NVM Direct API

Version 0.3 - 10/31/2015

This document describes NVM Direct, an open source implementation of a C API to simplify application development for byte addressable NVM. This API has been developed by Oracle. There are four major components for the API:

1. A precompiler that inputs a source file containing C extensions for NVM, and outputs a standard C source file, plus a USID (Unique Symbol ID) definition file. A Unique Symbol ID is a 128 bit random number that can be stored in NVM and used as if it was the value of a global symbol generated by the link editor. A USID is bound to the symbol value at runtime.

2. A utility to accept multiple USID definition files and output a C source file to compile and link with the application. This will define a map for the application to interpret USID's in NVM regions.

3. A library with code to implement NVM regions, heaps, locks, and transactions. Some of the functions are called by code inserted by the precompiler, and some are called by the application code.

4. A library and header file to provide service functions for the NVM library. This is customized for various environments that need to support NVM. There will be a Linux service library to support a single multithreaded user mode process, for example.

Table of Contents

Design Motivation

Reboot is the most powerful recovery tool that we have today. A reboot throws away all DRAM contents, including any corruption, and rebuilds DRAM from disk. There are many subtle bugs in existing production code that can cause DRAM corruption, but are rarely encountered. An OS panic or application crash will recover from these problems when they do occur. However NVM survives reboot and is just as susceptible to corruption as DRAM. This is going to be a big problem.

Applications that directly store data into NVM must flush the data out of the processor caches to ensure it is actually stored in NVM. Failure to code the flush will result in a bug that is very rarely encountered since data tends to eventually be flushed out of the processor cache. On rare occasions such a bug will result in a corrupt NVM data structure at a customer production site. This is so rare that it is unlikely to be caught in testing.

Failure to generate undo can also lead to subtle bugs that will be difficult to find in testing.

Prevention and early detection of corrupt NVM data structures is a primary goal of the API. Thus there are several features in this API that are designed to catch bugs at compile time, automate some operations, avoid some problems, and provide redundant data for runtime checking.

Another goal is to make NVM code easy to read and similar to code for data structures kept in DRAM. It needs to be clear which code is modifying NVM, both to the compiler and to a person browsing the code. C extensions are a requirement for writing robust NVM applications.

Functionality

This section gives an overview of the functionality provided for managing data kept in NVM.

Execution and Memory Model

The Open NVM API as released by Oracle supports single process multi-threaded user mode applications. It is possible to create a service library that does not supply multi-threading thus limiting

itself to single threaded applications. It should be possible to create a service layer that allows the API to be used within an operating system.

The API presumes that every thread will exit cleanly after releasing its resources. If a thread dies unexpectedly the application process should exit. Application death at any point will leave the NVM in a recoverable state. The next process to attach to the NVM will recover it to be consistent.

All execution threads are presumed to live in the same address space. Any DRAM allocated by one thread will be visible to all threads, except with the possibility of per thread data. The service layer provides dynamic allocation of thread local storage that may be visible to only the requesting thread, but is usually global.

NVM Region

An NVM region is a file in an NVM file system that is mapped into a process's virtual address space for direct load/store access to the NVM backing the file. The NVM file system is responsible for providing access control and zeroing NVM before it is added to a file. The Linux Ext4 file system can be used for this purpose. An NVM region is generally a sparse file that contains all the related data for a single application. An application can map more than one NVM region, but each region is independently managed.

The NVM library provides the ability to create new NVM region files and map them into a process. The application must create a root struct for the region to make it fully formed and suitable for later remapping into another process. An exclusive lock is acquired on a region file when it is attached to a process so that only one process at a time can attach to a region.

An NVM region is created with a virtual size that defines the amount of virtual address space it consumes when mapped into a process. The virtual size is typically much larger than the amount of NVM that will ever be in the file. Creation also takes a physical size for the base extent of NVM which starts at byte 0 of the file. Additional extents can be added or deleted from the region as needed. The base extent cannot be deleted. All extents can be resized. When an NVM region is mapped into a process, a region descriptor is returned to identify the region to NVM library functions.

NVM is attached to a single server since it is connected as memory. This means that the NVM file system is local to that server, and its region files can only be mapped into processes executing on that server.

NVM Extents

Applications may manage multiple extents of NVM that can be independently created, deleted, grown, and shrunk. Related data can go into the same extent. This makes it easier to delete all the related data by deleting the extent. If there is corruption in one extent it is unlikely to spread to another extent. This is an important failure isolation feature. Each extent can have its own heap. If all application data had to come from the same heap then corruption of the heap could lose all the data.

Each extent is a range of NVM pages assigned to an offset in an NVM file. To improve performance the pages should be as large as possible. This means that the extent size should be a multiple of the desired page size and start at a file offset that is a multiple of the page size. The NVM file is thus a sparse file. When it is mapped into a process, the addresses between the extents are set to disallow load or store access. This is important to catch bugs that cause accesses to wrong locations.

An extent could be used to contain an NVM heap, or it could contain client data that is not managed by the NVM library.

NVM Heaps

When an NVM region is created the base extent is formatted as an NVM heap. The root struct is allocated from the heap in the base extent. Additional application data is allocated as needed. Additional heaps can be created and deleted as other region extents. Each heap can be given a name for administrative purposes.

USID

USID stands for Unique Symbol IDentifier. It is a 128 bit random number that can be stored in a persistent struct to represent an address defined by an executable at link time. At run time a map of USID to global symbol values is constructed so that the USID can represent a different address if the executable is relinked.

NVM Structs

A struct is declared to be either in persistent memory or volatile memory. A volatile struct is any struct that is not declared with the keyword “persistent”. A persistent struct can only be allocated from an NVM heap, and a volatile struct cannot be allocated in NVM except as a field in a persistent struct. A volatile struct used as a field in a persistent struct cannot have any pointers in it. A persistent struct has some additional attributes that can be set to assist in managing NVM.

For each persistent struct there is a global data structure that describes the attributes of the struct and all its fields. The data structure is a const struct in the executable with a global symbol based on the struct name. Usually there is a USID for the global symbol which is stored as the first field of the persistent struct. The struct description is used to allocate instances of the struct from an NVM heap. The heap allocator uses the description to initialize the allocated struct.

A persistent struct that ends in a zero length array is an extensible struct. The actual size of the array can be specified when allocating the struct. A persistent struct type that is not extensible can be allocated as a contiguous array by passing an array size greater than one when allocating. An extensible struct cannot be allocated as an array since the entries would not be of uniform size.

NVM Locking

The NVM library introduces the concept of NMV mutexes for coordinating access to NVM data in the same region as the mutex. NVM mutexes are declared as members of a persistent struct. NVM locks are

similar to pthread mutex locks except that they are owned by a transaction rather than a thread of execution. They are released when the owning transaction commits, or if it rolls back all of the undo generated after the lock was acquired. Rollback can be the result of an abort or rollback to savepoint.

Since NVM locks are in NVM and associated with a transaction rather than a thread of execution, they survive process death and/or unmap of a region. When the region is reattached, recovery will release all NVM locks as part of recovering all transactions.

NVM locks may be shared or exclusive. They may be acquired wait, no wait, or with a timed wait.

To aid in deadlock avoidance, each NVM mutex has a level associated with it. A lock cannot be acquired with waiting if a lock of the same or higher level is already owned by the transaction. There is a lock initialization call that must be made to set the level.

NVM Transactions

Unlike volatile memory, NVM retains its contents through a power failure or processor reset. This can happen at any time. If it happens in the middle of updating a data structure such as a doubly linked list, then a mechanism is needed to restore the linked list to a consistent state. This problem can also arise when an application thread throws an error or dies while manipulating persistent data structures.

A transaction mechanism is needed to atomically modify an arbitrary set of locations in NVM. Before a transactional store to NVM, the old value of the location will be saved in an NVM transaction data structure. This is the undo to apply if the transaction aborts or partially rolls back. When all stores for the transaction are complete, the undo is deleted thus committing the transaction. The application may choose to abort a transaction applying all the undo generated so far. An application may also create a savepoint and later apply all undo generated after the savepoint was created. This is a partial rollback of the transaction.

In addition to the above description of saving old values, we need the ability to associate NVM locks with the transaction. A multi-threaded application uses NVM locks to ensure only one thread at a time is modifying an NVM data structure. Such a lock needs to be held until any undo that modifies the data structure is either applied by rollback/abort or discarded by transaction commit. Thus the lock is associated with the transaction rather than the thread that acquired the lock. Since the transactions survive process death the locks need to be in NVM just as the transaction data.

When an application attaches to an NVM region, the region is checked to see if there are any transactions that need recovery. Recovery is completed before the attach function returns to the application. This ensures the application only sees a consistent state in NVM after a failure of the previous process to attach to the region. If recovery cannot be completed due to some error, then the attach fails. Note that this means a transaction can only modify one NVM region at a time because recovery happens when the region is attached and regions are attached one at a time. It is not possible to atomically modify data in two different regions.

On abort operations

An application can create an operation to be executed if the active transaction rolls back through the current point in the undo. This creates an undo record with a pointer to a persistent application function and a persistent struct to pass to the function. If the undo is applied, the function is called in a new nested transaction. The nested transaction is committed when the function returns. This can be used to implement logical undo to reverse the effects of a nested transaction that committed. On abort operations are also useful for undoing OS operations, such as deleting a file that was created during a transaction if the transaction aborts.

Creating an on abort operation returns a pointer to the struct in the undo record so that arguments to the function can be modified later. It is common to have a nested transaction do a transactional store into the on abort operation so that rollback does nothing unless the nested transaction commits.

Note that the function must not signal any errors, exit the process, or abort the nested transaction. This could result in an NVM region that could never be attached since every time recovery is attempted it would fail. Thus the only errors must be the result of serious NVM corruption.

Note that the function could be called at recovery time so it must not attempt to access any volatile memory data structures created by the process that created the on abort operation.

On commit operations

An application can create an on commit operation using a mechanism that is the same as creating an on abort operation except that the undo record does nothing if rolled back. All the on commit operations in a transaction are executed after the transaction has committed and all its locks released. While undo is applied in the reverse of the order it is created in, on commit operations are executed in the same order as they are created. Each on commit function is called in its own nested transaction that is committed when it returns.

On commit operations are useful for delaying the release of resources until the current transaction is guaranteed to commit. This can avoid generating large amounts of undo or acquiring locks that could cause deadlocks. On commit operations are also useful for delaying OS operations, such as deleting a file only if the transaction commits.

An on commit operation must not signal any errors, exit the process, or abort the nested transaction. This would make it impossible to complete the commit of a committed transaction. If an error is signaled or the nested transaction aborts, the application process exits. Recovery will first rollback the nested transaction then resume commit of the parent transaction beginning with the on commit operation that failed. If it fails again the region cannot be attached.

Note that the function could be called at recovery time so it must not attempt to access any volatile data structures created by the process that created the on commit operation.

On unlock operations

An application can create an on unlock operation using a mechanism that is the same as creating an on abort or on commit operation except that the undo record is applied both when its transaction aborts or when it commits. When a transaction aborts all undo records including on unlock records are applied in reverse order. When a transaction commits all its locks are released in reverse order. An on unlock operation is treated like an nvm_mutex unlock record and applied while previously acquired locks are still held. Each on unlock function is called in its own nested transaction that is committed when it returns.

An on unlock operation is useful for delaying an operation until the fate of a transaction is known and to ensure a previously acquired lock is still held. On unlock operations could also be used to implement application specific locks in NVM. Most applications will not use on unlock operations. The NVM library uses one to implement nvm_set_txconfig because it changes transaction metadata.

An on unlock operation must not signal any errors, exit the process, or abort the nested transaction. This would make it impossible to complete the commit or abort of a transaction. If an error is signaled or the nested transaction aborts, the application process exits. Recovery will first rollback the nested transaction then resume commit or abort of the parent transaction beginning with the on unlock operation that failed. If it fails again the region cannot be attached.

Note that the function could be called at recovery time so it must not attempt to access any volatile data structures created by the process that created the on commit operation.

Open nested transactions

The NVM library implements open nested transactions. Here is a paper with more details: Open Nested Transactions: Semantics and Support. The equivalent of closed nested transactions is also available with savepoints and rollback to savepoint as described in the next section.

Nested transactions suspend operation of the current transaction so that all transactional operations happen in the nested transaction. The nested transaction can commit or abort independently from its parent transaction. When it commits, its changes are persistent even if the parent transaction aborts. The parent of a nested transaction can itself be a nested transaction.

Locks acquired by a nested transaction are released when the nested transaction commits/aborts. This makes nested transactions useful for avoiding deadlocks and for reducing the time a lock is held so that there is less contention. For example allocation from a heap is done in a nested transaction. An on abort operation is created in the parent transaction before the nested transaction is started. If the allocation commits, but the parent transaction aborts, then the on abort operation undoes the committed allocation.

On commit operations created during a nested transaction are executed when the nested transaction commits. The parent transaction does not become the current transaction until all the on commit operations complete and the nested transaction code block is exited.

A nested transaction may operate on a different region than its parent.

Rollback to Savepoint

A point in the current transaction can be marked with a savepoint. A savepoint can be identified with an address that can be set after the savepoint is created. Rollback to savepoint can be done later within the same transaction. This will undo the transactional changes since the savepoint was created, and release any NVM locks acquired by the transaction since the savepoint was created. There may be multiple savepoints so rollback can specify the address that was set as the savepoint identity.

One use of rollback to savepoint is to release share locks that are no longer needed. Another use is to undo changes so an operation can be tried again.

Setting a savepoint is equivalent to beginning a closed nested transaction, and rollback is equivalent to aborting a closed nested transaction.

Explicit Commit/Abort

The current transaction can be explicitly committed or aborted by calling a library function. The abort function returns when all undo has been applied up to the beginning of the transaction. The commit function returns when all on commit operations have completed. In either case all locks acquired by the current transaction are released.

Once a transaction is explicitly committed/aborted it is still the current transaction, but it cannot be used for any more transactional operations. Explicit commit/abort of a nested transaction does not make the parent transaction current until the nested transaction is ended. A transaction is defined by a specially annotated code block. The transaction stays current until the code block is exited either by an explicit transfer, executing out the bottom of the block, or an NVM longjmp to an NVM setjmp outside the code block.

There is a library routine to report the current transaction nesting level and the current transaction status. A current transaction can be active, committed, or aborted. A nesting level of 0 means there is no current transaction. A nesting level of 1 means the base transaction is current.

NVM Precompiler

The NVM precompiler extends the C language to support NVM access for code that uses the NVM API. The NVM precompiler takes C preprocessor output from a source file with NVM extensions and converts the extensions to standard C which uses nvm.h declarations to implement the extra operations required for NVM access. The C extensions make the code easier to read, can catch coding errors that would not be enforceable in standard C, and automates code insertion that might be accidentally omitted.

The precompiler outputs a C file that has the same line number to source mapping as the original extended source. This allows debuggers to work with the original source. However current debuggers do not recognize the new C operators and may not be able to evaluate some expressions.

The precompiler outputs another file that contains a JSON description of all the USID's for global symbols that are either defined or referenced by the source file. This is known as a USID definition file. USID's for structs and symbols that are only declared are not included. All the USID definition files for a library or an application are given to the USID map utility to create a map of USID to symbol values in a C source file. The C source includes a function to register the USID to symbol mappings at startup. The application or library must call this function before the API is used.

Command Line

`clang -n -Wno-invalid-preprocessed-input -xc` *`ifile`* `-o` *`ofile`*

`-xc` *`ifile`* The precompiler reads the C source file with C extensions from *ifile*. This is the output from the preprocessor. The –E option to most C compilers will output the preprocessed source file. By convention this usually has the file extension ".ix" set with the –o option to the preprocessor.

`-o` *`ofile`* The precompiler writes the C99 source output to *ofile* and the USID definition file to *ofile* with the file extension changed to ".usd". By convention *ofile* usually has the file extension ".i" since that is the default for the preprocessor output. If *ofile* is not specified, then the C source is written to standard out and the USID definition file to *ifile* with the file extension changed to ".usd".

`-Wno-invalid-preprocessed-input` This is needed because gcc leaves a bunch of #define directives in its preprocessed output.

New Syntax

This section describes the new syntax enabled by the precompiler.

NVM Addresses

It is critical that the compiler knows which pointers contain NVM addresses and which contain volatile memory addresses. Dereferencing a pointer to NVM sometimes requires additional code. There is new syntax to identify pointers to NVM and for dereferencing them. This makes it clear to the programmer and code reader that this is an NVM access. The new syntax is similar to standard C syntax for pointers so that it is easier to understand. Using macros or inline functions for pointer dereferencing results in cluttered confusing code.

Declaring a pointer to a location in NVM uses "^" rather than "*" to indicate it is a pointer to NVM. Thus pointer to int in NVM and pointer to int in DRAM are different types. Pointers with multiple levels of indirection could be declared with a mixture of "^" and "*".

- int ^*pt; // declares a pointer to DRAM that contains a pointer to NVM
- int *^pt; // declares a pointer to NVM that contains a pointer to DRAM

Note that the pointer itself could reside in either DRAM or NVM for either of the above declarations. It is in NVM if it within a persistent struct or located through a pointer to NVM, otherwise it is in DRAM.

Pointers in NVM to NVM are self-relative so that they will work if the NVM is mapped in at a different virtual address. This means the pointer does not contain a virtual address, but instead contains the difference between the location of the pointer and the address pointed at. The compiler automatically converts between self-relative pointers and absolute pointers when using them. Note that a null pointer is represented as a 1 since 0 is a pointer to self.

It is a compile error to store a pointer to DRAM into a pointer to NVM since the types do not match. Similarly it is a compile error to store a pointer to DRAM into a pointer to NVM. Naturally this applies to arguments to functions as well as assignment. However a const pointer to DRAM can have a pointer to NVM stored in it since it will never be used to modify NVM. For reading, NVM behaves just like DRAM – no flushing is needed. You can use casts to get around these rules, but it is a dangerous thing to do.

```
int *a;
int ^b;
const int *c;
const int ^d;
a = b; // compile error
b = a; // compile error
c = b; // no error
c = d; // no error
d = c; // compile error
```

Indirection through an expression for an address in NVM must use "^" rather than "*". Similarly "=>" must be used rather than "->" when accessing a struct member in NVM. The compiler can know from the type of the expression that the address is in NVM and generate an error if the wrong indirection operator is used. This is possible because NVM pointers are declared with "^". The language could allow "*" to be used for indirection through a pointer declared with "^", but this would look inconsistent. Requiring a different operator makes sure the programmer is aware that the code is working with NVM.

Taking the address of a location in NVM uses "%" rather than "&". Again this is to ensure the programmer is aware of the persistence. In C++ "%" would also be used for declaring references to NVM locations.

Note that the structure member operator "." does not have an NVM equivalent. This is not needed since there cannot be any variables that are in defined in NVM. NVM addresses are always calculated at runtime based on the address where the NVM region is mapped. Thus the only way "." could be used with NVM is if there was a "^" or "=>" in the expression indicating the programmer is aware it is in NVM.

NVM Assignment

The compiler needs to generate additional code for storing in NVM. Stores to NVM are not persistent until the affected cache line is copied from the processor cache to the NVM DIMM. This requires generating code to track the modified cache lines and eventually flush the cache for every store.

NVM requires atomic transactions to maintain consistent data structures. Some stores to NVM need to be transactional so that they do not happen if the transaction does not commit. Other stores are non-transactional to avoid the transaction overhead or because they are outside a transaction. Non-transactional stores are often used when initializing a struct allocated in the current transaction since the allocation will be undone if the transaction does not commit. It is up to the programmer to indicate which kind of store is being done. To ensure the programmer has made a conscious choice there are new assignment operators that must be used when storing into NVM. This also makes it clear that the assignment is to NVM not DRAM.

A transactional store uses the normal assignment, increment, or decrement operator with "@" in front of it, and non-transactional is indicated with "~". Thus the transactional operators are:

```
@=   @+=   @-=   @*=   @/=   @%=   @<<=   @>>=   @&=   @|=   @^=   @++   @--
```

The non-transactional operators are:

```
~=   ~+=   ~-=   ~*=   ~/=   ~%=   ~<<=   ~>>=   ~&=   ~|=   ~^=   ~++   ~--
```

These look similar to the standard operators so the code is easier to read. It is a compile error to use a standard operator for an lvalue in NVM.

On some systems any modified NVM cache lines must be flushed from the cache to ensure the new value becomes persistent. It is too much to expect the programmer to add a function call for every flush. It would be easy to mistakenly miss a flush, and the chances of such a mistake being noticed are vanishingly small. Usually normal aging out of the cache would make the store persistent. Therefore the precompiler requires any store to NVM to be either transactional or non-transactional so that code can be added to flush the cache after the store. Note that it is a compile error to do a transactional store outside transactional code, but a non-transactional store can be done inside or outside a transaction. The flush code is necessary even if there is no current transaction.

Assignment to a pointer in NVM that points to NVM converts the pointer into a self-relative pointer automatically for both transactional and non-transactional stores. A self-relative pointer cannot point outside of the NVM region it is in. If there is an active transaction for the region containing the pointer, there is a runtime check to ensure the pointer is within the region. It is too expensive to do this check when there is no active transaction.

Note that a memcpy of NVM containing pointers corrupts the pointers since the relative offset is no longer correct. Struct assignment from NVM will copy the pointers correctly. If assigning to another NVM struct the offsets are corrected. If assigning to DRAM the pointer is made absolute. Similarly assigning a struct in DRAM to a struct in NVM will convert the absolute pointers to NVM into self-relative pointers.

Undo is automatically generated by a transactional store operator, but undo can also be explicitly generated. Explicit undo may be useful for generating undo more efficiently. Creating one undo record for a small struct will be faster than creating an undo record for each assignment to its fields. Once the

struct undo is created the assignments can be non-transactional. The library function

```
nvm_undo(void ^addr, size_t bytes);
```

will generate undo for the indicated NVM. There is a runtime check to ensure addr is in the same NVM region as the transaction.

Legacy Functions

There may be cases where a legacy function needs to be called to store results in NVM. This can be done as long as the legacy code does not allocate any memory that should be NVM or store any pointers in NVM. A good example of such a function is a matrix multiply where the output goes to NVM. There are four issues that need to be dealt with to do this.

1. Undo generation: Explicit undo generation for the locations to be modified must be done before the legacy function is called. The library function
   ```
   nvm_undo(void ^addr, size_t bytes);
   ```
   can generate the undo. This is not necessary if the NVM to be modified was allocated in the current transaction and will be rolled back if it aborts.

2. Casting to volatile memory pointer: Legacy functions will take pointers to volatile memory – i.e. their pointer arguments are declared with "*" rather than "^". An explicit cast is needed to indicate the programmer knows an NVM pointer is being treated as a volatile memory pointer. The cast "(*)" changes the expression from an NVM pointer to a volatile memory pointer of the same type. Note that const pointers do not need to be cast since they can be implicitly cast.

3. Explicit flush from cache: Legacy functions do not flush cache lines out of the cache when storing through a pointer argument. After the legacy function returns, any modified cache lines must be explicitly flushed. The function
   ```
   nvm_flush(void ^addr, size_t bytes);
   ```
   does an explicit flush of the indicate NVM. Since flushing does not change the value associated with a memory location, there is no correctness problem with unnecessary flushes.

4. Force persistence: Flushing, explicit or compiler generated, just schedules the flush. It does not ensure the flushed data is actually persistent. If the legacy function is called outside a transaction then the caller needs to ensure the flushes complete before updating data to indicate the flushed data is usable. The library function `nvm_persist()` will block until all flushed data is truly persistent. If this legacy function is called in a transaction then this is not necessary since the commit of the transaction is preceeded by a persist barrier to ensure all transactional stores are persistent.

USID

USID stands for Unique Symbol IDentifier. It is a 128 bit true random number based on some physical phenomenon. These are defined in the source code by an attribute of the form USID("xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx") where the x characters are hex digits in 8 groups of 4 separated by white space. You can click on this link to get 8 truly random hex values of 16 bits each separated by white

space that you can paste into your code. About one time in 7300 the random number will not conform to the restrictions imposed on a USID. If this happens then you will get a compile error and must choose another random number. You can avoid most invalid USID values by not using a random number where one group of 4 characters is “0000” or “ffff”. If you do not see any bytes that start with 8, 9, a, b, c, d, e, or f, then it will also be rejected.

A USID is similar to a GUID or UUID, but has some differences. It does not conform to RFC4122 because all bits are randomly chosen. However about one in 7,300 random numbers are rejected. This ensures a USID is different depending on the bit field ordering and endianess of the platform. However, most values are rejected to reduce the chance of a USID matching application data. There are three reasons that a 128 bit random number would be rejected as a USID.

1. At least one byte must have a sign bit set to eliminate ASCII strings looking like a USID.
2. There must not be any uint16_t values that are 0x0000 or 0xFFFF. Many application data values are in fields that are larger than the actual value and are thus sign extended with 0 or 1 bits.
3. The final 8 bytes must not be a palindrome that is the same for both big and little endian platforms.

Any symbol defined in an executable (i.e. nm will display it) can be associated with a USID that can be used as the symbol value in NVM. This is accomplished by preceding the declaration of the symbol with a USID attribute. Note that the value must never be changed once chosen, since that could make existing NVM regions corrupt. However the struct name might be changed so a new struct with the same name but different USID can be defined for in place upgrade.

At run time a hash table is constructed to map the USID values to symbol values. Normally “&<symbol>” is an expression that evaluates to the symbol value. The USID of a symbol is represented by the expression “|<symbol>”. It is a link time error if the symbol does not have a USID defined. A variable can be declared as a USID pointer by preceding the variable name with “|”. For example, “int |iptr;” would declare a pointer to an integer in volatile memory that is stored as a USID. The pointer is bound to a real address at run time by looking up the USID in the hash table. Each process may have a different value for the symbol since it might have a different executable.

Since USID values are random numbers it is theoretically possible that there could be two values that are the same. This is detected when building the hash table at process startup. The process will exit if there are two attempts to register the same USID with different symbol values. Thus any conflict will be detected long before the code is debugged. In any case, the chances of two independently chosen USID values being the same is vanishingly small. If you need a thousand USID’s for your application, then the chance of any two being the same is less likely than the earth being simultaneously hit by two independent extinction event asteroids in the same second on Friday Jan 13, 2113 at 13:13:13 GMT. This is based on the presumption that such an asteroid hits the earth about once every 128 million years, or 2**52 seconds.

Persistent struct assignment will copy the USID from the source struct after it is validated. The target does not need to contain a valid USID before the assignment. This allows copying a persistent struct through a pointer cast to the correct type. If persistent struct assignment is copying between regions then any self-relative pointers will be set to null. Note that this is different than assigning a pointer from one region to another in that pointer assignment will fire an assert if the address is in another region.

Persistent struct

A struct declaration can be preceded by the keyword “persistent”. This declares a struct that can only exist in NVM. A persistent struct cannot be a member of a non-persistent struct. However a non-persistent struct can be a member of a persistent struct. A non-persistent struct embedded in a persistent struct is considered effectively persistent and must follow the restrictions placed on persistent structs.

Forward declarations and type references to persistent structs must also include the “persistent” keyword. For example if there is a persistent struct named “mystruct”, the following typedef would be used to make “mystruct” a type:

```
typedef persistent struct mystruct mystruct;
```

Restrictions

A persistent struct cannot be anonymous. There must be a name after the keyword “struct”. This name is used to construct a global symbol associated with the struct. Thus all persistent struct names in a single executable must be unique, just like function names. This is not a requirement for effectively persistent structs.

A persistent struct or effectively persistent struct may not have any members that are bit fields. This limitation is because the implementation of bit field varies from one compiler to the next. In DRAM this is not a problem because recompiling a program with a different compiler necessarily requires it to restart with new DRAM. However NVM survives through a recompile. Switching to a different compiler should not make all of an application’s existing region files corrupt. Shifting and masking can be used to implement data packing portably.

A persistent struct or effectively persistent struct cannot contain a persistent pointer to DRAM. DRAM pointers are not allowed because pointers to DRAM kept in NVM are meaningless when the NVM attached by a new process. However a transient pointer to DRAM is allowed.

Transient

A field in any struct can be declared “transient” to indicate its contents are not expected to be meaningful through detach or application failure. A transient field may contain a pointer to volatile memory, or NVM. A transient pointer is never self-relative. A transient field is not flushed when updated, not affected by a persistent memory barrier, and cannot be transactionally modified. Assignment to a transient member does not use the NVM assignment operators. The compiler treats it

as if it was declared in volatile memory. A non-persistent struct can have transient members. This is useful for effectively persistent structs that really need pointers to DRAM.

Transient fields are useful for data that can be recalculated at reattach but is too expensive to recalculate on every use. For example a free list of NVM structs could use a transient link in each struct. At attach all of the structs could be scanned for those that are marked free building up the free list. At run time a struct could be added to the free list without the overhead of a transaction. Transient data is data that could be stored in volatile memory, but is easier and more efficient to keep in the NVM struct it is associated with. An attach count can be used to recognize the first access to an object after the NVM is attached by a new process.

Attributes

A persistent struct declaration may optionally have a number of attributes of the form keyword(args) following the struct name and before the opening "{". The following attributes are supported.

- USID("xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx"): This defines the USID for the nvm_type struct describing this persistent struct. The x characters are 32 hex digits with any amount of optional white space. Every instance of the struct will start with a hidden member containing this value.
- tag("description") : This provides a verbose description of the struct. It will be used for describing instances of the struct when found in NVM while debugging.
- version(uint32_t): This is the version number of the first software release to support this struct. If no version is given then zero is used.
- alignas(uint32_t):This takes a compile time constant giving the required alignment of an instance of the struct. It must be a power of 2. The actual required alignment may be greater if there is an embedded struct with a greater alignment. Allocating a persistent struct from an NVM heap will honor the alignment.
- upgrade(<struct type>, int (*func@)(<this struct type> ^ptr)): This defines an upgrade function that can do an in place conversion of this struct to a <struct type>. The new struct type must be the same size as this struct type. The function pointer will be used to upgrade a struct of this type to a new type when the old type is found. It will always be called in the context of a transaction that commits on its return.
- size(size_t) : The struct must be this size. The size must be a compile time constant. It is padded with zeroes if the declarations do not consume this much space. It is a compile error if it does not fit. This is useful for leaving space to add new members in a future software release.

Corruption Detection

A persistent struct with a USID attribute starts with a hidden member that contains the type USID of the struct. There is no need to access this field explicitly. It is initialized when the struct is allocated. The use of a pointer to a persistent struct with a USID attribute validates that the correct USID is at the

beginning of the struct. Failure usually indicates that the pointer is corrupt, the struct is corrupt, or the struct was freed and the pointer is stale. This will fire a corruption assert.

The USID check is useful for detecting and possibly preventing corruption of NVM. Software corruption of NVM is going to be a serious problem. Reboot is the most powerful recovery tool that we have today. A reboot throws away all DRAM contents, including any corruption, and rebuilds DRAM from disk. There are many subtle bugs in existing production code that can cause DRAM corruption, but are rarely encountered. An OS panic or application restart will recover from these problems when they do occur. However NVM survives reboot and is just as susceptible to corruption as DRAM.

Upgrade

A mismatch with the expected USID is not a corruption if the USID found in NVM is registered with another persistent struct which can be upgraded to the expected type. The persistent struct registered with the USID must have an upgrade attribute giving an upgrade function to call. If in non-transactional code, a base transaction is created before calling the upgrade function. If in transactional code a nested transaction is created. If the upgrade function discovers that the struct is corrupt it returns zero to indicate failure. The upgrade function should do as much validation of the data as is reasonably possible to reduce the chance that the USID was wrong. Failure or death will abort the upgrade transaction, so it is important that all changes to the struct are transactional. When the upgrade function returns success the upgraded USID is stored in the struct and the transaction is committed. The comparison is retried possibly doing another upgrade. An upgrade lock is acquired by the upgrade transaction before calling the callback. This will cause a deadlock if another upgrade is started from the callback. Thus the callback should not look at any data outside of the struct being upgraded. Note that the region containing the struct being upgraded could be different from the region being modified by the current transaction.

Persistent Pointer to Global Symbol

A persistent struct may not contain a pointer to volatile memory except in a transient member. However it can contain a USID pointer for a symbol in volatile memory defined by the linker. Volatile memory pointers are wrong if the region is attached to a new process, but a USID can be mapped to the correct address at runtime by any executable that defines the symbol. This is commonly done to store a callback function pointer in NVM. To call the function the USID is looked up in the hash table to get the proper address in the currently running process. A USID pointer could also be used to find a global DRAM data structure.

Extensible Struct

A common technique in C is to end a struct declaration with an empty array. When the struct is allocated in memory it is allocated with enough space to hold the desired array size. This supports variable size structs. In this document this is called an extensible struct. Allocation of a persistent struct from an NVM heap supports allocation of an extensible struct with a non-zero array size for the empty array. Allocation also supports allocating a variable size contiguous array of non-extensible structs.

However you cannot allocate a contiguous array of extensible structs since they can have different sizes and the compiler will not do the correct index calculations.

Assignment of an extensible struct to another extensible struct of the same type presumes the size of the extensible array is zero. The array contents must be explicitly copied.

Type description

For every persistent struct or effectively persistent struct there is a const global of type nvm_type that describes the struct. This includes the values of all the persistent struct attributes, and the fields in the struct with their type and name. A field that is an embedded struct or a pointer to a struct will include a pointer to the nvm_type of the struct. An nvm_type is used for initializing an instance of the struct, and for allocation of a struct from an NVM heap. If the persistent struct has a USID then the address of the nvm_type is registered under the USID.

The nvm_type definitions are generated by the compiler for every persistent struct declared by the compile unit. This will often create multiple identical definitions since type declarations are often in .h files included in multiple .c files. The linker combines these definitions into a single read only instance much the same way as identical string constants are combined. An application may use dynamically loaded libraries or shared object libraries that define their own persistent struct types in the .h file defining the API. This means that both the library and the application will have nvm_type definitions for the persistent structs, but only one should be registered under the USID. The same may be true for functions that have USIDs defined for use as function pointers. Thus there is the concept of an owner for a particular USID and only the owner should register the USID to prevent conflicting registrations. Only the software developer knows the ownership of a USID and thus must be responsible for the registration. This means the language needs a means for the developer to express this relationship. This could be syntax to do the registration, or it could be arguments to a utility that generates the registration code.

Built-In functions

There are new built-in functions that work like sizeof(type) but require the lvalue or type to be a persistent struct.

- `const nvm_type *shapeof(type)`: This returns a pointer to the nvm_type constructed by the compiler to define the attributes and fields of the persistent struct type.
- `const char *tagof(type)` : This returns the tag string from the tag attribute. It returns a zero length string constant if there is no tag.
- `uint32_t versionof(type)` : This returns the value of the version attribute. It is zero if there is no version attribute.
- `nvm_usid usidof(type)`) : This returns the 16 byte USID from the USID attribute. It is zero if there is no USID attribute.

Persistent Union

A persistent union is a union of two or more persistent structs. Every member of the union must be a persistent struct that has a USID defined for it, or an array of persistent structs with a USID defined. This ensures the current union member type can be determined by examining the USID that appears as the first 16 bytes of every member type.

A persistent union cannot be directly allocated from an NVM heap, or used to initialize application managed NVM. This can be worked around with a persistent struct declared with only one field, which is a persistent union. If the struct has a USID defined it can be used to allocate an instance which consists of a USID followed by the persistent union. If the struct does not have a USID defined, then the struct description can still be used to initialize NVM not allocated from a heap. Alternatively the NVM can be initialized using the shapeof() for the desired member.

A persistent union is declared by putting the keyword “persistent” before “union”. As with a persistent struct it must have a name and the name must be unique within the application. However there are no attributes that can be defined for a persistent union. Just as a persistent struct has an automatically generated nvm_type describing it, a persistent union has an nvm_union automatically generated describing the union. The built-in function shapeof() can be used on a persistent union type or instance to get a pointer to the nvm_union for the persistent union.

Here is an example of a persistent union declaration:

```
persistent struct article
USID(“ee9e     a349    92d6    5e9e    87bd    0438    5503    7632”)
{
    char magazine[128]; // magazine that published the article
    int  words;         // size of article in words
};
typedef persistent struct article article;

persistent struct book
USID(“c55a     7577    7777    2025    e11e    7ae5    e010    9294”)
{
    char publisher[128]; // publishing company
    int  pages;          // size of book in pages
};
typedef persistent struct book book;

persistent union description
{
    article a;
    book    b;
}
typedef persistent union description description;

/* Either a magazine article or a book */
persistent struct publication
USID(“3cf3     d0d3    16c7    5044    96a1    9ca0    e471    e36d”)
{
    char author[128]; // author’s name
    char type;        // 0 if article, 1 if book
    description desc; // description based on type
}
```

This defines a persistent struct called “publication” which can describe either a magazine article or a book. An instance of publication can be allocated from an NVM heap. Note that it would be an error if either book or article did not define a USID.

Transaction Block

A base transaction is represented by a block of code preceded by “@<region>” where <region> is an expression that evaluates to the region descriptor returned from nvm_create_region or nvm_attach_region. A region descriptor is a small int similar to a file descriptor returned by open().

On entry to the code block a transaction is allocated for the indicated region and becomes the current transaction. If the thread already has a current transaction, then a nested transaction is begun, possibly in a different region. The transaction remains the current transaction until the code block is no longer active. Allocation of a new transaction might have to block for some other transaction to commit freeing up a transaction slot.

The transaction is committed if the code block is exited by any explicit means such as falling out the bottom, goto, break, or continue. The transaction is aborted if the code block is exited via a longjmp caught outside the code block. There are also function calls to explicitly commit or abort the transaction. If the transaction is explicitly aborted or committed, then exiting the code block has no effect other than making the transaction no longer current. Attempting to do any transactional operations after the transaction has ended is an error.

A nested transaction is syntactically the same as a base transaction except that the region descriptor is optional. The nested transaction operates on the same region as its parent when there is no descriptor. If there is a descriptor, the same code could be a base transaction when its function is called outside a transaction and a nested transaction when called inside a transaction.

The nested transaction becomes the current transaction until the code block is exited. When the code block is exited the parent transaction becomes current again. Note that the parent transaction does *not* become current if the nested transaction is explicitly committed or aborted. Thus an assert fires if a transactional operation is attempted, the transaction is completed (abort or commit), and the transaction code block is still executing.

Here is an example of a base transaction with nested transactions:

```
nvm_desc region;
int i;
.
.
.

// begin and commit a base transaction each time around the loop
for (i = 0; i<n; i++)
@ region {
```

```
        // A base transaction is now current
        .
        .
        .
        @{ // begin nested transaction
            .
            .
            .

            if (done)
                break; // commit parent and final nested transaction
        }
    }
```

Transactional Function

A function or function pointer can be declared to be transactional. A transactional function can only be called from a transaction code block or another transactional function. This allows compile time checking for inappropriate function calls. Note that a transactional function can be called even if the current transaction has ended, but this is unlikely to succeed since any transactional operation will fail.

A function is transactional if "@" follows the function name in the declaration and definition of the function. Similarly a transactional function pointer declaration has "@" following the pointer name. The pointer and the function it points to must have the same transactionality. Casting cannot change transactionality.

There are situations where the code knows that it is inside a transaction or not in a transaction based on some state such as a call to nvm_txdepth(). To allow calls to transactional functions from a function that was not declared transactional, a code block can be labeled as known to be in a transaction. If the opening curly brace is "extern @{" then the precompiler presumes the block is inside a transaction even if there is no other syntax to ensure it. This does not start a transaction. There are likely to be runtime errors if there actually is not a current transaction.

Functions that contain a transaction code block may be declared transactional meaning the code block is a nested transaction. If the function is not declared transactional then the code block will be a nested transaction if called during a transaction, and a base transaction if called when no transaction is active.

USID Function Address

A global function can be defined to have a USID to be used as a function pointer in a persistent struct or in NVM undo. The USID is defined by preceding the function declaration with a USID definition as can be done for any global symbol. Declare the function pointer with "|" rather than "*" to be a USID that will be bound to the function entry point at runtime. Use "|<function>" as the value of the USID to store in the function pointer. Note that in a multi-process application each process will have its own USID hash map since it may have a different address for the function than other processes.

The precompiler will recognize use of a function USID to call a function, and look it up in a hash table to find the entry point in the current executable. If there is no function registered under the USID, then the function pointer will be null.

The function USID of all zero represents a null function pointer, and evaluates to false if used as a Boolean.

A function with a USID and a pointer to a persistent struct as its only argument may be a callback function. It is suitable for passing to nvm_onabort, nvm_oncommit or nvm_onunlock. The persistent struct normally does not have a USID associated with it, but it may.

For example, the following declares a function that has a USID. This would usually appear in a .h file. :

```
USID("3191 c3be 57cf 5408 3492 134c df53 6df7")
int myfunc@(int ^loc, int n); // must be in a transaction
```

The body of the function could do the following to store an int and return its square. Note the USID is not here because it was declared above. The value of a USID does not need to be in more than one place in the code.:

```
int myfunc@(int ^loc, int n)
{
      ^loc @= n;   // store the argument
      return n*n;  // return the square
}
```

The following persistent struct could hold a USID as a pointer to the function and an array to store up to 50 integers followed by the result of calling the function. In this example the function is myfunc:

```
persistent struct mystruct
{
      int (|func@)( int ^loc, int n); // USID function pointer
      int cnt;
      int results[100];
}
typedef persistent struct mystruct mystruct;
```

The following code stores a USID function pointer into an instance of mystruct and clears the array. Non-transactional stores are used because the … code allocates the instance of mystruct so transaction abort would free the memory.

```
mystruct ^ms;
...
ms=>func ~= |myfunc; // store the function pointer
ms=>cnt ~= 0;        // empty array of function results
```

The following function is passed a pointer to an instance of mystruct, stores the results of one call and returns the new count of results. It must be called in a transaction. Note that the function code address is returned by indirection through the USID pointer.

```
int add_result@(mystruct ^ms, int x)
{
      int (*f)( int ^loc, int n) = *ms=>func; // get code address
      /* get and save one result */
      ms=>results[cnt+1] @= (*f)(%ms=>results[cnt], x);
      cnt @+= 2; // one more result pair stored
      return cnt;
}
```

Long Jump

A long jump that jumps to a setjmp that is outside an active transaction code block must abort the transaction. The standard setjmp/longjmp mechanism does not know about transactions and cannot accomplish this. The NVM library has alternative versions of setjmp, longjmp, and jmp_buf that take care of transaction abort if needed and can be used instead of the standard versions.

The compiler will add some code around the normal setjmp to save the current transaction depth in a local variable when creating the jmp_buf. When setjmp returns from a longjmp it calls a library routine to abort uncommitted transactions until the same transaction depth is achieved.

Error Checks

There are a number of coding errors that can be caught or prevented by the NVM precompiler which could not be caught or prevented without it.

- An expression cannot contain, compare, or subtract pointers with different persistence.
- A pointer to volatile memory cannot be declared in a persistent struct except as transient.
- Pointers to NVM cannot be stored as pointers to volatile memory. The precompiler can determine the persistence of every store.
- Function arguments can specify the persistence of any pointers. The actual arguments must have the correct persistence. However a pointer to const can be any persistence since it is ensured to only be used for reading data.
- Functions that contain transactional assignment or other transactional operations outside of a transaction code block, must be declared transactional.
- Stores to a persistent address must be explicitly declared as transactional or non-transactional to ensure the programmer made a decision about undo creation.

- Every transaction is guaranteed to either be committed or aborted. The use of code block annotation ensures a transaction cannot be started then abandoned.
- Cache flushing is automatically handled by the precompiler if and only if it is needed. The programmer cannot forget to add a flush.
- Locks acquired by a transaction are automatically released so that they cannot be left locked.
- Undo generation ensures the address is a pointer to NVM and in the same NVM region as the transaction.
- Undo cannot be generated for transient NVM pointers since they may be reinitialized before recovery begins.

Implementation

The NVM precompiler needs to parse the output of the standard C preprocessor. It is an enhanced version of the open source static code analyzer clang. It replaces the new syntax with calls to the NVM library and/or standard C syntax. If it discovers errors they will be reported like errors from the compiler. Errors reported by the NVM precompiler will terminate compilation before the real compiler is called.

NVM pointers

A prefix "^" operator in a declaration indicates a pointer to NVM rather than a pointer to volatile memory. The precompiler type information for a pointer needs to include a flag for every level of indirection indicating if that level is an address in NVM. This info is used to verify that the address stored in the pointer has the correct type of persistence. Indirection through the pointer must also use a prefix "^" for calculating an address in NVM or "=>" for fetching a member of a persistent struct. This must match the pointer expression persistence. Every address expression not only has the type of the addressed object, but also has the persistence of the object. This allows the precompiler to recognize every store into NVM and give an error if it does not use a persistent store operator.

A prefix "^" is replaced with "*" in the precompiler output so that the compiler will generate the standard code for a load or store. Similarly "=>" is replaced by "->" in the precompiler output.

A prefix "%" is used to take that address of an lvalue expression for a NVM location. The result is an address that can be stored in an NVM pointer declared with a prefix "^". It is an error to use a prefix "&" on an NVM lvalue. The precompiler replaces the "%" with "&" in its output file.

NVM struct

The new keyword "persistent" followed by "struct" begins the declaration of a persistent struct. The "struct" is followed by zero or more attribute descriptions. The attribute descriptions and other data from the declaration are used to add a JSON record to the USID definition file. After the attributes, there must be a struct name before the opening "{". The struct name must be application unique since it is used to construct global names.

The USID map utility will use the JSON record to construct a global const instance of nvm_type named __nvm_type_<name> where <name> is the struct name. Occurrences of the attribute functions are replaced by fields in the nvm_type global.

- tagof(<name>) becomes (__nvm_type_<name>.tag)
- versionof(<name>) becomes (__nvm_type_<name>.version)
- usidof(<name>) becomes (__nvm_type_<name>.usid)
- shapeof(<name>) becomes (&__nvm_type_<name>)

If a size attribute is included then the precompiler calculates the size of the struct and compares it with the size attribute value. If the size is greater than the attribute, then an error is reported. If the size is less than the size attribute, then an array of ub1 to make the struct the requested size is added under the member name "_padding_to_<size>" where <size> is the attribute value in decimal.

The word "persistent" and the attribute descriptions are removed from the output file.

If a non-persistent struct is embedded in a persistent struct, then a JSON persistent struct definition is also output for the non-persistent struct. The definition is the same as if the struct was declared persistent with no attributes. This results in it having an __nvm_type constructed for it so that the persistent struct nvm_type can refer to the type of the embedded struct.

Self Relative Pointers

A pointer in NVM that points to another location in NVM must be self relative since NVM may be attached at different virtual addresses at different times. A persistent struct field that is a pointer to NVM (i.e. uses "^" rather than "*" in the declaration) is a self relative pointer. The precompiler changes its declaration to be a nvm_srp which is just a typedef to int64_t. Any expression that contains a nvm_srp is modified by the precompiler to convert the int64_t into a virtual address by adding its own address to its contents. However a null pointer is represented by the value 1 so that a self relative pointer can point to itself. Any code that stores into a nvm_srp is modified to subtract the address of the location from the address being stored in it.

Persistent Symbols

Any global symbol such as a function or global struct instance can be assigned a USID to allow reference from an NVM region. A JSON record is added to the USID definition file containing the symbol name and its USID. The USID map utility uses the record to construct a global const nvm_usid named nvm_usid_<symbol> where <symbol> is the global symbol name.

Any occurrence in an expression of "|<symbol>" is replaced by "nvm_usid_<symbol>" thus becoming a copy of the USID.

A struct member or any variable, can be declared to be a USID pointer to volatile memory rather than a virtual address pointer by replacing the "*" with "|". The precompiler replaces the variable type with nvm_usid and removes the "|" from the declaration. The precompiler remembers the type and validates that any stores into the variable are for an expression that is a USID derived from a compatible type.

The indirect operator "*" will convert the USID into a virtual address via a call to nvm_usid_volatile inserted by the precompiler. The return value is cast to a pointer to the original type of the USID pointer.

Persistent Assignment

For any lvalue expression the precompiler can determine if it represents NVM or volatile memory. This is based on pointers to NVM being declared with "^" and all global symbols being for volatile memory.

There are three types of assignment recognized by the precompiler:

1. Volatile memory assignment: standard C assignment operators.
2. Transactional NVM assignment: standard C assignment operators prefixed with "@"
3. Non-transactional NVM assignment: standard C assignment operators prefixed with "~"

The precompiler reports an error if the persistence of the lvalue does not match the type of assignment. The "~" or "@" in a persistent store is removed by the precompiler.

Transactional assignment requires insertion of a call to nvm_undo before the assignment and a call to nvm_flush after the assignment. Non-transactional assignment only requires the nvm_flush. The precompiler turns the assignment statement into a "do{ …}while(0)" statement that includes the extra library calls. To avoid unexpected side effects from multiple calculations of the lvalue address, the code block includes a local variable that holds the address of the NVM. The variable is passed to the NVM routines. Note that this means persistent assignment cannot be used as an expression with a side effect of storing to memory. This is a precompiler limitation that would not exist if the compiler natively recognized the syntax.

The persistent increment/decrement operators are supported by moving the increment or decrement into a separate statement either before or after the statement containing the increment/decrement. Then it can be treated as a statement that adds/subtracts one from the value.

USID map utility

The precompiler outputs a USID definition file named "<name>.usd" where <name> is the name from the .c input file. The USID map utility accepts multiple USID definition files and outputs one combined USID definition file that eliminates duplicate entries. The input and output USID definition files have exactly the same format. The files describe the USID definitions found in the original C source file with NVM extensions. Since the USID's tend to be defined in .h files, there are lots of duplications.

An error is reported if there are different descriptions for the same USID. This is most likely the result of mixing an old version of a struct definition with a new upgraded version without changing the USID. The probability of two randomly chosen USID's having the same value is vanishingly small.

The utility can optionally produce a C source file that contains a function that will register all the USID to symbol mappings. The C source will also contain an initialized nvm_type for every persistent struct defined in the USID definition files, an initialized nvm_union for every persistent union, and an initialized nvm_extern for every persistent function or other global. One C source file should be created for each library and each application that uses the NVM Direct API. The registration function must be called before the process attaches to any NVM region file. The function will register the USID to symbol mappings for all the symbols in the USID definition file. These registrations are needed for finding callbacks that may need to be called by recovery at region attach.

Command Line Options

`usidmap [-r func ] [-c cfile] [-d dfile] ifile ...`

-r *func* The C output file will define a global function named *func*. It takes no arguments and returns an int that is 0 if there were any duplicate mappings from USID to a symbol. The return value can safely be ignored. This function must be called at startup of every application process before any other threads are spawned. It is common to call it in an initialization routine before main is entered.

-c *cfile* The USID C source file to initialize the USID mappings is written to *cfile*. No C source file is produced if this option is missing. Generally one C source file is produced for each library and each executable.

-d *dfile* The utility writes the merged USID definition file to *dfile*. If this is not specified then the USID definition file is written to standard out.

ifile There are one or more *ifile* parameters specifying input USID definition files to merge. These are generally the output from the NVM precompiler for the sources comprising an application. Any USID extern, NVM union or NVM struct mentioned in an *ifile* will be in the output USID definition file. The file extension is usually ".usd".

NVM library

The NVM library provides a number of functions and struct declarations for use by NVM applications. Some are generally used by precompiler inserted code, and others are called directly by application code.

Struct types

There are a number of struct types that are part of the NVM library interface.

nvm_usid

```
struct nvm_usid
{
    uint64_t d1;
    uint64_t d2;
};
typedef struct nvm_usid nvm_usid;
```

This defines a USID for a global symbol or persistent struct type. There is an inline function to efficiently compare two USID ‘s for being equal in value. USID’s are created by software developers and added to the source code when defining a type or global symbol. A truly random number must be chosen to be the USID. The same USID definition will generate different values when stored in servers with different endianess or generated by compilers with different bit field allocation algorithms. USID registration will reject USID’s that are the same with different endianess.

nvm_type

```
struct nvm_type {
    const nvm_usid usid;   // USID to identify an instance (0 if none)
    const char *name;      // The struct name (0 if none)
    const char *tag;       // The tag attribute string (0 if none)
    const size_t size;     // struct size in bytes
    const size_t xsize;    // extensible array entry size
    const uint32_t version;// software version number (0 if none)
    const uint32_t align;   // required alignment (a power of 2)
    int (*const upgrade@)();// upgrade function pointer (0 if none)
    const nvm_type *const uptype;// type definition of upgrade result
    const size_t field_cnt;  // number of fields not counting null
    const nvm_field field[0];// null terminated field descriptions
};
typedef struct nvm_type nvm_type;
```

There is one instance of nvm_type for every persistent struct type defined by an executable. It is a constant in the executable like a string constant. It contains all the information about the struct from the definition. This includes the following:

- The USID as a nvm_usid (Zero if no USID is defined)
- The struct name
- The tag string
- The version number
- Pointer to the upgrade function or null if there is none.
- Pointer to the nvm_type that the upgrade function will upgrade to

- The size of the struct including USID
- If the struct ends in a zero length array, then the length of each array entry for variable length allocation.
- Required memory alignment

A hash table is constructed at run time that maps each USID to the corresponding nvm_type. The table is updated every time a new shared object is loaded into the process. Every instance of nvm_type is given a global name of the form __nvm_type_XXX where XXX is the name of the persistent struct.

nvm_union

```
struct nvm_extern
{
    const nvm_usid usid; // a USID defined for some global symbol
    const void *addr;    // the value of the symbol from the linker
    const nvm_type *arg; // type data for arg to callback function
    const char *name;    // the symbol name
};
typedef struct nvm_extern nvm_extern;
```

There is one instance of nvm_union for every persistent union type. It is followed by a null terminated array of pointers to the nvm_type descriptions of the various persistent structs that are in the union. Note that an nvm_union does not have a USID. This is because a union must be embedded in a persistent struct, and cannot be directly allocated from a heap.

nvm_extern

```
struct nvm_extern
{
    const nvm_usid usid; // a USID defined for some global symbol
    const void *addr;    // the value of the symbol from the linker
    const nvm_type *arg; // type data for arg to callback function
    const char *name;    // the symbol name
};
typedef struct nvm_extern nvm_extern;
```

The USID map utility outputs a C file which includes a static initialized array of nvm_extern entries. There is one entry for every persistent symbol declared with a USID. There could be more than one such array in a process. Each shared library will have its own array if it contains any persistent symbols.

If the symbol is for a persistent callback function, then the arg field will point to an nvm_type describing the persistent struct that is passed as an argument to the function. Otherwise it is null. A persistent callback is any function that takes a pointer to a persistent struct as its only argument.

An application could create additional nvm_extern arrays at run time. However all USID's must be registered before attaching to an NVM region that contains such a USID.

nvm_field

```
struct nvm_field
{
    const uint64_t ftype:7;  // field type
    const uint64_t tflag:1;  // transient flag
    const uint64_t count:56; // array size
    const union {
        const uint64_t bits;            // size of the element in bits
        const nvm_usid_purpose purp;    // purpose of a USID field
        const nvm_field * const ptr;    // pointer indirect definition
        const nvm_type * const structptr;//persistent struct pointer
        const nvm_type * const pstruct; // embedded struct definition
        const nvm_union * const punion; // embedded union definition
    } desc; // description of the field
    const char *name;        // field name
};
typedef struct nvm_field nvm_field;
```

An nvm_field provides the type information for one field of a persistent struct. A null terminated array of these follows the nvm_type definition.

The desc type depends on the ftype field value. It is one of the following.

- 0 – This nvm_field is the null terminator at the end of a persistent struct description.
- 1 – Unsigned integer value. The desc member is bits. This gives the size of the unsigned integer in bits.
- 2 – Signed integer value. The desc member is bits. This gives the size of the signed integer in bits.
- 3 – Floating point value. The desc member is bits. This gives the size of the field in bits. It will be either 32 or 64.
- 4 – USID. The desc member is purp and it contains the purpose of the USID. This is its own ftype so it can be checked for being in the USID map. The purpose is one of the following
    - self: This contains the USID representing this struct. This normally is the first field in a struct.
    - struct: This is the USID for some other struct definition. This is uncommon.
    - func: This is the USID for a function.
    - data: This is a USID for a global data location.

- 5 – Pointer to other. The desc member is ptr. It is a pointer to another nvm_field that defines the type pointed to. With multiple levels of indirection this nvm_field would be another pointer ftype. A pointer to void has a null pointer in ptr.
- 6 – Pointer to struct. The desc member is pstruct. It is a pointer to an nvm_type that defines the persistent struct type pointed to.
- 7 – Embedded struct. The desc member is pstruct. It points to the nvm_type that defines the embedded struct.
- 8 – Embedded union. The desc member is punion. It points to a null terminated array of nvm_type pointers representing a persistent union. Each nvm_type pointer in the array points to a persistent struct definition that could be stored in the union. Note that a persistent union member must be a persistent struct that has a USID defined. This allows determining which member is currently stored in the union.
- 9 – Padding. The desc member is bits. This gives the size of the padding in bits. If padding begins at a byte boundary, then the size is 8 and the count is the number of bytes of padding.

The tflag field is one if the struct member was declared transient.

The count field defines the number of elements in the member. This is usually 1 to indicate the field is not an array (or an array of length 1). If it is an array then it will be the number of elements in the array. Note that an array might have zero elements in it.

nvm_desc

```
typedef uint32_t nvm_desc;
```

This type is used to hold a region descriptor. A region is identified by a small integer in a manner similar to a Posix open file descriptor. Zero is not a valid region descriptor. An application can choose the descriptor for each region it attaches or it can let the NVM library choose an unused region descriptor. An application that attaches a small number of regions for different purposes could have define constants for each region descriptor.

nvm_region_stat

```
struct nvm_region_stat
{
    nvm_desc desc;      // Descriptor of the region that was queried
    const char ^name;   // The region name given at creation
    void ^base;         // virtual address of base extent
    size_t vsize;       // virtual address space reserved for region
    size_t psize;       // total size in bytes for extents in region
    uint32_t extent_cnt;// Total extents including the base extent
    uint64_t attach_cnt;// Number of attaches since creation
    nvm_heap ^rootheap; // The root heap
```

```
    void ^rootobject;   // The root application object
};
```

This struct describes the current status of a region that is mapped into a process. It is used for returning status to the application via nvm_query_region.

nvm_extent_stat

```
struct nvm_extent_stat
{
    nvm_desc region; // Descriptor of the region containing the extent
    void ^extent;    // Virtual address of extent
    size_t psize;    // Physical size of extent
    nvm_heap ^heap;  // Pointer to heap or null if none
};
```

This struct describes the status of one extent of an NVM region. An ordered array of these can be returned by nvm_query_extents.

nvm_heap

```
typedef persistent struct nvm_heap nvm_heap;
```

An nvm_heap is used for allocating persistent structs. The root heap is created when the NVM region is created. An extent can be added to a region formatted as another heap.

Multiple heaps are useful for limiting resource consumption and for grouping multiple allocations for fast simultaneous deallocation by deleting the extent.

nvm_heap_stat

```
struct nvm_heap_stat
{
    const char ^name;   // Administrative name assigned to the heap
    nvm_desc region;    // The region that contains the heap
    void ^extent;       // The extent that holds the heap
    void ^rootobject;   // The optional heap local root object pointer
    size_t psize;       // The current extent physical size
    size_t free;        // Total free bytes in the extent
    size_t consumed;    // Space consumed by existing allocations
    int inuse;      // -1 if deleting and transaction slot if creating
};
```

This struct describes the current state of a heap. It is used for returning status to the application. All the heaps can be found through querying the extents.

nvm_mutex

```
persistent struct nvm_mutex { uint64_t m; }
typedef struct nvm_mutex nvm_mutex;
```

An nvm_mutex is used to protect data allocated in NVM. It must be initialized via a call to nvm_mutex_init after it is allocated in NVM. An nvm_mutex is only used as a field in another struct. It has a size of 8 bytes but is otherwise opaque to the application. An NVM mutex can be locked exclusive or shared via a call to nvm_lock. The lock is associated with a transaction and is not released until the transaction commits or aborts.

If an NVM struct containing a nvm_mutex is deallocated, the mutex is checked for being locked. If it is locked then the deallocation is not allowed.

nvm_mutex_array

```
typedef persistent struct nvm_mutex_array nvm_mutex_array;
```

This is an array of NVM mutexes that can cover a large number of persistent structs. The address of a struct is hashed to pick one mutex from the array. The size of the array is specified when it is created. The larger the array the fewer false collisions there will be. Elements of the array can also be found by index into the array.

A mutex array may be allocated out of any heap. The type is opaque to the application. Thus it cannot be embedded in a persistent struct, but it can be pointed to.

nvm_txconfig

```
struct nvm_txconfig
{
    uint32_t txn_slots;   // Max number of simultaneous transactions
    uint32_t undo_blocks; // 4K undo blocks available in region
    uint32_t undo_limit;  // Max undo blocks in a single transaction
};
typedef struct nvm_txconfig nvm_txconfig;
```

Every region has a fixed number of transaction slots that can be used and a fixed number of undo blocks to be shared by all running transactions. To catch a runaway transaction, there is a limit to the number of undo blocks consumed by a single transaction. An assert fires if this limit is exceeded. This struct describes the configuration for a region. It is also used for changing the configuration.

nvm_jmp_buf

```
struct nvm_jmp_buf
{
    int depth;   // Transaction nesting depth at nvm_setjmp
    jmp_buf buf; // Standard jmp_buf for setjmp/longjmp
};
```

Long jumps for NVM applications need to abort transactions if they are going to jump out of a transaction code block. This version of a jump buffer includes the transaction depth to recognize when there are transactions that must be aborted before peeling back the stack. Note that this is a volatile memory struct not an NVM struct.

NVM_SRP(type)

```
typedef int64_t type##_srp;
void type##_set(type##_srp *srp, type *ptr)
void type##_txset(type##_srp *srp, type *ptr)
void type##_ntset(type##_srp *srp, type *ptr)
type *type##_get(type##_srp *srp)
```

The macro NVM_SRP defines a self-relative pointer type for a given data type. It also defines three functions for storing an absolute pointer into the self-relative pointer type and one function for getting the absolute pointer from a self-relative pointer. This mechanism is not needed if using C extensions for NVM.

An NVM region is not necessarily always mapped in at the same address. Thus pointers within a region are self-relative. This type represents a self-relative pointer. The offset is added to the address of the pointer to get a real pointer. An offset of one is used to represent a null pointer.

Applications should never need this since the precompiler will automatically convert a persistent struct member that is a pointer to persistent memory into a self-relative pointer. The precompiler will insert the code to convert between virtual addresses and self-relative pointers.

Entry points

This section describes the library functions that are provided for use by an NVM application. Some functions detect runtime errors storing an appropriate error code in errno and returning a value that indicates failure. Errors that represent coding bugs in the application will often be handled via an assert mechanism. The assert mechanism may differ with different implementations, but generally it results in the termination of the process. There are some checks for corrupt NVM data structures. If this happens the corruption is reported and handled much like an assert.

All of the library functions are thread safe. Locks are acquired as needed to coordinate access to the data structures used by the library. All locks acquired by the library are released before the function returns. This includes locks on NVM mutexes. Any NVM mutex used by the library has a higher lock level than the application can use. Thus locks acquired by the library cannot cause any lock ordering errors.

Some of the functions require the caller to have an active NVM transaction. The effect of these functions will be undone if the caller's transaction rolls back to a savepoint set before the function call, or if the caller's transaction aborts. When the function returns, the caller's transaction is in the same state it was when the function was called except that it may have some more undo. The function may create a nested transaction, but it will be ended before the function returns.

nvm_thread_init

```
void nvm_thread_init(void);
```

This initializes a thread to use the NVM library. It must be called before any calls other than USID registration.

nvm_thread_fini

```
void nvm_thread_fini(void);
```

When a thread is gracefully exiting it needs to call nvm_thread_fini to clean up the data the NVM library is maintaining for it. After this call returns the thread may not make any other calls to the NVM library except nvm_thread_init. If the thread has a current transaction an assert will fire.

This call is only necessary if the application needs to continue execution without this thread.

nvm_create_region

```
nvm_desc nvm_create_region(
    nvm_desc region, // Zero or a specific region descriptor
    char *pathname,  // Path name of the region file to create
    char *regionname,// Persistent name for the region
    void *attach,    // Virtual address to attach it at
    size_t vspace,   // Virtual address space to consume
    size_t pspace,   // Physical memory for the base extent
    mode_t mode      // Permissions for creation system call
);
```

This function creates a region file in an NVM file system, maps it into the creating process's address space, and initializes the transaction and heap mechanisms for application use. A region descriptor is returned. If the region argument is zero then the library will choose an unused descriptor. If a non-zero region descriptor is passed in then it will be used. The application must ensure it does not attempt to use the same region descriptor twice. The caller will need to call nvm_set_root_object() to complete initialization making the region valid for reattaching.

The region descriptor is needed for beginning transactions in the region and for some of the other library functions.

The virtual address for attach must be non-zero, page aligned, and not in use. It is an error if this address cannot be used in mmap.

All application calls to nvm_usid_register must be done before this is called.

This cannot be called in a transaction since it would have to be for another region.

If there are any errors then errno is set and the return value is zero.

nvm_set_root_object

```
int nvm_set_root_object(
    nvm_desc region,  // Region descriptor from nvm_create_region
    void ^rootobj     // Pointer to application's root object
);
```

All the application data in an NVM region must be reachable through NVM pointers starting from the root object. If there are any application NVM variables that need to be static or global they will be fields in the root object.

Immediately after creating a region there is no root object so the region is still marked as invalid. To make the new region valid this function must be called to save the root object. The root object is typically allocated from the root heap immediately after the region is successfully created. The allocation and initialization of the new root object must be committed before this is called. There may not be any active transactions in the region when this is called.

If there are any errors then errno is set and the return value is zero.

nvm_new_root_object

```
void ^nvm_new_root_object@(
    void ^rootobj     // Pointer to a new root object
);
```

There may be circumstances where a new object needs to become the root object for the region. This must be done in a transaction to ensure the region remains valid. The transaction must create the new root object and pass it to nvm_new_root_object() before committing. The transaction will usually delete the old root object either before or after nvm_new_root_object() is called, but in the same transaction. The return value is the old root object.

This operates on the region of the current transaction.

If there are any errors then errno is set and the return value is zero.

nvm_destroy_region

```
int nvm_destroy_region(
    char *name    // Name of the region file to destroy
);
```

This will delete a region returning its NVM to the file system. The region must not be attached to any process. The region may also be deleted via the file system.

This cannot be called from inside an NVM transaction, since there is no means of undoing this operation. It cannot be called in an on commit operation since it could get an error that makes the transaction unrecoverable.

If there are any errors then errno is set and the return value is zero.

nvm_attach_region

```
nvm_desc nvm_attach_region(
    nvm_desc desc, // 0 or specific region descriptor
    char *name,    // Name of the region file to attach
```

```
    void *attach   // Virtual address to attach it at
);
```

Attach a Non-Volatile Memory region to this application and return a region descriptor. If the region argument is zero then the library will choose an unused descriptor. If a non-zero region descriptor is passed in then it will be used. The application must ensure it does not attempt to use the same region descriptor twice. The returned region descriptor is valid for all threads in the application.

If the last detach was not clean, recovery will be run to roll back any active transactions and complete any committing transactions. Recovery completes before this returns.

All application calls to nvm_usid_register must be done before this is called.

This cannot be called in a transaction since it would have to be for another region.

There are a number of things that can result in this returning with an error:

- The file name must be for a file in an NVM file system.
- The user must have read/write permission for the file.
- There must be adequate virtual address space available at the address requested. The virtual address space size required is in the file.
- The virtual address must be properly aligned.
- The executable must contain definitions for every persistent function and struct in the region.
- The calling thread must not have a current transaction in another region.
- No other processes can have the region attached
- The region must be properly formed with a root object.

If there are any errors then errno is set and the return value is zero.

The region descriptor is used for beginning transactions in the region and for a parameter to other library functions that operate on a region.

nvm_query_region

```
int nvm_query_region(
    nvm_desc region,      // Region to query or zero if transactional
    nvm_region_stat *stat // Stat buffer for return data
);
```

This returns the status of a region that is mapped into the current process. Note that the returned status might not be current if some other thread is changing the region.

If called within a transaction then the region to query may be zero to use the region owning the current transaction.

If there are any errors then errno is set and the return value is zero.

nvm_detach_region

```
int nvm_detach_region(
    nvm_desc region // Region to detach
);
```

This function cleanly detaches this NVM region from this process. The contents of the region are not changed. It is an error to detach while there are transactions active or committing in the application. If this is a multi-process application then the other processes must call nvm_region_change to unmap the region from their address space. It is not always necessary to detach a region. Application exit will detach.

It is not an error to detach a region that was already detached.

This cannot be called in a transaction since it would have to be for another region.

If there are any errors then errno is set and the return value is zero.

nvm_add_extent

```
void ^nvm_add_extent@(
    void ^extent,       // Virtual address of new extent
    size_t psize        // Physical size of extent in bytes
);
```

This function adds a new application managed extent of NVM to a region that is currently attached. This must be called within a transaction that saves a pointer to the extent in some application persistent struct. If the transaction rolls back the extent add will also rollback. On success the return value is the extent address as passed in.

The address range for the new extent must not overlap any existing extent. It must be within the NVM virtual address space reserved for the region. It must be page aligned and the size must be a multiple of a page size.

This operates on the region of the current transaction.

If there are any errors then errno is set and the return value is zero.

nvm_resize_extent

```
int nvm_resize_extent@(
    void ^extent,      // Address of an existing extent
    size_t psize       // New physical size of extent in bytes
);
```

This function changes the size of an existing application managed extent. This must be called within a transaction. If the transaction rolls back the resize will also rollback.

The address range for the extent must not overlap any existing extent. It must be within the NVM virtual address space reserved for the region. The new size must be a multiple of a page size and not zero. This must not be a heap managed extent.

This operates on the region of the current transaction.

If there are any errors then errno is set and the return value is zero.

nvm_remove_extent

```
int nvm_remove_extent@(
    void ^addr     // Address within an the extent to delete
);
```

This removes an extent from an NVM region returning its space to the NVM file system. It must be called within a transaction. The extent is deleted when the transaction commits. The extent must be application managed. Call nvm_delete_heap to remove a heap managed extent. The contents of the extent are ignored so this will work even if the extent is corrupt. The application is responsible for removing any pointers to the extent as part of the transaction. The base extent cannot be removed.

This operates on the region of the current transaction.

If there are any errors then errno is set and the return value is zero.

nvm_query_extents

```
int nvm_query_extents(
    nvm_desc region,       // Region to query
    int extent,            // Index of first extent to report
    int count,             // Number of extents to report
    nvm_extent_stat stat[] // Array of count statuses to fill in
);
```

This returns the status of one or more extents from a region. Extents are ordered by their address within the region. The base extent is number zero. To query all extents in a region, call nvm_query_region to find the number of extents. Call nvm_query_extents with extent == 0 and count == ext_count from the nvm_region_stat. To query extent 42 pass extent==42 and count==1. Note that adding or deleting an extent can change the index to other extents since the index is based on the address order of the extent relative to all existing extents, and extents can be added or deleted between other extents.

If called within a transaction then the region to query may be zero to use the region owning the current transaction.

If there are any errors then errno is set and the return value is zero. It is not an error to query extents that do not exist. The data returned is simply zero.

nvm_create_heap

```
nvm_heap ^nvm_create_heap(
    nvm_desc region, // Region to create heap in
    void ^extent,    // Address of new extent
    size_t psize,    // Physical size of extent in bytes
    const char *name // Administrative name for the heap
);
```

This function adds a new heap managed extent of NVM to a region that is currently attached. It formats the extent as an NVM heap. The return value is a pointer to the new heap. The new heap has all the space from the new extent. Note that the heap pointer might not be the same address as the extent address passed in. The heap pointer can be retrieved later via nvm_query_extents if it is not saved in NVM by the application.

This must be called in a transaction for the region containing the heap. If the transaction aborts or rolls back through this call, then the heap and its extent will be deleted. The transaction that creates this heap may allocate and free from the heap even before the heap creation itself commits. However other transactions may not use the heap until the creating transaction successfully commits.

The address range for the new extent must not overlap any existing extent. It must be within the NVM virtual address space reserved for the region. It must be page aligned and the size must be a multiple of a page size.

If there are any errors then errno is set and the return value is zero.

nvm_resize_heap

```
int nvm_resize_heap(
    nvm_heap ^heap, // Heap pointer returned from nvm_create_heap
    size_t psize    // New physical size of heap extent in bytes
);
```

This function changes the size of an existing heap managed extent. If the new size is larger, then the heap in the extent will have more memory available for allocation. If the new size is smaller, then there must not be any allocations in the area to be removed. The root heap may be resized.

This may be called in a transaction for the same region or outside a transaction. In either case a successful heap resize is persistently committed, even if called in a transaction that later aborts. Other threads could allocate the new space even before nvm_ resize _heap returns. Thus the library cannot generate undo to rollback the resize if the caller's transaction aborts.

An error will be returned if the extent cannot be resized. The address range for the extent must not overlap any existing extent. It must be within the NVM virtual address space reserved for the region. The new size must be a multiple of a page size and not zero.

If there are any errors then errno is set and the return value is zero.

nvm_delete_heap

```
int nvm_delete_heap(
    nvm_heap ^heap  // Heap to delete
);
```

Delete a heap and remove its extent returning its space to the file system. The contents of the extent are ignored, except for the nvm_heap itself, and the extent is simply removed from the region. Thus a corrupt heap can be deleted without encountering any corruption. The caller must ensure all pointers to the heap or to any objects in the heap are removed before committing.

This must be called in a transaction for the region containing the heap. If the transaction aborts or rolls back through this call, then the heap and its extent will be not be deleted. However any attempt to allocated, or free from the heap will fail until the transaction commits removing the heap extent. If the transaction rolls back through this deletion, then the heap will return to normal operation.

If there are any errors then errno is set and the return value is zero.

nvm_heap_setroot

```
void nvm_heap_setroot@(
    nvm_heap ^heap, // heap to store root object pointer into
    void ^root      // new root object pointer
);
```

Every heap extent can have a root object. This routine will store a pointer to a root object that will be saved in the heap metadata. It is returned as part of the heap status from nvm_heap_query. The root object pointer is null when a heap is first created.

This must be called in a transaction so that it will be undone if the calling transaction aborts. The caller is responsible for ensuring only one thread at a time sets a root object pointer.

An assert will fire if the root object is not in the heap where the pointer is stored.

Having a root object is useful for creating a heap that is not pointed to by any other heap. The heap and its contents can be found through nvm_query_extents, then the root object in the heap can be found to find all other heap contents. If there is a corruption in the heap, then the heap can be deleted removing it from the region without having to update any other data structures.

nvm_query_heap

```
int nvm_query_heap(
    nvm_heap ^heap,     // Pointer to heap to query
    nvm_heap_stat *stat // Stat buffer for return data
);
```

This returns the status of a heap. Note that the data returned might not be current if some other thread has used the heap.

By querying all extents all heaps can be found and queried.

If there are any errors then errno is set and the return value is zero.

nvm_alloc

```
void ^nvm_alloc@(
    nvm_heap ^heap,      // Heap to allocate from
    const nvm_type *def, // Type definition of struct to allocate
    int count            // Size of the array
);
```

This allocates one or more instances of the persistent struct described by the nvm_type, from the nvm_heap passed in. The struct type must have a USID defined. Usually the nvm_type is acquired through the shapeof() built-in function. Allocation must be called in a transaction that stores the returned pointer in an NVM location that is reachable from the root struct. However the precompiler cannot enforce this requirement.

The count is the number of instances of a struct to allocate in an array. This must be at least one. A persistent struct is extensible if its last field is a zero length array. If the persistent struct defined by the nmv_type is extensible, then the count describes the actual size of the empty array at the end of the struct. This allocates one instance of the persistent struct defined by *def with a non-zero array size for the last field in the struct. The allocation will be for (def->size+(def->xsize*count)) bytes. If the persistent struct defined by *def is not extensible, then the count argument is the array size of structs defined by *def. The allocation will be for (def->size*count) bytes.

The bytes consumed, including overhead, are added to the consumed amount for the heap. It is an error if there is insufficient space. There must be adequate space in the extent free list as well. Fragmentation can result in no free block in the heap being large enough for the allocation even though there is sufficient space.

The space returned is initialized. Numeric values and transient fields are set to zero. All USID fields at the beginning of a struct are set to the correct values. This includes the USID for the struct being allocated as well as any embedded structs. An embedded union is initialized as its first member. Self-relative pointers are set to the null pointer value which is 1.

If the transaction rolls back through this allocation, the allocated space will be released back to the free list. Thus it is not necessary to create undo for any stores to the new space until the transaction commits.

Note that an nvm_union cannot be passed to nvm_alloc to allocate a union. A persistent union must be embedded in a persistent struct to be allocated from an NVM heap.

If there are any errors then errno is set and the return value is zero.

NVM_ALLOC

```
NVM_ALLOC(heap, type, count)
```

This is a macro that calls nvm_alloc based on the struct type. It casts the return value to the correct type and finds the nvm_type address based on the struct type.

nvm_free

```
int nvm_free@(
    void ^ptr     // Pointer to allocation returned by nvm_alloc
);
```

This frees memory that was allocated by nvm_alloc. This must be called in a transaction. The space is not available for other allocations until the transaction successfully commits. Releasing the space and clearing the memory is done as an on commit operation in the current transaction. The caller is obligated to clear or deallocate any pointers to the allocated space before committing the transaction.

If there are any errors then errno is set and the return value is zero.

nvm_verify

```
void nvm_verify(
    void ^ptr,           // A pointer to verify
    const nvm_type *def, // Type definition of expected struct
);
```

This will verify that the NVM pointer points to the indicated type of persistent struct. The USID pointed to is compared with the USID in the type definition. If they are equal the routine returns. If they do not match then the USID in NVM is looked up to find its nvm_type. If it is not found, or is for a type that cannot be upgraded to the expected type, then a corruption is reported. The upgrade functions are called to upgrade the current object to the desired one. Each upgrade function is called in its own transaction, which may be nested if there is a current transaction. It is presumed that the application will only use pointer types that are compatible with the current compatibility setting. After a successful call to the upgrade function the new USID is transactionally stored and the transaction committed.

This function is not normally called directly by an application. The precompiler adds calls to this function before the first use of an NVM pointer within a function. The goal is to catch stale pointers before they can corrupt an NVM region.

If there are any errors then corruption is reported and the process exits.

nvm_mutex_init

```
void nvm_mutex_init@(
    nvm_mutex ^mutex, // The NVM mutex to initialize
    uint8_t level     // Lock ordering level
);
```

This initializes an NVM mutex so it can be locked. This must be called in the transaction that allocates the NVM for the mutex. No undo is created for the initialization since rollback is presumed to release the mutex back to free memory. The mutex must not be locked until the transactions commits.

The level is used for ensuring there are no deadlocks in the code. A transaction can only do a wait get of a lock on a mutex if all the locks it currently holds are at a lower lock level. A lock level of zero can be given, but that means that all locking must be no-wait. Lock levels 200 and above are reserved for the NVM library itself.

An assert fires if there are any errors.

nvm_mutex_fini

```
void nvm_mutex_fini@(
        nvm_mutex *mutex // The NVM mutex to finalize
 );
```

Finalize use of a mutex before freeing the NVM memory holding it. This must be called in the transaction that is freeing the memory. This releases any volatile memory structures allocated for it. When the transaction commits the delete, the mutex is zeroed making it invalid.

An assert is fired if the mutex is held or has any waiters.

nvm_lock

```
int nvm_lock@(
    nvm_mutex ^mutex, // The NVM mutex to lock
    int excl,         // If true exclusive lock, else share
    int timeout         // Maximum time to sleep in microseconds
);
```

This acquires an NVM lock and hangs it on the current transaction. The lock may be a shared or exclusive lock. If the lock is successfully acquired the return value is 1. If the lock cannot be granted immediately, then the thread will sleep until the lock is granted or timeout microseconds have passed. A negative timeout sleeps forever. If the timeout expires then the return value is zero and errno is EBUSY.

If the mutex is already locked exclusive by the current transaction, or any of its parent transactions, then this returns success immediately. This is not a locking order violation.

The lock will be released when the current transaction commits, or if all undo generated while the lock was held is applied by rollback or abort. Releasing the lock may wake up one or more waiters. Note that there is no means to explicitly release the lock without applying the undo or committing the transaction. This is important because the undo represents stores to the data protected by the lock. Thus the lock must be held until the undo is applied or discarded. The lock must be persistent since recovery after a crash may acquire locks in on abort or on commit operations.

An assert fires if the locking order for the mutex is violated and timeout is negative. This would indicate that the code has a potential deadlock. The mutex must be in the same region as the transaction.

nvm_xlock

```
int nvm_xlock@(
    nvm_mutex ^mutex // The NVM mutex to lock
);
```

This is a simplified version of nvm_lock for the most common case of waiting for an exclusive lock.

nvm_slock

```
int nvm_slock@(
    nvm_mutex ^mutex // The NVM mutex to lock
);
```

This is a simplified version of nvm_lock for the case of waiting for a shared lock.

nvm_create_mutex_array

```
nvm_mutex_array ^nvm_create_mutex_array@(
    nvm_heap ^heap, // Heap to allocate array from
    uint32_t count, // Number of mutexes in the array
    uint8_t level   // Lock level for initializing all mutexes
 );
```

Create and initialize an array of mutexes. This must be called in a transaction for the region containing the heap. If the transaction rolls back then the mutex array will be released. A pointer to the array is returned. A null pointer is returned if there is insufficient memory to allocate the array. Other errors will fire an assert.

nvm_pick_mutex

```
nvm_mutex ^nvm_pick_mutex(
    nvm_mutex_array *array,
    void ^ptr
 ;)
```

Pick one mutex from a mutex array based on hashing an NVM address in the same region. A pointer to the chosen mutex is returned. This must be called in a transaction for the region containing the array. Any errors will fire an assert.

nvm_get_mutex

```
nvm_mutex ^nvm_get_mutex@(
    nvm_mutex_array ^array,
    uint32_t index
 );
```

Get one mutex from a mutex array by an index into the array. A pointer to the mutex is returned. If the index is past the end of the array a null pointer is returned. This must be called in a transaction for the region containing the array. Any errors will fire an assert. This is useful for acquiring all the mutexes one at a time.

nvm_destroy_mutex_array

```
 void nvm_destroy_mutex_array@(
     nvm_mutex_array ^array        // mutex array to release
);
```

Destroy a mutex array returning its space to the heap it was allocated from. This must be called in a transaction for the containing region. The array will be usable if the transaction rolls back through this destroy. An assert will fire if any of the mutexes are held or have waiters.

nvm_flush

```
void nvm_flush(
    void ^ptr,    // address of NVM to flush from caches
    size_t bytes  // size of area to flush
);
```

This ensures that all cache lines in the indicated area of NVM will be scheduled to be made persistent at the next nvm_persist. This is a no-op if the platform does not need an explicit flush per cache line to force cached data to NVM.

This is not normally called directly from the application. The precompiler adds a flush with every store to an NVM address. An application only needs to do an explicit flush if it calls a legacy function to update NVM. The legacy function will not do any flushes since it is not written to use NVM. The application must do the flushes when the legacy function returns.

nvm_persist

void nvm_persist();

This is a barrier that ensures all previous stores to NVM by the calling thread are actually persistent when it returns. On some systems it is necessary to flush each cache line that was modified before executing the barrier. If the system uses Flush On Shutdown (FOS) to force NVM data to be persistent, then this barrier is a no-op. If the system uses Commit On Shutdown (COS) this forces the dirty NVM to the memory controller write queue.

This is not normally called directly from the application. The creation of each undo record executes a barrier as well as commit or abort of a transaction. An application may need to do this explicitly if it is storing data in NVM without using transactions. For example, an application might copy client data to an NVM buffer then update a counter to make it visible. A persist barrier is needed after the copy and before updating the counter. Another persist barrier is needed after the counter update to make the counter persistent. Note that in this case a flush is also required if bcopy or memcpy is used to copy the data. Presumably the counter update uses a non-transactional store which automatically includes a flush if needed.

nvm_persist1

```
nvm_persist1(
    void ^ptr    // address of NVM to flush from caches
);
```

This is a barrier for a single cache line. It includes both a flush and a persist. This is slightly faster for situations where a single store needs to be made persistent. Note that at least half of the persist barriers are for a single store. The basic mechanism for atomic update of NVM is to update some number of locations that are not visible to other threads, force them to be persistent, and then persist a single store to make them visible.

nvm_txbegin

```
void nvm_txbegin(
    nvm_desc region, // Region transaction modifies
);
```

This begins a new transaction. If there is no current transaction then a new base transaction is started in the indicated NVM region. If there is a current transaction then a nested transaction is begun. For beginning a nested transaction the region descriptor can be zero to begin it in the same region as the current transaction. If it is not zero, and not the same as the current transaction, then the nested transaction is an off region nested transaction in that region.

This is not normally called directly from the application. Normally "@" is used to begin a transactional code block. The precompiler inserts the appropriate nvm_txbegin call at the beginning of the code block.

If there are already too many allocated transactions, this call may have to block until some other transaction commits or aborts. Note that a nested transaction in the same region as the current transaction uses the same transaction slot as the base transaction so it never sleeps waiting for a slot. An off region nested transaction has to allocate a transaction slot in the new region.

An assert is fired if there is an error such as passing a region descriptor of zero when starting a base transaction.

nvm_txend

```
void nvm_txend@();
```

This ends the current transaction. If this is a nested transaction then the parent becomes the current transaction. If this is a base transaction without a parent, then the transaction slot is made available for a new transaction, and there is no current transaction any more.

This is not normally called directly from the application. Normally the precompiler inserts the nvm_txend call at the end of the transaction code block or before any transfer of control out of the transaction code block.

If the current transaction is not committed or aborted then it is committed.

nvm_txdepth

```
int nvm_txdepth();
```

This returns the current transaction nesting depth. If there is no current transaction then the depth is zero. If the current transaction is the base transaction then the depth is one. If the current transaction is a nested transaction then the depth is two or more depending on how deep it is nested. There is no specific limit on nesting, but there is a limit on the amount of undo retained for a base transaction plus all of its nested children.

This may be called even if nvm_thread_init has not been called by this thread, as long as it has been called by some thread in the process.

nvm_txstatus

```
int nvm_txstatus(
    int parent    // Number of parents back to get status for
);
```

This returns the status of the current transaction or one of its parents. If the parent value is zero then the current transaction is being queried. If the parent value is 1 then the immediate parent is being queried. If greater than 1 then an earlier parent is being queried. Negative values are an error.

Note that the callback function for an onabort or oncommit operation is called within its own nested transaction. Thus the status of the current transaction will always be active when the function is entered. Pass a count of 1 to find out the status of the transaction that the operation is part of.

The return value is one of the following:

- NVM_TX_NONE: There is no such transaction. The value of parent is greater than or equal to the current transaction depth.
- NVM_TX_ACTIVE: The transaction is active. Its fate is not yet resolved.
- NVM_TX_ROLLBACK: The transaction is still active but is currently being rolled back to a savepoint. Once the rollback completes the transaction will return to status active.
- NVM_TX_ABORTING: The transaction is in the middle of aborting. Undo application is not yet complete. The transaction will eventually become aborted when all undo is applied.
- NVM_TX_ABORTED: The transaction was successfully aborted, but execution is still within the transaction code block. In other words nvm_txend() has not yet been called.
- NVM_TX_COMMITTING: The transaction is in the middle of committing. It may be releasing locks or calling on commit operations as part of committing. The transaction will eventually become committed when all on commit operations are complete.

- NVM_TX_COMMITTED: The transaction was successfully committed, but execution is still within the transaction code block. In other words nvm_txend() has not yet been called.

nvm_txdesc

```
nvm_desc nvm_txdesc@();
```

This returns the region descriptor for the current transaction. It is useful for on abort or on commit callbacks to know which region they are in.

nvm_txslot

```
uint16_t nvm_txslot@();
```

This returns the transaction slot for the current transaction. It is useful for setting a flag that limits access to a single transaction.

nvm_undo

```
void nvm_undo@(
    void ^data,    // The address of the data about to be modified
    unsigned bytes // The number of bytes of data to save in undo
);
```

This saves the current contents of a contiguous piece of NVM in one or more undo records of the current transaction. If the transaction aborts or rolls back to a previously created savepoint, then the contents will be restored to the values saved here. Rollback may also be the result of process death. Recovery by another process will apply the undo.

Usually undo is automatically created via a transactional store to an NVM address. The precompiler will insert calls to nvm_undo before storing in NVM. An application might need an explicit call for creating undo before passing an NVM address to a legacy function that will store results in NVM. Explicitly generating undo for an entire struct before multiple non-transactional stores into it, may be more efficient than multiple transactional stores that each create an undo record.

It is possible that there are not enough undo blocks available to hold the new undo. In that case the thread will sleep until more undo space is available.

There is a region specific limit on the total amount of undo that a single base transaction and all its nested transactions can consume. This is necessary to prevent a runaway loop from consuming all available undo locking up the system. If this limit is exceeded then an assert is fired.

nvm_onabort

```
void ^nvm_onabort@(
    void (|func@)(), // USID of a callback function for applying
);
```

This creates an undo record in the current transaction. The function must be a persistent callback, meaning it has a pointer to a persistent struct as its only argument. The record contains an instance of the persistent struct. A pointer to the argument struct is returned so that the caller can configure the arguments. The struct will be initialized as if was allocated from an NVM heap.

If the undo record is applied, the indicated function is called with a pointer to the persistent struct allocated in the undo record. The function is called inside its own nested transaction. The nested transaction is committed when the function returns. The nested transaction must not be aborted because this would make the NVM region unrecoverable. Note that during recovery the function may be called in another process.

This allows arbitrary application code to be called during rollback or transaction abort. The on abort operation can implement logical undo which acquires its own locks rather than holding locks until the parent transaction commits. A nested transaction can be started once the undo is created. The nested transaction can acquire locks update NVM and transactionally store in the undo record the information needed to undo the update. The nested transaction commits releasing any locks it holds. This reduces lock contention, and can be used to avoid deadlocks.

The callback function must not call longjmp to return to a setjmp that was set before it was called. The jmpbuf will not exist if this is called during recovery.

If a new software release changes the callback function to take an argument with a new field, then the old function should be kept with the same USID and a new USID defined for the new version of the function. The old version of the function may be registered under a new function name as long as it has the same USID. The old callback is only needed if attaching to a region that was last in use by the previous software version. However there could potentially be region files that have not been attached, and thus recovered, in years. Thus the old function needs to be kept for multiple software releases.

If there are any errors then errno is set and the return value is zero.

nvm_oncommit

```
void ^nvm_oncommit@(
    void (|func@)(), // USID of a callback function for commit
);
```

This creates an undo record in the current transaction. The function must be a persistent callback, meaning it has a pointer to a persistent struct as its only argument. The record contains an instance of the persistent struct. A pointer to the argument struct is returned so that the caller can configure the arguments. The struct will be initialized as if was allocated from an NVM heap.

If the current transaction commits, the indicated function is called with a pointer to the persistent struct allocated in the undo record. The function is called after all locks have been released by the containing transaction. It is called inside its own nested transaction. The nested transaction is committed when the

function returns. The nested transaction must not be aborted because this would make the NVM region unrecoverable. Note that during recovery the function may be called in another process.

This allows arbitrary application code to be called as part of transaction commit. Once the transaction is put in state committing and all locks are released, then all the on commit operations are executed. Thus the functions are only called if the transaction is guaranteed not to abort or do any rollback. This is useful for avoiding deadlocks and reducing lock contention. If the process executing the function dies, then recovery will rollback the nested transaction and re-execute the oncommit function

The callback function must not call longjmp to return to a setjmp that was set before it was called. The jmpbuf will not exist if this is called during recovery.

If a new software release changes the callback function to take an argument with a new field, then the old function should be kept with the same USID and a new USID defined for the new version of the function. The old version of the function may be registered under a new function name as long as it has the same USID. The old callback is only needed if attaching to a region that was last in use by the previous software version. However there could potentially be region files that have not been attached, and thus recovered, in years. Thus the old function needs to be kept for multiple software releases.

If there are any errors then errno is set and the return value is zero.

nvm_onunlock

```
void ^nvm_onunlock@(
    void (|func@)(), // USID of a callback function for unlock
);
```

This creates an undo record in the current transaction. The function must be a persistent callback, meaning it has a pointer to a persistent struct as its only argument. The record contains an instance of the persistent struct. A pointer to the argument struct is returned so that the caller can configure the arguments. The struct will be initialized as if was allocated from an NVM heap.

When the current transaction either commits or aborts, all the locks acquired by the transaction are released in the reverse order they were acquired. Similarly, rollback to a savepoint releases locks acquired since the savepoint was established. For abort or rollback, all undo records are applied in reverse order so that the locks are released after all undo generated while the lock is held is applied.

The undo record created by this call is treated like a lock. The indicated function is called with a pointer to the persistent struct allocated in the undo record. The function is called at commit, abort, or rollback that applies the record. When the callback is called, locks that were held when nvm_onunlock() was called are still held. It is called inside its own nested transaction. The callback function can call nvm_txstatus(1) to determine if the parent transaction, containing the unlock record, committed, aborted, or is being rolled back to a savepoint.

The nested transaction is committed when the function returns. The callback must not call nvm_abort because this would make the NVM region unrecoverable. Note that during recovery the function may be

called in another process. If the thread executing the function dies, then recovery will rollback the nested transaction and re-execute the onunlock function.

This allows arbitrary application code to be called while locks previously acquired are still held, even if the transaction is committing. This is useful for executing code that only runs if the transaction commits, and runs before relevant locks are released. This mechanism can also be used to have application implemented locks that are different than nvm_mutex locks.

The callback function must not call longjmp to return to a setjmp that was set before it was called. The jmpbuf will not exist if this is called during recovery.

If a new software release changes the callback function to take an argument with a new field, then the old function should be kept with the same USID and a new USID defined for the new version of the function. The old version of the function may be registered under a new function name as long as it has the same USID. The old callback is only needed if attaching to a region that was last in use by the previous software version. However there could potentially be region files that have not been attached, and thus recovered, in years. Thus the old function needs to be kept for multiple software releases.

If there are any errors then errno is set and the return value is zero.

nvm_commit

```
void nvm_commit@();
```

This commits the current transaction. The current transaction is committed releasing all its locks and executing any on commit or on unlock operations. The transaction remains the current transaction until its code block is exited. No more transactional operations are allowed in this transaction.

This is not normally needed by an application since exiting the transaction code block automatically commits the transaction and nvm_txend calls nvm_commit if the transaction is not committed or aborted.

It is an error if there is no current transaction or it is already committed or aborted. If there are any errors then an assert is fired.

nvm_abort

```
void nvm_abort@();
```

This aborts the current transaction applying all undo and releasing locks as the undo is applied. The transaction remains the current transaction until its code block is exited. No more transactional operations are allowed in this transaction.

This is not normally needed by an application since longjmp outside the transaction code block automatically aborts the transaction.

It is an error if there is no current transaction or it is already committed or aborted. If there are any errors then an assert is fired.

nvm_savepoint

```
void ^^nvm_savepoint@(
    void ^name;   // Name of the savepoint (changeable)
);
```

This creates a savepoint in the current transaction. The savepoint is an undo record that defines a named point in the undo of a transaction. The name for the save point is an NVM address of some location in the NVM region the transaction modifies. The return value is a pointer to the NVM location where the name is stored. This allows a null pointer to be passed when creating the savepoint and then setting the real name later when it has been chosen. For example the name could be the address of an object that is allocated after setting the savepoint. Then rollback to savepoint will delete the object.

Setting a savepoint allows subsequent changes to be rolled back without aborting the entire transaction. Unlike a nested transaction, the changes under a savepoint do not commit until the current transaction commits and locks acquired after the savepoint are held until the current transaction commits.

There may be multiple savepoints in the same transaction. They may even have the same name. Rollback to savepoint stops at the first matching savepoint.

If there are any errors then errno is set and the return value is zero.

nvm_rollback

```
int nvm_rollback@(
    void ^name   // Name of target savepoint
);
```

This rolls back the current transaction until the first savepoint with the given name is encountered. This is the most recently created savepoint with the given name since undo is applied in reverse of the order it is generated. It is not an error to have multiple savepoints with the same name. An error is returned if there is no savepoint with the given name in the current transaction. A nested transaction cannot rollback to a savepoint in its parent transaction.

If there are any errors then errno is set and the return value is zero. No rollback is done if there is an error.

nvm_get_txconfig

```
int nvm_get_txconfig(
    nvm_desc region,   // region to read nvm_tx_config from
    nvm_txconfig *cfg  // buffer to copy nvm_tx_config into
);
```

This returns the current transaction configuration for a region. It may be stale if another thread is setting a new configuration. Config is all zero and zero is returned if invalid descriptor is passed.

nvm_set_txconfig

```
int nvm_set_txconfig(
    nvm_desc region,      // region to change nvm_tx_config of
    nvm_txconfig *cfg     // new nvm_tx_config for the region
);
```

This changes the transaction configuration for a region. The new values may be greater than or less than the old values. If there are fewer transaction slots than application threads then there is the possibility that a thread that wants to begin a transaction may have to wait until a transaction commits or aborts. If the number of transaction slots times the limit of undo blocks per transaction is greater than the total number of undo blocks, then there is the possibility that a transaction may have to wait when generating undo until another transaction releases an undo block. However this may be unlikely.

Transaction slots and undo blocks are allocated in megabyte chunks of 63 slots and 254 undo blocks (this could change). Thus requesting 64 slots will actually allocate 126 slots. However the slot limit will be honored and the extra slots left unused. Similarly, requesting 255 undo blocks will actually allocate 508 undo blocks. Since extra undo blocks cannot hurt, all available undo blocks will be usable and reported by nvm_get_txconfig.

The configuration is changed in a transaction that commits before returning. If this is called inside a transaction then a nested transaction is created and committed. The configuration is changed atomically.

If there are any errors then errno is set and the return value is zero. The new configuration must be reasonable.

nvm_CAS

```
uint16_t nvm_cas2(
    uint16_t ^ptr,   // address of NVM location to modify
    uint16_t oldval, // expected value to change
    uint16_t newval  // new value to store if compare is successful
);

uint32_t nvm_cas4(
    uint32_t ^ptr,   // address of NVM location to modify
    uint32_t oldval, // expected value to change
    uint32_t newval  // new value to store if compare is successful
);

uint64_t nvm_cas8(
    uint64_t ^ptr,   // address of NVM location to modify
    uint64_t oldval, // expected value to change
    uint64_t newval  // new value to store if compare is successful
);
```

These functions do a 2 byte, 4 byte or 8 byte compare and swap of a location in NVM. The return value is the value of the location before the CAS executed. If it is equal to the old value then the swap succeeded.

A CAS is like a nested transaction that either commits or aborts on its own. It cannot be part of a larger transaction that updates multiple locations atomically, and might rollback.

The NVM version of CAS is different than a normal CAS because it first forces any previous NVM stores to be persistent, and if the swap happens, it forces the new value to be persistent before returning.

Note that before this returns another thread could see the new value before it is persistent. NVM can become corrupt if the other thread does a store to NVM based on seeing the value set by this thread, unless the other thread ensures the value set by this CAS is persistent. One way of doing that is to CAS it to yet another value by calling this CAS function again. Another means is to call nvm_persist1 to make the value persistent before reading it.

nvm_usid_qualify

```
int nvm_usid_qualify(
        nvm_usid usid // USID to check
);
```

This qualifies a 128 bit random number for use as a USID. A 128 bit random number does not qualify as a USID if it looks too much like application data or if it has the same value when endianess is changed. For debugging and checking corruption it is sometimes useful to scan NVM looking for a specific USID. Eliminating numbers that are more likely to just be application data reduces false positives for these scans. If a region file is copied from one platform to another it could have the wrong endianess. A USID mismatch is used to recognize when a persistent struct is from an incompatible platform.

There are three reasons that a 128 bit random number would be rejected as a USID.

1. At least one byte must have a sign bit set to eliminate ASCII strings looking like a USID.
2. There must not be any uint16_t values that are 0x0000 or 0xFFFF. Many application data values are in fields that are larger than the actual value and are thus sign extended with 0 or 1 bits.
3. The data4 field must not be a palindrome that is the same for both big and little endian platforms.

nvm_usid_register

```
int nvm_usid_register_types(const nvm_type *const types[]);
int nvm_usid_register_externs(const nvm_extern *const syms[]);
```

These two routines register USID mappings for persistent structs and other global symbols. The argument is a null terminated array of pointers to const structs describing each mapping. Registering persistent struct descriptions takes pointers to nvm_type structs. The address of the struct is associated

with the USID. Registering global symbols takes pointers to nvm_extern structs that contain an address and USID to register.

The usidmap utility will output a C file with these calls in a routine named by the –r argument to usidmap. Thus the application does not normally need to call these, but it does need to call the routine created by usidmap before any region files are attached.

Typically every library and independently linked object will have its own calls to register the symbols and persistent structs that it defines. The application must arrange to have the registrations called before any calls to attach or create a region file. Registration for the NVM Direct library happens on the first call to nvm_thread_init.

There must not be any duplicate USID's that map to different addresses in the same process. Every USID must be a true random number that passes the tests in nvm_qualify. An assert fires if any of these tests fail.

nvm_usid_find

```
const nvm_extern *nvm_usid_find(
        nvm_usid usid // USID to lookup in the usid map
);
```

This finds an nvm_extern entry in the USID map. If there is not an entry for the USID then a NULL pointer is returned, and errno is set to ENOENT. If the USID was used in the definition of a persistent struct, then the addr field in the nvm_extern points to the nvm_type describing the persistent struct. If the USID was associated with a function, then addr is the entry point to the function. If used with some other global symbol definition, then addr is the value of that symbol.

nvm_usid_volatile

```
void *nvm_usid_volatile(
    nvm_usid usid // USID to map to a volatile memory address
);
```

This converts a USID into an address in volatile memory. The USID must have been declared as a persistent struct type or with some global symbol. If the USID is for a persistent struct type then a pointer to the nvm_type instance for the struct is returned. If it is for some other global symbol, then the value of the symbol is returned. The lookup is in a hash map built at runtime so that it is relatively fast.

This is not normally called directly by an application. Usually a USID pointer is declared using "|" rather than "*", and dereferencing the USID pointer calls nvm_usid_volatile to convert the USID into a volatile memory address.

The hash map is built by nvm_usid_register so the USID to address mapping must be passed to it. A null pointer is returned if the USID is not found in the USID hash map.

nvm_setjmp/nvm_longjmp

```
int nvm_setjmp(
    nvm_jmp_buf *buf // jmp_buf equivalent
);

void nvm_longjmp(
    nvm_jmp_buf *buf, // jmp_buf equivalent
    int val           // nvm_setjmp return value
);
```

These functions are equivalent to the standard setjmp/longjmp functions except that they abort transactions if necessary. If a transactional code block is exited by a long jump to a setjmp outside of the block, then the transaction has to be aborted, unless it is already committing. It would be a serious bug to use a normal longjmp to jump outside of a transaction code block.

nvm_init_struct

```
size_t nvm_init_struct(
    void ^addr,          // Persistent memory location to initialize
    const nvm_type *def, // Type definition of struct to initialize as
    int count            // Size of the array
);
```

This initializes one or more instances of the persistent struct described by the nvm_type and located at addr. If the struct is extensible (last member is a zero length array), then count is the actual size of the array. If the struct is not extensible, then count is the number of instances of the struct in the array starting at addr. This is the same as nvm_alloc.

The memory is cleared just as if it had been allocated from an NVM heap. All numeric fields and transient fields are set to zero. All self-relative pointers are set to the null pointer value which is 1. If the struct or any embedded structs have a USID defined for them, then the correct USID is stored. If there is a union then it is initialized as if it was the first persistent struct in the union.

The return value is the number of bytes that were initialized.

nvm_copy

```
void ^nvm_copy(
    void ^dest,      // destination in NVM
    const void *src, // source in NVM or volatile memory
    size_t n         // number of bytes to copy
);
```

This copies data into NVM from either NVM or volatile memory. It works just like the standard C function memcpy except that it flushes all the data to NVM. It returns the destination address. It does not generate any undo for the destination.

This should be more efficient at flushing than a loop that does assignment.

WARNING! Copying NVM data that contains self-relative pointers will corrupt the pointers. Use the C extensions for persistent struct assignment to correctly copy the pointers. Standard C struct assignment will corrupt the pointers. The pointers can be fixed after copying by copying them one at a time using the inline functions created by NVM_SRP.

nvm_set

```
void ^nvm_set(
    void ^dest,  // destination in NVM
    int c,       // value to store in every byte
    size_t n     // number of bytes to set
);
```

This initializes a range of bytes in NVM and ensures they are persistently stored in NVM. It works just like the standard C function memset except that it flushes all the data to NVM. It returns the beginning address of the byte range.

This should be more efficient at flushing than a loop that does assignment.

Services

The NVM library requires a number of services from the application in order to accomplish its functions. Having the application provide these services allows the library to be used in a number of different environments.

The services required by the library are defined in nvms.h, which is included in every NVM library source file. The application may want to provide a modified version of nvms.h which defines some of the services as inline functions or macros. Naturally this requires access to the NVM library sources so that they can be recompiled with the new nvms.h. It is expected that each platform will provide a services implementation so that most applications will not need to create their own services.

NVM Region File Services

These services are for managing region files that contain the NVM managed by the library. A region file can be sparse with multiple extents of NVM. On Linux this will be implemented by OS calls to an NVM file system.

These services must provide access control to ensure only authorized users may open or attach an NVM region file. When NVM is allocated to a region file the service must ensure it is zeroed so that any previous content cannot be seen by any application.

Region File Handle

```
typedef struct nvms_file nvms_file;
```

An nvms_file holds the data about an open region file. It is allocated in process local memory by the service library. A pointer to an nvms_file is used as an opaque handle to an open file.

Create Region File

```
nvms_file *nvms_create_region(
    char *pathname,// path name of the region file to create
    size_t vspace, // file size for consuming virtual address space
    size_t pspace, // physical memory for the base extent
    mode_t mode    // permissions for creation system call
);
```

This creates a region file and returns a handle to it. Initially it contains pspace bytes of NVM. An error is returned if there is insufficient NVM to allocate pspace bytes. The file size is set to vspace so that the file consumes vspace bytes of virtual address space when mapped. The file contents are zero.

It is an error if the region file already exists.

If there are any errors then errno is set and the return value is zero.

Destroy Region File

```
int nvms_destroy_region(
    const char * pathname // name of the region file to destroy
);
```

This will delete a region returning its NVM to the file system. The region must not be attached to any process. The region may also be deleted via the file system.

If there are any errors then errno is set and the return value is zero.

Open Region File

```
nvms_file *nvms_open_region(
    const char *pathname // name of the region file to open
);
```

This will return a handle to an existing region file. It is an error if the file is not in an NVM file system.

If there are any errors then errno is set and the return value is zero.

Close Region File

```
int nvms_close_region(
    nvms_file *handle // handle from open or create region
);
```

This will close a region file opened by nvms_open_region or nvms_create_region. Note that this should only be done if the file is not mapped into the current process. After this returns the handle is no longer usable.

If there are any errors then errno is set and the return value is zero.

Write Region File Header

```
size_t nvms_write_region(
    nvms_file *handle, // handle from open or create region
    void *buffer,      // buffer to write from
    size_t bytes       // number of bytes to write
);
```

This writes the beginning of a region file from a volatile memory buffer and returns the number of bytes written. It is an error to write beyond the end of the base extent.

If there are any errors then errno is set and the return value is zero.

Read Region File Header

```
size_t nvms_read_region(
    nvms_file *handle, // handle from open or create region
    void *buffer,      // buffer to read into
    size_t bytes       // number of bytes to read
);
```

This reads the beginning of a region file into a volatile memory buffer and returns the number of bytes read. This could be less than the amount requested if the file is shorter.

If there are any errors then errno is set and the return value is zero. Note that reading a zero length file will return zero bytes read and errno will still be zero.

Lock Region File

```
int nvms_lock_region(
    nvms_file *handle, // handle from open or create region
    int exclusive // if true lock exclusive else lock shared
);
```

Acquire an advisory lock on the region file. If the process already owns a lock on the region then convert the lock mode of the existing lock. A lock conflict is returned as an EACCES error rather than blocking for the conflict to be resolved. The lock is owned by the calling process on behalf of all threads in the process.

If there are any errors then errno is set and the return value is zero.

Map Region File

```
int nvms_map_region(
    nvms_file *handle,  // handle from open or create region
    void *attach,  // virtual address to map it at
    size_t vspace, // virtual address space to consume
    size_t pspace  // physical space of base extent
);
```

This read/write maps an open region file into the current process and returns the virtual address where it is mapped. The attach parameter is the virtual address to attach at. An error is returned if the file cannot be mapped at this address.

The vspace parameter is the same value passed to nvms_create_region. This much process virtual address space must be reserved for potential extension of the file. However, only the first pspace bytes should actually appear in the address space. Any access to the virtual addresses between pspace and vspace must result in a memory fault – not NVM allocation.

The other extents of the region file will be mapped by calls to nvms_map_extent.

If there are any errors then errno is set and the return value is zero.

Map Region Extent

```
int nvms_map_extent(
    nvms_file *handle,  // handle from open or create region
    off_t offset,  // the extent offset in the region file
    size_t len     // the extent size in bytes
);
```

This maps one extent of NVM into the process address space. The extent was previously created via nvms_add_extent and possibly resized to len by nvms_size_extent. If len is different than a previous mapping at the same offset, then this is a remap after an extent resize by another process. Any old addresses that were previously mapped as part of this extent must now result in a memory fault if accessed. The virtual address of the extent must be the return value from nvms_map_region plus offset. Note that a len of zero means the extent was deleted.

It would be an indication of corruption if there was no NVM allocated to the specified portion of the region file. It would be an error if the extent address range was not within the vspace parameter passed to nvms_map_region, or if offset+len overlaps the offset of another mapped extent.

If there are any errors then errno is set and the return value is zero.

Add Region Extent

```
int nvms_add_extent(
    nvms_file *handle,  // handle from open or create region
    off_t offset,  // the extent offset in the region file
    size_t len     // the extent size in bytes
);
```

This allocates len bytes NVM in a region file at offset. The data is initialized to zero. All processes will have to call nvm_map_extent to include it in their address space.

If there are any errors then errno is set and the return value is zero.

Size Region Extent

```
int nvms_size_extent(
    nvms_file *handle,  // handle from open or create region
    off_t offset,  // the extent offset in the region file
    size_t len     // the new extent size in bytes
);
```

This changes the size of an existing extent either allocating more NVM or freeing NVM. If the new len is zero then the extent is deleted. The extent mapping in the calling process is changed to reflect the new size as if nvms_map_extent was called. If this is a multi-process application the other processes will have to call nvm_map_extent to fix their address space.

If there are any errors then errno is set and the return value is zero.

Unmap Region

```
int nvms_unmap_region(
    void ^base,
    size_t bytes
);
```

This unmaps any region files in the given address range. It is not an error if there is nothing mapped there. If the range overlaps with only part of a region, the entire region will be unmapped.

If there are any errors then errno is set and the return value is zero.

Unique Id

```
uint64_t nvms_unique_id();
```

This returns a 64 bit number to use as a unique id for ensuring a region file is not mapped in multiple times. It needs to be reasonably unique within the server. For example it could be the current process id in the upper 32 bits and microseconds since boot in the lower 32 bits.

Memory Management

Memory management services provide 2 classes of volatile memory based on the visibility to different threads of execution:

1. Thread local memory is only accessed by a single thread of execution. It may be accessible to other threads, but the NVM library does not do that. If the owning thread exits cleanly the memory must be automatically released by the services layer. For Linux this could be allocated via malloc.

2. Application global memory can be accessed by any thread in any process via the same virtual address. For Linux this could be allocated via malloc.

For each class of memory, there is an allocate and a free service. This could be malloc/free for both classes.

All volatile memory used by the NVM library is dynamically allocated from one of the two classes. The library does not declare any global or static variables other than those that are const. In order to maintain its state, it needs the services library to maintain root pointers. The NVM library allocates a root struct and stores a pointer to it in a root pointer. There are two kinds of root pointers based on how they are shared amongst threads:

1. Application root pointer: There is only one application root pointer in an application. Every thread sees the same value. It is in application global memory.
2. Thread root pointer: Each thread has its own private thread root pointer. It is in thread local memory so that it disappears if the thread calls nvm_fini.

Thread Initialization

```
void nvms_thread_init();
```

This is called once before any other calls to the service library. It provides the library a means of doing any one time initialization such as creating a pthread key.

Thread Finalization

```
void nvms_thread_fini();
```

This is called once after all other calls to the service library. It provides the library a means of doing any cleanup such as deleting a pthread key. This does not terminate the thread. It could be followed by another call to nvms_thread_init() in the same thread.

Thread Root Pointer

```
void nvms_thread_set_ptr(
    void *ptr // root pointer for the current thread
);

void *nvms_thread_get_ptr();
```

These functions set and get the root pointer for the current thread. The root pointer will be a value returned by nvms_thread_alloc.

Thread Allocate

```
void *nvms_thread_alloc(
    size_t size // amount of thread class memory to allocate
);
```

This allocates size bytes of zeroed memory that is visible to the calling thread. If the thread calls nvms_fini the memory will be automatically released as if it was passed to nvms_thread_free.

If there are any errors then errno is set and the return value is zero.

Thread Free

```
void nvms_thread_free(
    void *ptr // pointer previously returned by nvms_thread_alloc
);
```

This releases a chunk of thread class memory previously allocated to the calling thread.

Application Root Pointer Address

```
void **nvms_app_ptr_addr();
```

This returns the address of the application root pointer for the current application. The application root pointer itself must be zero when the application is started

This routine is used both to get and set the application root pointer.

Application Allocate

```
void *nvms_app_alloc(
    size_t size // amount of application class memory to allocate
);
```

This allocates size bytes of zeroed memory that is visible to the calling application.

If there are any errors then errno is set and the return value is zero.

Application Free

```
int nvms_app_free(
    void *ptr   // pointer previously returned by nvms_app_alloc
);
```

This releases a chunk of application class memory previously allocated to the calling application.

Volatile Memory Mutexes

The NVM library needs mutexes for coordinating access to volatile memory between threads. The services library needs to provide these mutexes. For Linux a pthread mutex can be used

The NVM library always releases any mutex it is has locked before it returns from a library call and it never does a longjmp. Thus it does not need a means of releasing mutexes before an nvm_longjmp. However the application may need that facility for locks that it acquires.

Mutex Handle

```
typedef void *nvms_mutex;
```

This defines an opaque type used as a handle to an application global mutex that can be used to coordinate between threads within the same application.

Mutex Creation

```
nvms_mutex nvms_create_mutex(void);
```

This allocates and initializes an application global mutex.

If there are any errors then errno is set and the return value is zero.

Mutex Destruction

```
void nvms_destroy_mutex(
    nvms_mutex mutex // mutex to destroy
);
```

This destroys an application global mutex, created via nvms_create_mutex, freeing the memory it consumed. The mutex must not be locked when destroyed.

Mutex Locking

```
int nvms_lock_mutex(
    nvms_mutex mutex, // mutex to lock
    int wait          // if the lock is not available wait for it
);
```

When this returns 1 the calling thread has an exclusive lock on the indicated mutex. If the wait parameter is true then the thread might sleep waiting for another thread to release its lock. If wait is false then 0 is returned rather than wait for the holder to release its lock.

Mutex Unlocking

```
void nvms_unlock_mutex(
    nvms_mutex mutex // mutex to unlock
);
```

When this returns the calling thread no longer has a lock on the indicated mutex. This might wakeup another thread waiting for a lock. The calling thread must have locked the mutex via nvms_lock_mutex.

Compare and Swap

```
uint16_t nvms_CAS2(
    uint16_t *ptr,   // address of location to modify
    uint16_t oldval, // expected value to change
    uint16_t newval  // new value to store if compare is successful
);

uint32_t nvms_CAS4(
    uint32_t *ptr,   // address of location to modify
    uint32_t oldval, // expected value to change
    uint32_t newval  // new value to store if compare is successful
);
```

```
uint64_t nvms_CAS8(
    uint64_t *ptr,   // address of location to modify
    uint64_t oldval, // expected value to change
    uint64_t newval  // new value to store if compare is successful
);
```

These functions do a 4 byte or 8 byte compare and swap of a location in NVM or volatile memory. The return value is the value of the location before the CAS executed. If it is equal to the old value then the swap succeeded.

Volatile Memory Condition Variables

The NVM library needs a mechanism similar to pthread condition variables for blocking a thread when it needs to sleep until another thread releases a resource. Condition variables must be allocated in application global memory so that they are available to all threads in an application. On Linux these can be pthread condition variables.

Condition Variable Handle

```
typedef void *nvms_cond;
```

This type represents an opaque handle to a condition variable provided by the services library. The handle must be usable by any thread in the application. Thus it is a pointer to application global memory.

Create Condition Variable

```
nvms_cond nvms_cond_create(void);
```

This function allocates and initializes one condition variable in application global memory and returns a handle to it.

Destroy Condition Variable

```
void nvms_cond_destroy(
    nvms_cond cond // the condition variable to destroy
);
```

This destroys one condition variable making its handle no longer usable and releasing any memory associated with it. There must not be any threads waiting on the condition variable when it is destroyed.

Wait on Condition Variable

```
void nvms_cond_wait(
    nvms_cond cond,   // condition variable to wait on
    nvms_mutex mutex, // mutex to release while sleeping
    uint64_t stoptime // 0 or abs time to stop wait (see nvms_utime)
```

The calling thread will block until the condition variable is posted to wake it up. The application global mutex passed in must be locked by the calling thread. It will be unlocked while the thread is blocked and locked when this returns. The same mutex must be held by all waiters and posters of the same condition variable. Passing a zero stop time blocks until posted or a signal is received. A non-zero stop time will stop waiting when that time arrives. The current time can be acquired through nvms_utime(). Using an absolute stop time allows multiple waits to be aggregated into one stop time.

Post a Condition Variable

```
void nvms_cond_post(
    nvms_cond cond, // condition variable to post
    int all         // true to post all threads rather than one
);
```

This function will wake up one thread waiting on the condition variable or all threads waiting on the condition variable. If there are no threads waiting then nothing happens. To avoid starvation, the thread waiting the longest should be posted first. The caller must hold the same mutex that the waiters held when calling nvms_cond_wait.

Processor Cache Control

Processor caches can hold a dirty version of an NVM cache line. To make the contents of NVM consistent it is necessary to flush the caches to NVM. With flush on shutdown nothing is flushed until the system is shutting down. However flush on barrier requires the software to have barriers that ensure previous stores to NVM are persistent. The implementation of barriers is different from one platform to another. Thus there needs to be service functions to implement barriers. The precompiler inserts these barrier calls if the persistence model requires them. This is controlled by command line options on the precompiler.

Parameters

```
void nvms_get_params(nvms_parameters *params);
```

Return a copy of the configuration parameters for this application.

This returns the size of a cache line for the current environment. Commonly it is 64. This is needed for knowing how many stores are covered by the same cache line flush.

Here is a partial list of the parameters to be provided

- Cache line size
- Size of the table of cache lines needing to be flushed
- Max regions to attach
- USID hash table size

- Number of wait buckets used for waiting on NVM mutexes
- Alignment needed for mapping NVM into a process

Flush a Cache Line

```
void nvms_flush(
    void ^ptr
);
```

This starts one cache line on its way from a processor cache to its NVM DIMM. It is possible that there is nothing to be done. The ptr argument might not be cache line aligned. The cache line may remain in cache after it is made persistent, but some implementations remove it from the processor cache.

Persistent Barrier

```
void nvm_persist();
```

This does not return until all outstanding stores to NVM by the current thread are truly persistent. Some platforms require nvms_flush to be called for each cache line before this will wait for it to be flushed.

Error Handling

There are some error situations that cannot be handled by returning failure and setting errno. These are reported to the service library which exits the process so that recovery can be run.

Assert Failure

```
void nvms_assert_fail(
    const char *err // error message
);
```

When the application makes a logical error that cannot be returned as an error nvms_assert_fail is called to end the process

A good example of this is generating too much undo.

NVM Corruption

```
void nvms_corruption(
    const char *err, // error message
    void ^ptr1,      // a relevant address to report
    void ^ptr2       // a relevant address to report
);
```

When an impossible data structure state is discovered this is called to report the problem. The error message describes the problem. Depending on the problem there may be one or two addresses that help identify where the corruption is. This exits the process rather than return.

Transaction Recovery

```
void nvms_spawn_txrecover(
    void *ctx // context to pass on to nvm_txrecover
);
```

This is called to spawn a thread to recover a transaction owned by a thread that died during the transaction. This is called at region attach to recover transactions that were running at the last detach.

The new thread must call nvm_txrecover passing the ctx to it.

If this service library only supports one single thread of execution in one process, then nvm_txrecover can simply be called from nvms_spawn. However this will only work if the region is only accessed by single threaded applications. If region attach discovers two dead transactions, the first one to recover might hang.

Time

```
uint64_t nvms_utime();
```

This returns the time in microseconds since the epoch. For a Posix system this is based on gettimeofday().

Glossary

This section lists terms used in this document that might not have obvious meanings.

NVM

NVM stands for Non-Volatile Memory. This is computer memory that looks to the software exactly like DRAM except that it retains its contents through power failure and reboot. Some NVM technologies are slower than DRAM.

NVM file system

This is a standard OS file system that stores its data in NMV on the local server. When a file is mapped into a process via mmap, the NVM holding the file is directly mapped into the process address space so that load and store instructions go directly to the NVM.

Region file

A region file is a file in an NVM file system that was created by the nvm_create_region call described in this document

Region

When a region file is mapped into a process it is referred to as a region. A region consumes a fixed amount of virtual address space. Usually the virtual address space is much larger than the amount of NVM allocated.

NVM application

An NVM application consists of one multi-threaded processes that has one or more NVM regions in its virtual address space. It uses the NVM API to directly load and store from the NVM.

Extent

An extent is a contiguous range of virtual addresses backed by NVM in the virtual address space of a regio. An extent is in the region file at an offset that is the same as its offset from the beginning of the region.

Base extent

Every region has a base extent that starts at offset zero. It contains metadata about the region as well as application defined data.

Heap managed extent

An extent can be formatted to be an NVM heap that is managed by the NVM library. The base extent is always a heap managed extent since it contains metadata created by the NVM library. There is only one set of free lists in a heap managed extent.

NVM heap

When a heap managed extent is added to a region it is formatted to contain one heap that owns all the space and allcations in the extent. It can be resized later. The API provides functions to allocate persistent structs from an NVM heap and release them later.

Root heap

The root heap is the heap of the base extent. It is the only heap created when a region is created. The NVM library allocates its metadata from the root heap.

Root object

Every application needs to create a root object which is used to find all the application data in an NVM region. A region is not properly formed until the application registers its root object. It is common for the root object to be allocated from the root heap.

Application managed extent

An extent can be added to a region and not formatted as an NVM heap. It is just raw bytes to be used by the application in any way it wishes. It can be updated transactionally to do atomic changes to it. NVM in the extent can be initialized as a persistent struct via nvm_init_struct.

Persistent struct

A persistent struct is a struct that is declared following the keyword “persistent”. It can be allocated in NVM directly through nvm_allocate, if it has a USID. If it does not have a USID then it may be embedded in another persistent struct.

Effectively persistent struct

An effectively persistent struct is a plain struct that is embedded in a persistent struct. There is an nvm_type created for its initialization. The shapeof() built-in function does not work on an effectively persistent struct, but it can be found through shapeof() the persistent struct it is embedded in.

Persistent function

A persistent function is a function that has a USID defined for it. This means the USID can be stored in NVM as a function pointer that is bound to the actual function address at runtime.

Persistent callback function

A persistent callback function is a persistent function that takes one argument which is a pointer to a persistent struct. A callback can be used for an on abort, on unlock, or on commit operation to create an application defined undo record.

Extensible struct

An extensible struct is a struct that ends in a zero length array of any type. The length of the array if fixed at allocation time by the amount of memory allocated for the zero length array.

NVM transaction

An NVM transaction atomically updates multiple NVM locations in the same region. This allows complex data structures to be managed in NVM. As NVM data structures are modified, undo is saved with the transaction so that the NVM contents can be reverted to the state they were in when the transaction began.

Transaction code block

A transaction is defined by declaring a C code block that begins a transaction when it is entered and ends the transaction when it exits.

Base transaction

A base transaction begins when a transaction code block is entered and the thread of execution does not have a transaction already running.

Nested transaction

A nested transaction begins when a transaction code block is entered and the thread of execution already has a transaction. A nested transaction commits or aborts independently of its parent transaction. A nested transaction can modify a different region than its parent.

Current transaction

When executing in a transaction code block or a subroutine called from a transaction code block the current transaction is the transaction begun by the closest transaction code block. This can be a nested or base transaction. Only the current transaction can be operated on. Note that the transaction is still the current transaction until its transaction code block is exited, even if it has committed or aborted.

Parent transaction

The parent transaction of a nested transaction is the transaction that was current when the nested transaction code block was entered. There can be many levels of transaction nesting. A base transaction does not have a parent transaction. The parent transaction becomes the current transaction when the nested transaction code block is exited, not when the nested transaction commits or aborts.

Savepoint

A savepoint marks a place in the undo generated by a transaction. This can be used to return the NVM to the state it was in when the savepoint was created.

Rollback

Rollback is the act of applying and discarding undo to return the NVM to a previous state. Rollback to savepoint applies undo until the designated savepoint is reached. Transaction abort rolls back to the beginning of the transaction applying all undo generated by the transaction.

NVM mutex

NVM mutexes are very much like pthread mutexes except that they reside in NVM rather than volatile memory and they are locked by transactions rather than threads of execution. They are released when the owning transaction commits, or if all undo is applied that was generated after the mutex was locked by the transaction.

NVM precompiler

The NVM precompiler is a utility that implements the C extensions for NVM. It takes the output of the standard C preprocessor and outputs a standard C source file with calls to the NVM library inserted as needed. It also outputs a USID definition file for the USID map utility to use as input.

NVM library

The NVM library is a set of C functions that help the application manage NVM. It provides functions for managing regions, heaps, locks, and transactions. Some of the functions are called explicitly by the

application source code. Calls to other functions are automatically inserted by the NVM precompiler based on the NVM extensions to C in the source file.

Service library

The service library provides a set of services needed by the NVM library. Different service libraries are needed to run in different environments. An operating system usually supplies a service library that is suitable for single process multi-threaded applications. Complex applications may supply their own service library. For example there is a custom service library for the Oracle RDBMS.

USID

It is a 128 bit true random number that can be stored in NVM as a reference to a global symbol defined by the executable of an NVM application. At run time it is bound to the symbol value by the application that attaches to the NVM region. One of the more common uses is to associate a USID with a persistent struct definition. When this is done the first field in the struct is always initialized to the USID, and the USID maps to a struct nvm_type in the executable which describes the layout of the persistent struct type.

A USID is similar to a GUID or UUID, but has some differences. It does not conform to RFC4122 because all bits are randomly chosen. However about one in 7,300 random numbers are rejected. This ensures a USID is different depending on the bit field ordering and endianess. However, most values are rejected to reduce the chance of a USID matching application data. See nvm_usid_qualify for a more detailed description of the restrictions.

USID map utility

The USID map utility (usidmap) takes one or more USID definition files and merges them, removing duplications and outputting a single USID definition file. The definition files can be from the NVM precompiler or from a previous run of the USID map utility.

The USID map utility can optionally generate a C source file to compile and link with the application. It provides data structures for interpreting USID's used by the application and code to register the mappings for use at runtime.

USID definition file

A USID definition file is an ascii file conforming to the JSON standard. It defines all the USID's used in a set of sources. It maps all the USID's to global symbols. It contains details about all the persistent structs and persistent unions used.

USID C source file

The C source file provides an initialized global array of type pointer to nvm_extern and another array of pointers to nvm_type. This pairs every USID used in the original C source file with a global symbol value generated by the linker. For every persistent struct used there is an initialized global nvm_type that

defines the format of the struct. For every persistent union used there is an initialized global nvm_union that lists the persistent structs that form the union. It also includes a function to register the USID's with the NVM library.

Assert fires

There are application errors that can occur as a result of improperly coding an application. When this is detected, the assert failed service routine is called. This is referred to as firing and assert.

Thread local memory

Thread local memory is only accessed by one thread. Each thread needs one global pointer to find all the thread local memory it allocates. Thread local memory does not need to be visible to any other threads, but it is common for threads in the same process to share the same address space.

Application global memory

Application global memory is shared by all threads in an application. Application global memory sits at the same virtual address for all threads. A pointer to application global memory can be used by any thread. The NVM library cannot define global symbols in application global memory, but an application might have such a facility.