LIBNVDIMM: Non-Volatile Devices

libnvdimm - kernel / libndctl - userspace helper library

nvdimm@lists.linux.dev

Version 13

Glossary

PMEM:
: A system-physical-address range where writes are persistent. A block device composed of PMEM is capable of DAX. A PMEM address range may span an interleave of several DIMMs.

DPA:
: DIMM Physical Address, is a DIMM-relative offset. With one DIMM in the system there would be a 1:1 system-physical-address:DPA association. Once more DIMMs are added a memory controller interleave must be decoded to determine the DPA associated with a given system-physical-address.

DAX:
: File system extensions to bypass the page cache and block layer to mmap persistent memory, from a PMEM block device, directly into a process address space.

DSM:
: Device Specific Method: ACPI method to control specific device - in this case the firmware.

DCR:
: NVDIMM Control Region Structure defined in ACPI 6 Section 5.2.25.5. It defines a vendor-id, device-id, and interface format for a given DIMM.

BTT:
: Block Translation Table: Persistent memory is byte addressable. Existing software may have an expectation that the power-fail-atomicity of writes is at least one sector, 512 bytes. The BTT is an indirection table with atomic update semantics to front a PMEM block device driver and present arbitrary atomic sector sizes.

LABEL:
: Metadata stored on a DIMM device that partitions and identifies (persistently names) capacity allocated to different PMEM namespaces. It also indicates whether an address abstraction like a BTT is applied to the namepsace. Note that traditional partition tables, GPT/MBR, are layered on top of a PMEM namespace, or an address abstraction like BTT if present, but partition support is deprecated going forward.

Overview

The LIBNVDIMM subsystem provides support for PMEM described by platform firmware or a device driver. On ACPI based systems the platform firmware conveys persistent memory resource via the ACPI NFIT "NVDIMM Firmware Interface Table" in ACPI 6. While the LIBNVDIMM subsystem implementation is generic and supports pre-NFIT platforms, it was guided by the superset of capabilities need to support this ACPI 6 definition for NVDIMM resources. The original implementation supported the block-window-aperture capability described in the NFIT, but that support has since been abandoned and never shipped in a product.

Supporting Documents

ACPI 6:
: https://www.uefi.org/sites/default/files/resources/ACPI_6.0.pdf

NVDIMM Namespace:
: https://pmem.io/documents/NVDIMM_Namespace_Spec.pdf

DSM Interface Example:
: https://pmem.io/documents/NVDIMM_DSM_Interface_Example.pdf

Driver Writer's Guide:
: https://pmem.io/documents/NVDIMM_Driver_Writers_Guide.pdf

Git Trees

LIBNVDIMM:
: https://git.kernel.org/cgit/linux/kernel/git/nvdimm/nvdimm.git

LIBNDCTL:
: https://github.com/pmem/ndctl.git

LIBNVDIMM PMEM

Prior to the arrival of the NFIT, non-volatile memory was described to a system in various ad-hoc ways. Usually only the bare

minimum was provided, namely, a single system-physical-address range where writes are expected to be durable after a system power loss. Now, the NFIT specification standardizes not only the description of PMEM, but also platform message-passing entry points for control and configuration.

PMEM (nd_pmem.ko): Drives a system-physical-address range. This range is contiguous in system memory and may be interleaved (hardware memory controller striped) across multiple DIMMs. When interleaved the platform may optionally provide details of which DIMMs are participating in the interleave.

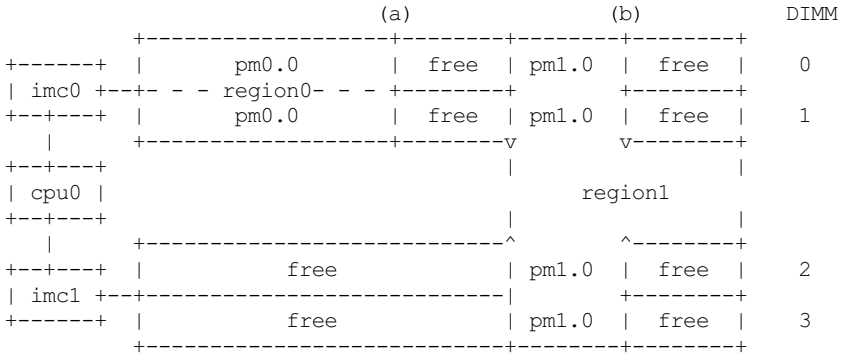
It is worth noting that when the labeling capability is detected (a EFI namespace label index block is found), then no block device is created by default as userspace needs to do at least one allocation of DPA to the PMEM range. In contrast ND_NAMESPACE_IO ranges, once registered, can be immediately attached to nd_pmem. This latter mode is called label-less or "legacy".

PMEM-REGIONs, Atomic Sectors, and DAX

For the cases where an application or filesystem still needs atomic sector update guarantees it can register a BTT on a PMEM device or partition. See LIBNVDIMM/NDCTL: Block Translation Table "btt"

Example NVDIMM Platform

For the remainder of this document the following diagram will be referenced for any example sysfs layouts:



In this platform we have four DIMMs and two memory controllers in one socket. Each PMEM interleave set is identified by a region device with a dynamically assigned id.

1. The first portion of DIMM0 and DIMM1 are interleaved as REGION0. A single PMEM namespace is created in the REGION0-SPA-range that spans most of DIMM0 and DIMM1 with a user-specified name of "pm0.0". Some of that interleaved system-physical-address range is left free for another PMEM namespace to be defined.
2. In the last portion of DIMM0 and DIMM1 we have an interleaved system-physical-address range, REGION1, that spans those two DIMMs as well as DIMM2 and DIMM3. Some of REGION1 is allocated to a PMEM namespace named "pm1.0".

This bus is provided by the kernel under the device /sys/devices/platform/nfit_test.0 when the nfit_test.ko module from tools/testing/nvdimm is loaded. This module is a unit test for LIBNVDIMM and the acpi_nfit.ko driver.

LIBNVDIMM Kernel Device Model and LIBNDCTL Userspace API

What follows is a description of the LIBNVDIMM sysfs layout and a corresponding object hierarchy diagram as viewed through the LIBNDCTL API. The example sysfs paths and diagrams are relative to the Example NVDIMM Platform which is also the LIBNVDIMM bus used in the LIBNDCTL unit test.

LIBNDCTL: Context

Every API call in the LIBNDCTL library requires a context that holds the logging parameters and other library instance state. The library is based on the libabc template:

https://git.kernel.org/cgit/linux/kernel/git/kay/libabc.git

LIBNDCTL: instantiate a new library context example

```
struct ndctl_ctx *ctx;

if (ndctl_new(&ctx) == 0)
        return ctx;
else
        return NULL;
```

LIBNVDIMM/LIBNDCTL: Bus

A bus has a 1:1 relationship with an NFIT. The current expectation for ACPI based systems is that there is only ever one platform-global NFIT. That said, it is trivial to register multiple NFITs, the specification does not preclude it. The infrastructure supports multiple busses and we use this capability to test multiple NFIT configurations in the unit test.

LIBNVDIMM: control class device in /sys/class

This character device accepts DSM messages to be passed to DIMM identified by its NFIT handle:

```
/sys/class/nd/ndctl0
|-- dev
|-- device -> ../../../ndbus0
|-- subsystem -> ../../../../../../../class/nd
```

LIBNVDIMM: bus

```
struct nvdimm_bus *nvdimm_bus_register(struct device *parent,
       struct nvdimm_bus_descriptor *nfit_desc);
```

```
/sys/devices/platform/nfit_test.0/ndbus0
|-- commands
|-- nd
|-- nfit
|-- nmem0
|-- nmem1
|-- nmem2
|-- nmem3
|-- power
|-- provider
|-- region0
|-- region1
|-- region2
|-- region3
|-- region4
|-- region5
|-- uevent
`-- wait_probe
```

LIBNDCTL: bus enumeration example

Find the bus handle that describes the bus from Example NVDIMM Platform:

```
static struct ndctl_bus *get_bus_by_provider(struct ndctl_ctx *ctx,
                const char *provider)
{
        struct ndctl_bus *bus;

        ndctl_bus_foreach(ctx, bus)
                if (strcmp(provider, ndctl_bus_get_provider(bus)) == 0)
                        return bus;

        return NULL;
}

bus = get_bus_by_provider(ctx, "nfit_test.0");
```

LIBNVDIMM/LIBNDCTL: DIMM (NMEM)

The DIMM device provides a character device for sending commands to hardware, and it is a container for LABELs. If the DIMM is defined by NFIT then an optional 'nfit' attribute sub-directory is available to add NFIT-specifics.

Note that the kernel device name for "DIMMs" is "nmemX". The NFIT describes these devices via "Memory Device to System Physical Address Range Mapping Structure", and there is no requirement that they actually be physical DIMMs, so we use a more generic name.

LIBNVDIMM: DIMM (NMEM)

```
struct nvdimm *nvdimm_create(struct nvdimm_bus *nvdimm_bus, void *provider_data,
                const struct attribute_group **groups, unsigned long flags,
                unsigned long *dsm_mask);
```

```
/sys/devices/platform/nfit_test.0/ndbus0
|-- nmem0
|   |-- available_slots
|   |-- commands
|   |-- dev
|   |-- devtype
|   |-- driver -> ../../../../../bus/nd/drivers/nvdimm
|   |-- modalias
|   |-- nfit
```

```
|   |   |-- device
|   |   |-- format
|   |   |-- handle
|   |   |-- phys_id
|   |   |-- rev_id
|   |   |-- serial
|   |   `-- vendor
|   |-- state
|   |-- subsystem -> ../../../../../bus/nd
|   `-- uevent
|-- nmem1
[..]
```

LIBNDCTL: DIMM enumeration example

Note, in this example we are assuming NFIT-defined DIMMs which are identified by an "nfit_handle" a 32-bit value where:

- Bit 3:0 DIMM number within the memory channel
- Bit 7:4 memory channel number
- Bit 11:8 memory controller ID
- Bit 15:12 socket ID (within scope of a Node controller if node controller is present)
- Bit 27:16 Node Controller ID
- Bit 31:28 Reserved

```
static struct ndctl_dimm *get_dimm_by_handle(struct ndctl_bus *bus,
       unsigned int handle)
{
        struct ndctl_dimm *dimm;

        ndctl_dimm_foreach(bus, dimm)
                if (ndctl_dimm_get_handle(dimm) == handle)
                        return dimm;

        return NULL;
}

#define DIMM_HANDLE(n, s, i, c, d) \
        (((n & 0xfff) << 16) | ((s & 0xf) << 12) | ((i & 0xf) << 8) \
         | ((c & 0xf) << 4) | (d & 0xf))

dimm = get_dimm_by_handle(bus, DIMM_HANDLE(0, 0, 0, 0, 0));
```

LIBNVDIMM/LIBNDCTL: Region

A generic REGION device is registered for each PMEM interleave-set / range. Per the example there are 2 PMEM regions on the "nfit_test.0" bus. The primary role of regions are to be a container of "mappings". A mapping is a tuple of <DIMM, DPA-start-offset, length>.

LIBNVDIMM provides a built-in driver for REGION devices. This driver is responsible for all parsing LABELs, if present, and then emitting NAMESPACE devices for the nd_pmem driver to consume.

In addition to the generic attributes of "mapping"s, "interleave_ways" and "size" the REGION device also exports some convenience attributes. "nstype" indicates the integer type of namespace-device this region emits, "devtype" duplicates the DEVTYPE variable stored by udev at the 'add' event, "modalias" duplicates the MODALIAS variable stored by udev at the 'add' event, and finally, the optional "spa_index" is provided in the case where the region is defined by a SPA.

LIBNVDIMM: region:

```
struct nd_region *nvdimm_pmem_region_create(struct nvdimm_bus *nvdimm_bus,
                struct nd_region_desc *ndr_desc);
```

```
/sys/devices/platform/nfit_test.0/ndbus0
|-- region0
|   |-- available_size
|   |-- btt0
|   |-- btt_seed
|   |-- devtype
|   |-- driver -> ../../../../../bus/nd/drivers/nd_region
|   |-- init_namespaces
|   |-- mapping0
|   |-- mapping1
|   |-- mappings
|   |-- modalias
|   |-- namespace0.0
|   |-- namespace_seed
|   |-- numa_node
|   |-- nfit
|   |   `-- spa_index
|   |-- nstype
```

```
|   |-- set_cookie
|   |-- size
|   |-- subsystem -> ../../../../../bus/nd
|   `-- uevent
|-- region1
[..]
```

LIBNDCTL: region enumeration example

Sample region retrieval routines based on NFIT-unique data like "spa_index" (interleave set id).

```
static struct ndctl_region *get_pmem_region_by_spa_index(struct ndctl_bus *bus,
                unsigned int spa_index)
{
        struct ndctl_region *region;

        ndctl_region_foreach(bus, region) {
                if (ndctl_region_get_type(region) != ND_DEVICE_REGION_PMEM)
                        continue;
                if (ndctl_region_get_spa_index(region) == spa_index)
                        return region;
        }
        return NULL;
}
```

LIBNVDIMM/LIBNDCTL: Namespace

A REGION, after resolving DPA aliasing and LABEL specified boundaries, surfaces one or more "namespace" devices. The arrival of a "namespace" device currently triggers the nd_pmem driver to load and register a disk/block device.

LIBNVDIMM: namespace

Here is a sample layout from the 2 major types of NAMESPACE where namespace0.0 represents DIMM-info-backed PMEM (note that it has a 'uuid' attribute), and namespace1.0 represents an anonymous PMEM namespace (note that has no 'uuid' attribute due to not support a LABEL)

```
/sys/devices/platform/nfit_test.0/ndbus0/region0/namespace0.0
|-- alt_name
|-- devtype
|-- dpa_extents
|-- force_raw
|-- modalias
|-- numa_node
|-- resource
|-- size
|-- subsystem -> ../../../../../../bus/nd
|-- type
|-- uevent
`-- uuid
/sys/devices/platform/nfit_test.1/ndbus1/region1/namespace1.0
|-- block
|   `-- pmem0
|-- devtype
|-- driver -> ../../../../../../bus/nd/drivers/pmem
|-- force_raw
|-- modalias
|-- numa_node
|-- resource
|-- size
|-- subsystem -> ../../../../../../bus/nd
|-- type
`-- uevent
```

LIBNDCTL: namespace enumeration example

Namespaces are indexed relative to their parent region, example below. These indexes are mostly static from boot to boot, but subsystem makes no guarantees in this regard. For a static namespace identifier use its 'uuid' attribute.

```
static struct ndctl_namespace
*get_namespace_by_id(struct ndctl_region *region, unsigned int id)
{
        struct ndctl_namespace *ndns;

        ndctl_namespace_foreach(region, ndns)
                if (ndctl_namespace_get_id(ndns) == id)
                        return ndns;

        return NULL;
}
```

LIBNDCTL: namespace creation example

Idle namespaces are automatically created by the kernel if a given region has enough available capacity to create a new namespace. Namespace instantiation involves finding an idle namespace and configuring it. For the most part the setting of namespace attributes can occur in any order, the only constraint is that 'uuid' must be set before 'size'. This enables the kernel to track DPA allocations internally with a static identifier:

```
static int configure_namespace(struct ndctl_region *region,
                struct ndctl_namespace *ndns,
                struct namespace_parameters *parameters)
{
        char devname[50];

        snprintf(devname, sizeof(devname), "namespace%d.%d",
                        ndctl_region_get_id(region), paramaters->id);

        ndctl_namespace_set_alt_name(ndns, devname);
        /* 'uuid' must be set prior to setting size! */
        ndctl_namespace_set_uuid(ndns, paramaters->uuid);
        ndctl_namespace_set_size(ndns, paramaters->size);
        /* unlike pmem namespaces, blk namespaces have a sector size */
        if (parameters->lbasize)
                ndctl_namespace_set_sector_size(ndns, parameters->lbasize);
        ndctl_namespace_enable(ndns);
}
```

Why the Term "namespace"?

1. Why not "volume" for instance? "volume" ran the risk of confusing ND (libnvdimm subsystem) to a volume manager like device-mapper.
2. The term originated to describe the sub-devices that can be created within a NVME controller (see the nvme specification: https://www.nvmexpress.org/specifications/), and NFIT namespaces are meant to parallel the capabilities and configurability of NVME-namespaces.

LIBNVDIMM/LIBNDCTL: Block Translation Table "btt"

A BTT (design document: https://pmem.io/2014/09/23/btt.html) is a personality driver for a namespace that fronts entire namespace as an 'address abstraction'.

LIBNVDIMM: btt layout

Every region will start out with at least one BTT device which is the seed device. To activate it set the "namespace", "uuid", and "sector_size" attributes and then bind the device to the nd_pmem or nd_blk driver depending on the region type:

```
/sys/devices/platform/nfit_test.1/ndbus0/region0/btt0/
|-- namespace
|-- delete
|-- devtype
|-- modalias
|-- numa_node
|-- sector_size
|-- subsystem -> ../../../../../bus/nd
|-- uevent
`-- uuid
```

LIBNDCTL: btt creation example

Similar to namespaces an idle BTT device is automatically created per region. Each time this "seed" btt device is configured and enabled a new seed is created. Creating a BTT configuration involves two steps of finding and idle BTT and assigning it to consume a namespace.

```
static struct ndctl_btt *get_idle_btt(struct ndctl_region *region)
{
        struct ndctl_btt *btt;

        ndctl_btt_foreach(region, btt)
                if (!ndctl_btt_is_enabled(btt)
                                && !ndctl_btt_is_configured(btt))
                        return btt;

        return NULL;
}

static int configure_btt(struct ndctl_region *region,
                struct btt_parameters *parameters)
{
        btt = get_idle_btt(region);
```

```
        ndctl_btt_set_uuid(btt, parameters->uuid);
        ndctl_btt_set_sector_size(btt, parameters->sector_size);
        ndctl_btt_set_namespace(btt, parameters->ndns);
        /* turn off raw mode device */
        ndctl_namespace_disable(parameters->ndns);
        /* turn on btt access */
        ndctl_btt_enable(btt);
}
```

Once instantiated a new inactive btt seed device will appear underneath the region.

Once a "namespace" is removed from a BTT that instance of the BTT device will be deleted or otherwise reset to default values. This deletion is only at the device model level. In order to destroy a BTT the "info block" needs to be destroyed. Note, that to destroy a BTT the media needs to be written in raw mode. By default, the kernel will autodetect the presence of a BTT and disable raw mode. This autodetect behavior can be suppressed by enabling raw mode for the namespace via the ndctl_namespace_set_raw_mode() API.

Summary LIBNDCTL Diagram

For the given example above, here is the view of the objects as seen by the LIBNDCTL API: