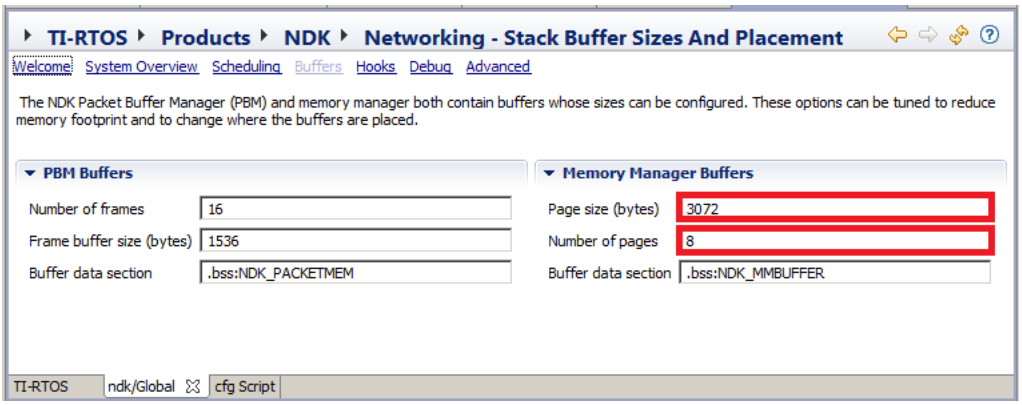
NDK Static Internal Memory Manager

Contents

The NDK has a component called a Memory Manager. The buffers are used as a scratchpad memory resource for many components within the NDK. For example, the `bind()` function needs a little memory and the DHCP client needs to store the lease option, this memory is used. This memory is statically defined and placed in the `.xxx:NDK_MMBUFFER` subsection (xxx depends on the target). The size of the section is dictated by the following configuration parameters:

- **Global.memRawPageCount**: Number of internal memory buckets.
- **Global.memRawPageSize**: Size of each bucket.

To set these parameters, first open the NDK settings in your application's configuration file (*.cfg) as described in Configuring NDK Memory Use. Once you see the **TI-RTOS > Products > NDK > Networking - Welcome** configuration panel, choose the **Buffers** link.



About Memory Manager buffers, buckets, and blocks

To reduce external fragmentation and improve performance, there are N buckets in the memory manager. N is determined by the `Global.memRawPageCount` field. Each bucket manages a buffer of size M (where M is determined by `Global.memRawPageSize`).

The following arrays hold the bucket information and the large buffer that will be divided between the buckets. <syntaxhighlight lang='c'> PITENTRY ti_ndk_config_Global_pit[RAW_PAGE_COUNT]; // RAW_PAGE_COUNT = Global.memRawPageCount UINT8 ti_ndk_config_Global_pitBuffer[RAW_PAGE_SIZE*RAW_PAGE_COUNT]; // RAW_PAGE_SIZE = Global.memRawPageSize </syntaxhighlight>

During startup, the `ti_ndk_config_Global_pitBuffer` buffer is sliced up into buckets. Each bucket in `ti_ndk_config_Global_pit` is given a buffer of size `Global.memRawPageSize`. Each bucket will slice up its buffer into same-sized blocks at run-time. To reduce internal fragmentation, the valid block sizes are (in MAUs): 48, 96, 128, 256, 512, 1536, and `Global.memRawPageSize`.

`Global.memRawPageSize` should be a multiple of these sizes to minimize wasted memory, since the buffer of size `Global.memRawPageSize` is sliced up into block-size pieces. For example, if `Global.memRawPageSize` is 3072, in a 96-MAU bucket there will be 3072/96 = 32 blocks.

When the internal components of the NDK need memory, they call `mmAlloc()`. Within `mmAlloc()`, each bucket is examined to determine which one matches the requested size the best. This approach reduces the amount of internal fragmentation.

The fixed block-size for a bucket is not determined during start-up. All N buckets are "un-initialized" at start-up and do not have a block size associated with them. When `mmAlloc()` is called at run-time, the list of valid block-sizes (48, 96, etc.) is examined to determine the best size to use. The buckets are examined to see if one that is already managing blocks of that size has free blocks available. If such a bucket exists, that bucket is used for this allocation. Otherwise, an uninitialized bucket is used and becomes associated with that block size.

If you look at the `ti_ndk_config_Global_pit` array, you might see that it has multiple buckets allocating blocks of the same size. This impacts performance somewhat, but allows a wide variety of memory allocation scenarios to be used with the NDK.

When `mmFree()` is called and a bucket has no more allocated blocks, the bucket is uninitialized so that it no longer has an associated block size.

Example

Suppose there are 6 buckets (`Global.memRawPageCount = 6`), and the size of each bucket is 3072 (`Global.memRawPageSize = 3072`). At start-up all buckets are "uninitialized".

The 6 buckets would have the allocations shown below if `mmAlloc()` is called by the NDK with the following sizes (in this order): 62, 1536, 1536, 48, and 1536.

```
bucket 0: size = 96;   number of total blocks: 32 (3072/96);   number of free blocks: 31
bucket 1: size = 1536; number of total blocks:  2 (3072/1536); number of free blocks:  0
bucket 2: size = 48;   number of total blocks: 64 (3072/48);   number of free blocks: 63
bucket 3: size = 1536; number of total blocks:  2 (3072/1536); number of free blocks:  1
```

```
bucket 4: un-initialized
bucket 5: un-initialized
```

If the 62-MAU block and all the 1536-MAU blocks are then freed (and no more allocation occurs), the buckets would have the following allocations:

```
bucket 0: un-initialized
bucket 1: un-initialized
bucket 2: size = 48;   number of total blocks: 64 (3072/48);   number of free blocks: 63
bucket 3: un-initialized
bucket 4: un-initialized
bucket 5: un-initialized
```

How to reduce RAM usage

If is difficult to give concrete guidelines for reducing RAM usages since the Memory Manager is very dynamic and application-dependent. You can call the `_mmCheck()` API to display current usage, but it is just a single snapshot.

The Memory Manager never uses a bucket with a larger size than is needed to fulfill a request. For example, suppose you have 6 buckets and their sizes are currently 48, 48, 512, 512, 256, and 1536. If a request for 94 MAUs is made, an allocation error will occur. A 256 MAU block will not be allocated for the 94 MAU request.

However, suppose you had 6 buckets and the current sizes were 48, 48, 96, 512, 1536, and uninitialized. The allocation for 94 MAUs would succeed because either the third bucket would be used, or if it had already allocated all of its blocks, the last bucket would be initialized to manage block sizes of 96.

If you reduce the `Global.memRawPageCount` too much, the likelihood of an allocation error increases. If the `Global.memRawPageSize` is greater than 1536, we recommend that you set `Global.memRawPageCount` to 6 or higher. Note that there are 7 block sizes the NDK can use--48, 96, 128, 256, 512, 1536, and `Global.memRawPageSize`.

You can reduce the `Global.memRawPageSize` setting. The best choice depends on the NDK components being used. For example the HTTP Server calls `mmAlloc()` with a size of 1,676 bytes.

Unfortunately, determining the optimal page size and count is currently done by trial and error. We are looking into the best way to add enhancements related to such optimization.

{{

1. switchcategory:MultiCore=
   - For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum
   - For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum

   Please post only comments related to the article **NDK Static Internal Memory Manager** here.

Keystone=

- For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum
- For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum

Please post only comments related to the article **NDK Static Internal Memory Manager** here.

C2000=*For technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments about the article **NDK Static Internal Memory Manager** here.*

DaVinci=*For technical support on DaVincoplease post your questions on The DaVinci Forum. Please post only comments about the article **NDK Static Internal Memory Manager** here.*

MSP430=*For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the article **NDK Static Internal Memory Manager** here.*

OMAP35x=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **NDK Static Internal Memory Manager** here.*

OMAPL1=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **NDK Static Internal Memory Manager** here.*

MAVRK=*For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum. Please post only comments about the article **NDK Static Internal Memory Manager** here.*

*For technical su please post you questions at http://e2e.ti.cor Please post on comments abo article **NDK Sta Internal Memo Manager** here.*

}}

Links

Amplifiers & Linear
Audio
Broadband RF/IF & Digital Radio
Clocks & Timers
Data Converters
DLP & MEMS
High-Reliability
Interface
Logic
Power Management
Processors

- ARM Processors
- Digital Signal Processors (DSP)
- Microcontrollers (MCU)
- OMAP Applications Processors

Switches & Multiplexers
Temperature Sensors & Control ICs
Wireless Connectivity