System Programming Guide

OS and Debug Libraries (os/dbg)

Version 0.3

2011/06/22

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

CTR-06-0034-001-C
Released: August 5, 2011

Table of Contents

Code

Figures

1 Introduction

This document explains the memory management required to create applications for CTR.

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.

- 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.

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 (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 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 the malloc, free, new, and delete subroutines to use the aforementioned fnd heap.

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.

3.3 (6) C++ Static Initializer

The C++ static initializer is part of 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 (5) nninitStartUp for details.

3.4 (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.

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

3.5 (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 Code 3-2) or include nn.h (as shown in Code 3-3). Including nn.h will cause the nnMain function to be declared with C linkage.

Code 3-2 Explicitly Declaring the C Linkage

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

Code 3-3 Including the nn.h Header

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

When control returns from `nnMain`, the user application will exit, and the system will return to the HOME Menu.

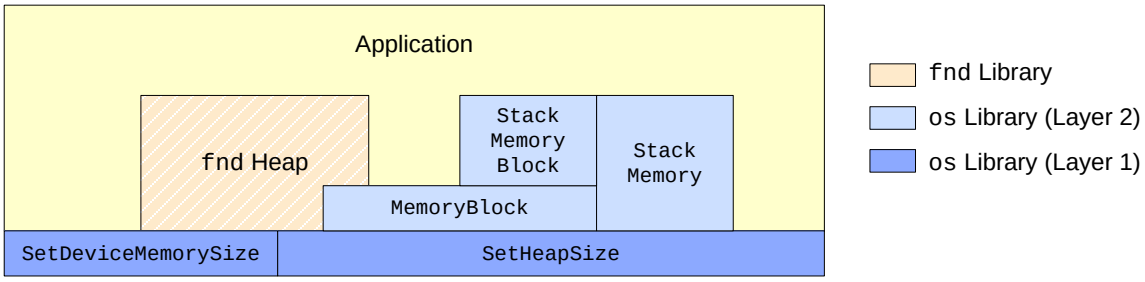
4 Memory Management

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.

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 Thead::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 (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.

Revision History

Version	Revision Date	Category	Description
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.