable and then polling it, other threads cannot run while a thread is waiting for the polled condition; while with `Event` and `LightEvent`, the thread pauses, and other threads are able to run.

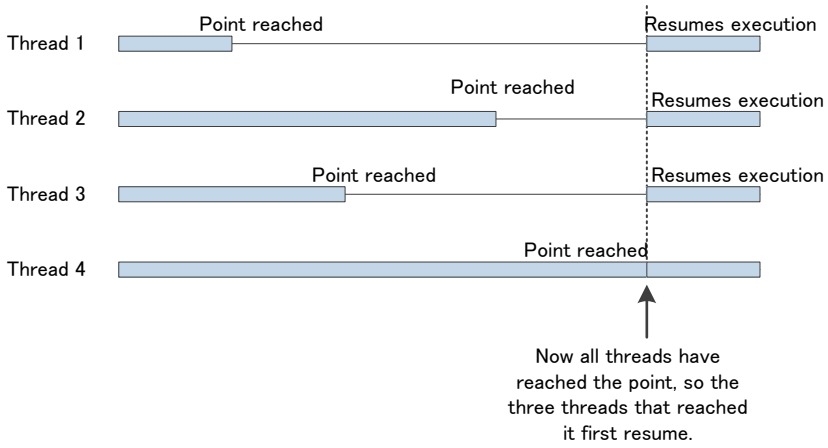
You can think of it as a Boolean flag that causes threads to wait until it is set without using CPU time.

6.5 LightBarrier

`LightBarrier` is a synchronization object that works by coordinating the flow of execution. `LightBarrier` is a lightweight object; no standard object is provided.

This is used to make N threads wait until all have reached a specified point in the code. The first threads that reach the specified point in the code wait until all N threads have reached that point. Once all N threads have reached that point, all N threads are resumed automatically.

Figure 6-3 Operation of LightBarrier (N=4)



7 Time

Two types of time are available with the CTR-SDK: ticks, and normal date and time.

7.1 Ticks

The `tick` value is a 64-bit signed value indicating the total count of CPU cycles since the system started.

7.1.1 The tick Count

The `tick` count starts at 0 when the system starts, and is incremented by one each CPU cycle. Since the CTR system's CPU runs at about 268 MHz, the value increases by approximately 268 million per second. This value would reach the maximum value of a 32-bit unsigned integer in about 16.0 seconds, and the maximum value of a 64-bit signed integer in about 1,090 years.

The current `tick` value will always be even, because the actual implementation on the CTR is to increment this value by two every two cycles.

7.1.2 Precision of the tick Count

`tick` is merely a count of clock cycles. It is only relevant to the internal timing of the CTR computer system. Accuracy is therefore not guaranteed when comparing this value with the actual time. In fact, the `tick` count is very inaccurate as a measure of time; it is several times to several dozens of times less accurate than a normal clock. There is also substantial variation between CTR systems. After a day, the difference in `tick` counts between two systems may be measurable in seconds.

7.1.3 Ticks and Sleep

The CTR system goes into Sleep Mode when closed, and applications must support this feature. During Sleep Mode, the `tick` count is not incremented. Even in Sleep Mode, however, the `tick` count is incremented while StreetPass, SpotPass, or other services are running. There is thus no guarantee as to the amount that the `tick` count will increase over a given period if the system goes into Sleep Mode during that period. If you want to manage the `tick` count over a long period, you must consider the effect of Sleep Mode on the `tick` value.

7.1.4 Tick

The `nn::os::Tick` class represents the `tick` value. Its only member variable is a 64-bit signed integer. It provides arithmetic operators, and can be converted to and from the `nn::fnd::TimeSpan` class (described below).

Instances of this class are locked to a specific value in their constructors. If you want to perform operations requiring the value to be changed, create a new instance.

This class has a static member function for obtaining an instance representing the current `tick` value.

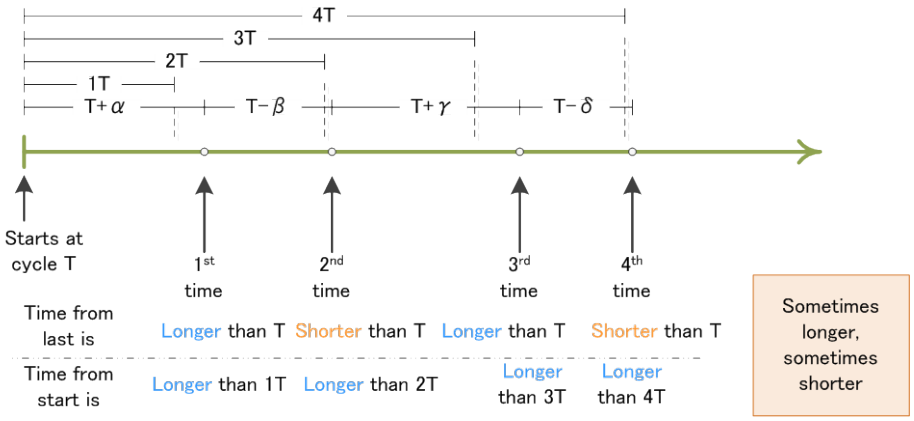
7.1.5 Timer

The `nn::os::Timer` class is used for determining whether a specified interval of time has elapsed. You first specify a time, and then determine whether that interval of time has elapsed. You can have a thread wait until the specified time interval has elapsed.

The time elapse may be detected after it has actually elapsed for a number of reasons, but the application will never determine that the time has elapsed before it actually has. If there is a gap between elapse and detection, the detection will always be later.

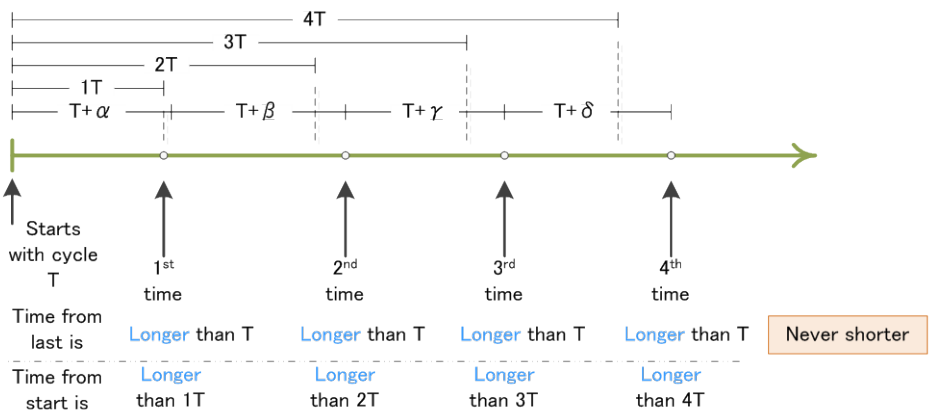
You can run `Timer` one time only, or as a cycle. If you run a `Timer` as a cycle, it will run at a predetermined interval. In other words, it will determine that the time has elapsed when the difference between the current time and the time when the class was initialized is an integer multiple of the initially specified interval. The difference between the current time and the initial time is given precedence over the time since the last interval. As a result, a time elapse may sometimes be detected before a full interval has elapsed since the last one. Even if this happens, the elapse of time will never be shorter than a multiple of the specified interval since the starting time.

Figure 7-1 Cyclic Operation of Timer (Predetermined Interval)



If you need the time elapsed from the last time to be the same each time (i.e., you need a constant delay), then you must use run-once behavior, and configure a new timer each time.

Figure 7-2 Constant Delay



The accuracy of the `Timer` class is no better than that of the tick count, because its operation is based on this value. You must take care with the handling of the `Timer` class before and after entering Sleep Mode, because it is similarly affected by Sleep Mode.

7.2 Dates and Times

7.2.1 DateTime

The `nn::fnd::DateTime` class represents a specific date and time. Its only member variable is a 64-bit signed integer, which represents the number of milliseconds elapsed since 00:00:00AM on January 1st, 2000. It can be used in arithmetic operations with the `nn::fnd::TimeSpan` class (described below).

Instances of this class are locked to a specific date and time in their constructors. The class does not have member functions for changing the date and time stored by a given instance. If you need to perform operations requiring the date/time to be changed, you must create a new instance.

This class has a static member function for obtaining an instance representing the current date and time from the RTC.

7.2.2 TimeSpan

The `nn::fnd::TimeSpan` class represents a specific time interval. Its only member variable is a 64-bit signed integer, which represents a time interval in nanoseconds. It can be used in arithmetic operations with the `nn::fnd::DateTime` class, and converted to and from the `nn::os::Tick` class.

As with the `DateTime` class, the value of an instance of this class is fixed in the constructor. If you need to perform operations requiring the interval to be changed you must create a new instance. It is

also not possible to generate an instance with a `0` interval using the public constructor. Use one of the static member functions starting with "`From`" to generate an instance with an arbitrary interval value.

7.2.3 The RTC

The real-time clock (RTC) is a hardware component on the CTR system that manages the current date and time. It maintains the correct time based on the time set by the user, even when the CTR system is powered off.

Use the `nn::fnd::DateTime::GetNow` function to get the current time from the RTC. This is a static member function of the `DateTime` class. It returns an instance of the `DateTime` class representing the current time.

The RTC has the same level of accuracy as an ordinary clock.

7.2.4 RTC Alarm

RTC Alarm is a feature provided by the RTC for giving time notifications. The RTC provides an `nn::os::Event` notification when the previously set date and time are reached.

Control the RTC Alarm feature using the following API.

- `nn::ptm::RegisterAlarmEvent(nn::os::Event&)`
- `nn::ptm::SetRtcAlarm(nn::fnd::DateTime)`
- `nn::ptm::GetRtcArlarm(nn::fnd::DateTime*)`
- `nn::ptm::CancelRtcAlarm()`

You can only set one RTC Alarm at a time. Because of this, if you need to set two or more alarms then you must set them sequentially, starting with the earliest date/time, and then setting the next time after that alarm notification is received. Although `DateTime` is used to set the alarm, RTC Alarm is only precise to the minute level, so the alarm may be up to one minute late.

Unlike `nn::os::Timer`, RTC Alarm is not affected by Sleep Mode. If the set time is reached during Sleep Mode, the application will receive the notification upon waking up.

8 Limits on Resources

For several classes of the os library there are upper limits on the number of instances that can exist simultaneously. It is important to design the application so these numeric limits are not exceeded.

8.1 Classes With Upper Limits

Table 8-1 lists those classes that have upper limits on instances, together with their limit values.

Table 8-1 Limit Values for Classes With Numeric Limits

Class	Upper Limit Value
nn::os::Event	32
nn::os::Mutex	32
nn::os::Semaphore	8
nn::os::Thread	32
nn::os::Timer	8

8.2 Getting Upper Limit Values and Usage

Classes with a limit on the number of instances have the class member functions s32 GetCurrentCount and s32 GetMaxCount. You can use the GetCurrentCount function to get the number of instances of that class that are currently being used, and you can use the GetMaxCount function to get the maximum number of instances that can be used simultaneously for that class.

8.3 Usage by the SDK

Instances of a class used internally by the various libraries and classes of the CTR-SDK are counted in the total number of instances used by the application. To learn which resources are used and to what extent by the various SDK libraries and classes, see the documents for those libraries and classes.

This document focuses on information pertinent to virtually all applications: the amount of resources used from when the application starts and the amount of resources consumed by the nngxInitialize function. The resources used from the start of the application are secured in Step 8 (initialize the user application environment) of the procedure described in section 3.1 Starting Applications, so when the nninitStartUp function and the C/C++ static initializer run these resources have not yet been secured.

Table 8-2 Amounts of Resources used by SDK

Class	At Application Startup	nngxInitialize
nn::os::Event	1	1
nn::os::Mutex	0	0
nn::os::Semaphore	0	0
nn::os::Thread	2	1
nn::os::Timer	0	0

9 Debugging Output

9.1 String Output Macros

The CTR-SDK does not provide access to standard input or standard output. It is thus not possible to output strings to standard output using `std::printf` or other functions. For this reason, the CTR-SDK provides dedicated functions for the output of debugging strings.

Meanwhile, the output of debugging strings only has meaning when running on the debugger; such processing is pointless in environments other than the debugger, such as development systems or retail systems. The processing load is thus different in the two types of systems, because string output is only actually processed by the debugger. Differences in processing load between the debugger and non-debugger environments can make it infeasible to debug timing bugs. Additionally, most of the strings used for debugging output tend to be literal strings, which increases the memory footprint by an unexpectedly large amount.

For these reasons, the CTR-SDK uses macros for debugging output rather than functions, in order to make it easy to exclude them from compilation.

The CTR-SDK provides three macros for string output: `NN_LOG`, `NN_LOGV`, and `NN_PUT`. By default, these macros output strings to the debugger as defined, but if you define the `NN_SWITCH_DISABLE_DEBUG_PRINT` macro, they will be replaced by empty macros, and will not perform any processing. Macro replacement also removes any string literals specified as arguments, which reduces the memory footprint.

If you are using the SDK build system, you can set the `DEBUG_PRINT` variable to `true` or `false` in your `OMakefile` to control whether the `NN_SWITCH_DISABLE_DEBUG_PRINT` macro is defined.

9.2 Formatted String Output

The `NN_LOG` and `NN_LOGV` macros output strings in the same way as `std::printf` and `std::vprintf`, respectively. However, the output buffer is allocated on the stack, and consequently, the macros restrict the maximum output string length to 256 bytes. If you attempt to output a string longer than this, it will be truncated to the maximum length. If you want to output a string longer than 256 bytes, you must make multiple calls to the output macro, or use `NN_PUT`.

9.3 Simple String Output

`NN_PUT` is a low-level string-output macro. It outputs the specified number of bytes from the specified pointer without modification to the debugger. It does not have a formatting feature like `NN_LOG` or `NN_LOGV`, so if you need to perform formatting, you must first do so using `std::snprint` or the equivalent.

10 Abnormal Termination of the Application

The application can be terminated if it is detected that the application or a library is abnormal. However, these features are primarily for development, and termination should not be used in the commercial product.

10.1 CPU Exceptions

`NN_PANIC` and its derived APIs are macro APIs to terminate application processing. When the application detects an inconsistent state from which recovery is impossible, it can terminate itself.

If the application is run from the debugger, when terminating with `NN_PANIC`, the reason for termination and the position in the source code can be sent to the output of the debugger. Also, in the debugger, the application is in a paused state for debugging, so debugging can be performed while in a terminated state.

PANIC group APIs are in a macro format for the same reason as string output macros.

10.2 ASSERT

An ASSERT is code that is inserted in the targeted source code for testing whether conditions at a location in the source code are met exactly as intended. The CTR-SDK provides APIs, such as `NN_ASSERT`, to perform an ASSERT in macro format. With this API, if an ASSERT fails, a message is sent to the debugger output, and the application is terminated.

Because an ASSERT is a test performed while an application is running, the application processing load is increased. Although testing with ASSERT is useful during development, because it is unnecessary for commercial products that have completed debugging, ASSERT is removed from commercial products. For this reason, the ASSERT group macros are replaced with empty macros when the `NN_SWITCH_DISABLE_ASSERT_WARNING` is defined, and no processing will be performed.

If the SDK build system is used, you can control whether `NN_SWITCH_DISABLE_ASSERT_WARNING` is defined by specifying `true` or `false` in the `ASSERT_WARNING` variable used by the `OMakefile`.

Actual termination when ASSERT fails uses PANIC.

10.3 Behavior during Abnormal Termination

Behavior after an ASSERT fails or a PANIC occurs differs according to whether the application is running on the debugger and depends on the “Other Setting” -> “Break Stop” settings in Config.

Table 10-1 Resources Used by the SDK

Executed from Debugger	Value Set for Break Stop	Result
No	**enable**	The entire system is terminated. However, the GPU/DSP and other hardware continue to run.
	disable	The application is forced to close. The screen is preserved for a fixed time, and then returns to the HOME Menu (or the development menu).
Yes	**(unrelated)**	The application is terminated in a state that allows debugging from the debugger.

The value of “Break Stop” is fixed to “disable” in the commercial product environment. For this reason, the application is forced to close when the application terminates abnormally with a commercial product.

10.4 Abnormal Termination Handler

A handler can be registered for the process to be performed by the application before the application terminates when an ASSERT fails or a PANIC occurs.

By specifying a pointer to a `nn::dbg::BreakHandler` type function with the `nn::dbg::SetBreakHandler` function, that function is called when an ASSERT fails or a PANIC occurs. Note that to prevent an infinite loop if an ASSERT fails or a PANIC occurs in the handler function, the handler registration is deleted before the handler is called.

Because the handler function may be called from multiple threads at different times and is originally called in a state when an inconsistency is occurring in the internal state of the application, the processes that can be run are considerably limited. In particular, SDK APIs that require exclusivity cannot be called, and the call is made presuming failure or a deadlock.

The abnormal termination handler structure is intended to enable the application to send the abnormal termination information to a screen for debugging on the development device.

11 CPU Exception Handlers

11.1 CPU Exceptions

11.1.1 CPU Exceptions and the CPU Exception Handler

A CPU exception is an illegal operation on the CPU, such as accessing a NULL pointer. It is completely different from the C++ exception object implemented by the `try`, `throw`, and `catch` statements.

You can configure the CTR-SDK to call a predefined function when a subset of CPU exceptions occur. This feature can only be used for debugging and is only enabled when you set **Exception Handler** to “enable” in the Config tool. Although this feature is enabled by default in the development environment, it is always disabled in retail products and cannot be enabled.

The function that is called when a CPU exception occurs is called a “CPU exception handler,” or simply an “exception handler.” By setting an appropriate exception handler, you can display information on the screen when a CPU exception occurs in a non-debugging environment (such as on a development system), such as what exception occurred and the CPU status when it occurred.

11.1.2 Types of Exceptions

You can handle the following four exceptions in an exception handler.

- Pre-fetch abort exception

  This exception occurs when the application jumps to a non-executable address. In most cases, this is caused by stack corruption due to a buffer overflow or other error, or by a virtual function call from a freed pointer.

- Data abort exception

  This exception occurs when the application tries to write to a read-only area, or read or write to a memory address that had never been assigned. It can be caused by a wide range of errors, including buffer overflows, using deallocated memory, or accessing an uninitialized variable.

- Undefined instruction exception

  This exception occurs when the application tries to execute an address that is executable, but which contains code that is invalid as an instruction. This exception also occurs when the application tries to execute a breakpoint instruction. In the first case, the cause is the same as that of a pre-fetch abort.

- VFP exception

  This exception occurs when the application tries to execute a floating-point operation that cannot be executed on the VFP coprocessor. The operations that cause this exception depend on the VFP

coprocessor settings. This exception does not occur when using the default settings of the CTR-SDK.

In the debugging environment, however, the exception handler will never be called, because the debugger will handle CPU exceptions other than the VFP exception before they can get to the exception handler.

11.1.3 Behavior When an Exception Occurs

Consider, for example, what will happen in the code below when `pVec` is set to `NULL`.

Code 11-1 Exception-Generating Code

```
void Set1(nn::math::VEC3* pVec)
{
    pVec->x = 1.0f;
    pVec->y = 1.0f;
    pVec->x = 1.0f;
}
```

Here, a data-abort exception will occur on the first line of the function, when assigning a value to `pVec->x`. If the application has set an exception handler, then it will be called at this point. The behavior in this case will be nearly identical to that of the code below:

Code 11-2 Pseudocode of Exception Occurrence

```
void Set1(nn::math::VEC3* pVec)
{
    goto pExceptionHandler;
    pVec->x = 1.0f;
    pVec->y = 1.0f;
    pVec->x = 1.0f;
}
```

The first key thing to note about this code is that the handler is called before the assignment is executed. To be precise, the application attempts to execute the assignment, but the exception causes the assignment to be canceled and the exception handler to be called. It thus appears to the exception handler that the assignment has not taken place.

Another key point is that the exception handler is invoked by a simple jump, and not by a function call. It is not possible to code a simple jump in the C or C++ language, and so the pseudocode in the sample above expresses the operation using a `goto` statement. To be precise, the system performs a number of operations before calling the exception handler, including collecting exception information and setting up the stack for the exception handler, but at the point when the exception handler is called, the state is nearly the same as if a simple jump like this had been performed.

It is not possible to return from the exception handler by a simple method, because it was invoked by a jump rather than a function call. Also, even if the function were able to return normally, this would result in an infinite loop as the application would again try to assign to `pVec->x`, which would again invoke the exception handler. Normally, there will be a call to `NN_PANIC` at the end of the exception handler, which causes the application as a whole to terminate.

11.2 Setting an Exception Handler

11.2.1 Global and Local Exception Handlers

There are two types of exception handlers: global and local.

It is possible to set a different exception handler for each thread. Local exception handlers are different exception handlers set for each thread.

In contrast, a global exception handler indicates a local exception handler set for the main thread. When there is no local exception handler set on a thread other than the main thread, then the setting for the main thread's exception handler will be used in its place. In other words, the local exception handler setting for the main thread is used as the default setting for the application as a whole.

Table 11-1 shows which of the set exception handlers is invoked when an exception occurs, depending on whether the exception occurred in the main thread, and whether a local exception handler has been set.

Use `nn::os::ARM::SetUserExceptionHandler` to set a global exception handler, and `nn::os::ARM::SetUserExceptionHandlerLocal` to set a local exception handler.

Table 11-1 Determining the Exception Handler Setting That Is Used

Exception Occurred in Main Thread?	Local Exception Handler Set?	
	No	Yes
Yes	Global exception handler	Global exception handler
No	Global exception handler	Local exception handler

11.2.2 Exception Handler Setting Patterns

To set an exception handler, specify (1) a pointer to the handler function to call when an exception occurs, (2) the stack for the handler function to use, and (3) a buffer in which to store the exception information. These can pass a pointer or an address simply for a memory region, or special constants defined by the SDK.

Table 11-2 describes how the buffer for the stack and exception information is actually allocated for each setting.

Table 11-2 Setting the Exception Handler

Value Specified		Values To Use	
stackBottom	**pExceptionBuffer**	**Stack**	**Exception Information Buffer**
Address of memory region	Pointer to memory region	Use `stackBottom`.	Use `pExceptionBuffer`.
	`nn::os::EXCEPTION_BUFFER_USE_HANDLER_STACK`	Use stackBottom.	Use stackBottom.
	`nn::os::EXCEPTION_BUFFER_USE_THREAD_STACK`	Use stackBottom.	Use the stack of the thread where the exception occurred.
`nn::os::HANDLER_STACK_USE_THREAD_STACK`	Pointer to memory region	Use the stack of the thread where the exception occurred.	Use pExceptionBuffer.
	`nn::os::EXCEPTION_BUFFER_USE_HANDLER_STACK`	Use the stack of the thread where the exception occurred.	Use the stack of the thread where the exception occurred.
	`nn::os::EXCEPTION_BUFFER_USE_THREAD_STACK`	Use the stack of the thread where the exception occurred.	Use the stack of the thread where the exception occurred.

Where the same region is used for the stack and the exception information buffer in the table above, the buffer for the exception information is first allocated from the stack, and then the remainder is used as the actual stack.

Revision History

Version	Revision Date	Category	Description
1.4	2012/06/22	Changed	・ 3.2.1 Default `nninitStartUp` Updated the process that the default `nninitStartUp` is performing.
		Deleted	・ 9.3.4 Terminating the Application
		Added	・ 10 Abnormal Termination of the Application
1.3	2012/04/23	Changed	・ 3.2.1 Default `nninitStartUp` Updated the process that the default `nninitStartUp` is performing. ・ 4.3.1 Required Functions and Classes (Layer 1) Added explanation that heap operations cannot be performed when using the default memory management system.
1.2	2011/12/12	Added	・ 3.6 Memory Allocation by STL ・ 8 Limits on Resources
1.1	2011/10/27	Changed	・ 5.6.2 Assigning CPU Time
1.0	2011/09/01	Added	・ 5 Threading ・ 6 Synchronization Mechanisms ・ 7 Time ・ 8 Debugging Output ・ 9 CPU Exception Handlers
0.3	2011/06/22	Deleted	Chapter 5 Threads
0.2	2010/09/27	Changed	Section 3.1 Starting Applications
0.1	2010/08/19	—	Initial version.