Contents
1 Introduction ............................................................................................................... 4
2 Programmer’s Kit Overview .................................................................................. 4
3 Simple Test Setup and Requirements .................................................................. 4
4 High Level Interface API ...................................................................................... 5
  4.1 Initialize/UnInitialize Software Subsystem .................................................. 5
  4.2 Get Capability ................................................................................................. 6
  4.3 Get Samples ..................................................................................................... 6
  4.4 Reset Statistics ............................................................................................... 7
  4.5 Get Policy ........................................................................................................ 8
  4.6 Set Policy ......................................................................................................... 9
  4.7 Delete Policy .................................................................................................. 10
  4.8 Set Policy Status ............................................................................................ 11
  4.9 Register Alerts – Initiate call backs as needed ............................................. 12
5 High Level APIs Data Structure Definitions ....................................................... 14
  5.1 Capabilities Structure ..................................................................................... 14
  5.2 Policy Structure ............................................................................................... 14
  5.3 Sample Power / Thermal Reading Structure ................................................ 15
6 IPMI Interface API Definition .............................................................................. 16
  6.1 Connect To a Device ....................................................................................... 16
  6.2 Disconnect a Device ....................................................................................... 17
  6.3 Run Ipmi Command ....................................................................................... 17
  6.4 Initialize a System for Local Commands ...................................................... 18
  6.5 Get SEL Info .................................................................................................. 19
  6.6 Get SEL Record ............................................................................................. 19
  6.7 Delete SEL Record ......................................................................................... 20
  6.8 Clear the SEL ................................................................................................. 21
  6.9 Register Event – Initiate call backs as needed ............................................. 21
  6.10 Unregister an Event Handler .......................................................................... 22
  6.11 Get SDR Info ............................................................................................... 22
  6.12 Get SDR Record ........................................................................................... 23
  6.13 Delete SDR Record ....................................................................................... 24
  6.14 Clear the SDR ............................................................................................... 24
  6.15 Get FRU Info ............................................................................................... 25
1 Introduction

This Intel® Node Manager Programmer’s Reference Kit implementation document discusses, in detail, the various interfaces and associated APIs of various layers of the kit. This document may refer to Intel Node Manager Programmer’s Reference Kit as Programmer’s kit in various sections.

2 Programmer's Kit Overview

Intel Node Manager Programmer’s Reference Kit is a software package that contains all the required source code, test application, and documents to manage Node Manager capable Intel servers. Target audiences of the programmers’ reference kit are ISVs, IPDCs, other private and public cloud vendors, and OEMs/ODMs, who do want to build their own consoles. This kit will facilitate the faster adoption of the NM servers by allowing easy deployment of essential software eco-system ingredients to manage NM capable servers.

3 Simple Test Setup and Requirements

The NMPRK is very simple to use and requires no additional external libraries to compile or run. All that is needed is a C/C++ compiler, and to run the configuration and compilation scripts. The library includes a set of unit tests that can be run by the consumer to verify that all the parts of the library are working correctly. There are unit tests specific to each targeted platform (NM/DCMI/DNM) and also one for the library as
a whole. Each unit test is very simple and is run in either In band or Out of Band mode. The unit test starts out testing basic communication and then continues on to perform readings and then “capping” functions. During all the tests, it checks the response data to make sure it is what we expect. As long as all tests passed, then the unit test exits, outputting an all tests passed message. If a test fails during the unit test, then the unit will stop testing, output what failed and why, and then exit.

Out band Test Setup –

Host computer to compile and run Intel Node Manager Programmer’s reference kit

Connected using Management LAN

Intel® Server with Intel Node Manager Technology

In band Test Setup –

Intel Server with Intel Node Manager Technology host computer to compile kit

4 High Level Interface API

These API will be available in the form a library in the first version and, in a later version, they can be extended to a Web interface.

4.1 Initialize/UnInitialize Software Subsystem

Discover capabilities of nodes and initialize all required data structures. This should always be the first function called when using the library and the first time running commands against a device.

Function Prototype:

```c
bool swSubSystemSetup (initType_t, nmprk::ipmi::device*);
```

Input Parameters:

- `initType_t`: Denotes the type of setup occurring, includes device init, device uninit, library init, library uninit. This is used by the library to determine what type of action to perform.

- `nmprk::ipmi::device*`: Denotes which device the init/uninit is supposed to happen on. When `initType_t` is equal to library init/uninit, this argument is ignored. When `initType_t` does not equal a library operation, then it is assumed that device* is not equal to NULL and that the field members address and type of device data structure are set (when address denotes a remote host, then user and password fields are also assumed to be populated).
Output:

**bool**: The output of this function is a bool used to indicate if the init/uninit was successful. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception.

### 4.2 Get Capability

Get the Node Manager related capabilities of the platform. This function should be used to see what type of support a device provides, including such a thing as the max policies supported and other information related to the BMC/ME.

**Function Prototype:**

```c
capabilities_t* getCapabilities(device*);
```

**Input Parameters:**

- **device**: Denotes which device to return the capabilities of. It is assumed that device is not equal to NULL and that the field members address and type are set (when address denotes a remote host then user and password fields are also assumed to be populated).

**Output:**

- **capabilities_t**: This function returns a pointer to a structure that holds all the capabilities of that device. The actual memory that the pointer points to is allocated by the library, using standard C++ memory allocation techniques (new). Once the consumer is finished with this variable, it is their responsibility to appropriately de-allocate this memory, using a standard call delete call. Refer to section 5.1 to see the full definition of the capabilities_t variable.

### 4.3 Get Samples

Get power, thermal, and other supported types of samples from the NM capable platform. Use this function to get the current power or thermal reading for a device, including the systems subsystems (if supported by the platform).

**Function prototype**

```c
sample_t* getSample(device*, sampleType_t, subSystemComponentType_t, policy_t*);
```
Input Parameters

**device**: Denotes which device to return the capabilities of. It is assumed that device is not equal to NULL and that the field members address and type are set (when address denotes a remote host then user and password fields are also assumed to be populated).

**sampleType_t**: Denotes the type of sample to take which will either be of type sampleType_t::reading_power or sampleType_t::reading_thermal. Because not all platforms implement or support taking thermal readings, calling for a thermal reading on said platforms will cause an nmprkExcept to be thrown, which will have an error message reading “Platform does not support thermal readings” with a matching error code.

**subSystemComponentType_t**: Denotes which subsystem component (equivalent to domains in NM speak) to take a reading from, including the entire system, cpu sub system, or memory subsystem. If the library consumer asks for the sample reading of a subsystem the platform does not support, then a nmprkExcept will be thrown with an error message reading “Platform does not support X subsystem”, where x is the subsystem designated by subSystemCompentType_t with a matching error code.

**policy_t**: Denotes which policy to take a reading from. In the case where the consumer does not want to get a reading for a specific policy, the consumer should set policy_t equal to NULL. If policy_t is not equal to NULL, then it is assumed that the field members component and policyId of policy_t data structure are populated.

Output:

**sample_t**: This function returns a pointer to a structure containing the requested sample. The actual memory that the pointer points to is allocated by the library, using standard C++ memory allocation techniques (new) so once the consumer is finished with this variable, it is his responsibility to appropriately de-allocate this memory using a standard call delete call. Refer to section 5.3 to see the full definition of the sample_t variable.

4.4 Reset Statistics

Reset statistics for a platform. This function resets all the internal statistics that the device uses in its averaging and reporting.

Note: DCMI doesn’t support the concept of resetting statistics, so calling this function for a dcmi machine will result in an exception with the code of NMPRK_CMD_NOT_SUPPORT_CODE.
**Function prototype:**

```c
bool resetStatistics (device*, subSystemComponentType_t, policy_t);
```

**Input Parameters:**

- `device*`: Denotes which device to return the capabilities of. It is assumed that `device*` is not equal to NULL and that the field members address and type of device data structure are set
  (when address denotes a remote host, then user and password fields are also assumed to be populated).

- `subSystemComponentType_t`: Denotes which subsystem component (equivalent to domains, in NM speak) to take a reading from, including the entire system, cpu sub system, or memory subsystem. If the library consumer asks for the sample reading of a subsystem the platform does not support, then a `nmprkExcept` will be thrown with an error message reading “Platform does not support X sub system”, where x is the subsystem designated by `subSystemComponentType_t` with a matching error code.

- `policy_t`: Denotes which policy to take a reading from. In the case where the consumer does not want to get a reading for a specific policy, the consumer should set the `policy_t` equal to NULL.

**Output:**

- `bool`: The output of this function is a bool used to indicate if the statistics reset was successful. In most cases, this should return true because, most of the time, a result of false will cause an `nmprkExcept` to be thrown, which will include further error information about why it failed to be included in the exception (that is, `subSystemComponent_t` not supported, `policy_t` not valid).

4.5 **Get Policy**

Note: DMCI does not support multiple policies. When calling any policy function on a dcmi machine, the library will ignore the policyId and just set it as the default policy. Calling into a dcmi machine with 2 different policies, where the first policy id is 2 and the second policy id is 3, what happens is policy id 3 is the only active policy or, in case of this function, it returns only policy id 3.

Get an NM policy. This function is used to get a currently set policy. You can use this function to loop thru and see all the currently set policies.
Function prototype:

    policy_t*  getPolicy (device*,policy_t*);

Input Parameters:

    device*: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type of policy_t data structure are set
    (when address denotes a remote host then user and password fields are also assumed to be populated).

    policy_t*: Denotes which policy to take a reading from. At this point, policy_t is assumed to not be equal to NULL and that the field members component and policyId of policy_t data structure have been correctly populated.

Output:

    policy_t*: This function returns the policy_t* that was passed to it as an argument, once it has populated all the field members. Because the library is just returning the same policy_t* it was passed, no additional memory allocation is done by the library. When this variable is no longer in use, it should be de-allocated through whichever means the consumer allocated the memory. Refer to section 5.2 to see the full definition of the policy_t variable.

4.6 Set Policy

Set an NM policy. This function can be used to set a policy on a device. This is equivalent to setting a “cap” on that system that will keep the power or thermal readings from going over the limit specified in the policy.

Note: DMCI does not support multiple policies. When calling any policy function on a dcmi machine, the library will ignore the policyId and just set it as the default policy. So calling in to a dcmi machine with 2 different policies, where the first policy id is 2 and the second policy id is 3, what happens is policy id 3 is the only active policy.

Function prototype:

    bool                 setPolicy(device*,policy_t*);

Input Parameters:

    device*: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type of device data structure are set
(when address denotes a remote host, then user and password fields are also assumed to be populated).

**policy_t**: Denotes which policy to take a reading from. At this point, policy_t is assumed to not be equal to NULL and the field members component, policyId, policyType, policyLimit, sendAlert (send alert if system exceeds powerLimit for longer than correctionTime), shutdown (shutdown system if system exceeds powerLimit for longer than correctionTime) of policy_t data structure are set correctly. The user can also optionally set the members to correctionTime and statReportingPeriod to a custom value for any specialized needs. If the user doesn’t need a custom value here, but wants to go with the default recommended (30 second statReportingPeriod and 10 second correctionTime) value, they can populate it to zero.

**Output:**

**bool**: The output of this function is a bool used to indicate if the policy was set successfully. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (that is subSystemComponent_t not supported, policy_t not valid)

### 4.7 Delete Policy

Delete an NM policy. This function is used to remove and delete a currently set policy.

Note: DCMI doesn’t have the concept of “delete a policy”, or even the concept of policies. Calling this function on a dcmi machine translates to calling setPolicyStatus and passing it a policyStatusType_t of policyDisabled.

**Function prototype**

```c
bool delPolicy(device*,policy_t*);
```

**Input Parameters**

**device**: Denotes which device to return the capabilities of. It is assumed that the device* is not equal to NULL and that the field members address and type are set (when the address denotes a remote host, then user and password fields are also assumed to be populated).

**policy_t**: Denotes which policy to take a reading from. At this point policy_t is assumed not to be equal to NULL and that the field members component and policyId have been correctly populated.
Output

**bool**: The output of this function is a bool used to indicate if the policy was deleted successfully. In most cases, this should return back true because, most of the time, a result of false will cause an nmprkExcept to be thrown, which will include further error information about why it failed to be included in the exception (that is policy_t not valid)

### 4.8 Set Policy Status

Set a policy status to Enable or Disable. This function allows you to enable or disable a policy without having to delete or re-add it to the device. This is useful when you want to turn off a policy, but you know you will use it again later and you don’t want to delete it.

**Note**: DMCI does not support multiple policies. So, when calling any policy function on a dcmi machine, the library will ignore the policyId and just set the status of the default policy.

**Function prototype**

```c
bool setPolicyStatus(device*,policy_t*,policyStatusType_t);
```

**Input Parameters**

- **device***: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set (when the address denotes a remote host, then the user and password fields are also assumed to be populated).

- **policy_t***: Denotes which policy to take a reading from. At this point, policy_t is assumed not to be equal to NULL and that the field members component and policyId of policy_t data structure have been correctly populated.

- **policyStatusType_t**: Denotes what state to set the policy referred to by policy_t. This value will either be policyStatusType_t::policyEnabled or policyStatusType_t::policyDisabled.

**Output**

**bool**: The output of this function is a bool used to indicate if the policy status was successfully set. In most cases, this should return true because, most of the
time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (that is, policy_t not valid)

4.9 Register Alerts – Initiate call backs as needed

There are two types of events you can register for in the library. The first is events that happen on a policy and the second one is events that happen on the system. They are described below.

Register callback mechanism to trigger alerts for certain events. Call callback when a policy has an alert/event. This function can be used to track when a policy goes into or out of effect (power/thermal reading reaches limit set by policy).

Function prototype

```c
bool registerPolicyHandler (device*, policy_t*, alertHandler_t);
```

**Input Parameters**

- **device***: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type of device data structure are set (when address denotes a remote host, then the user and password fields are also assumed to be populated).

- **policy_t***: Denotes which policy to take a reading from. At this point, policy_t is assumed not to be equal to NULL and that the field members component and policyId have been correctly populated.

- **alertHandler_t** : Denotes the call back function when a policy raises an alert/event. alertHandler_t is defined as void(*alertHandler_t)(alertType_t,alertDirType_t,alert_t*). The alertType_t is used so the callback function can know what type of alert this is. The alertDirType_t defines if this is a assert(alarm raised) or deassert(alarm released). The alert_t holds the actual alert info. It should be noted that the alert_t is allocated by the library thru a standard new allocation. The user is responsible to de-allocate the memory using standard C++ methods (delete).

**Output**

**bool**: The output of this function is a bool used to indicate if the policy was deleted successfully. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (that is,
Call callback when an alert of the desired type happens on a system.

**Function prototype**

```c
bool registerAlertHandler(device*, alertType_t, alertHandler_t);
```

**Input Parameters**

- `device*`: Denotes which device to return the capabilities of. It is assumed that `device*` is not equal to NULL and that the field members address and type are set (when address denotes a remote host, then user and password fields are also assumed to be populated).

- `alertType_t`: Denotes the alert type for the library to catch (that is Exception Event, Health Event, Operation Capabilities Change, and Alert Threshold Exceeded). When the library sees an alert of type `alertType_t`, it will call the `alertHandler_t` registered with it passing it all the event info.

- `alertHandler_t`: Denotes the call back function for when an alert of type `alertType_t` is raised an alert/event. `alertHandler_t` is defined as `void(*alertHandler_t)(alertType_t,alertDirType_t,alert_t*)`. The `alertType_t` is used so the callback function can know what type of alert this is. The `alertDirType_t` defines if this is a assert(alarm raised) or deassert(alarm released). The `alert_t` holds the actual alert info. It should be noted that the `alert_t` is allocated by the library through a standard new allocation. The user is responsible to de-allocate the memory using standard C++ methods (delete).

**Output**

- `bool`: The output of this function is a bool used to indicate if the policy was deleted successfully. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (that is, the device is not valid)
5 High Level APIs Data Structure Definitions

5.1 Capabilities Structure

This structure defines the capabilities for a device. Mainly contains the min and max values supported.

typedef struct {
    unsigned int maxConSettings; // The total number of connects this device supports
    unsigned int maxTriggerValue; // max value the device supports for a reading
    unsigned int minTriggerValue; // min value the device supports for a reading
    unsigned int minCorrectionTime; // the min value accepted for how long a system has
    // to lower its power/thermal below the limit set by a //
    // policy before the system takes corrective action
    unsigned int maxCorrectionTime; // same as previous but the max value
    unsigned int minStatReportPeriod; // the shortest time supported for averaging stats
    unsigned int maxStatReportPeriod; // the longest time supported for averaging stats
} capabilities_t;

5.2 Policy Structure

Used to define the type of policy, which is either Thermal (temperature) or Power (watts).

typedef enum {
    policyPower,
    policyThermal
} policyType_t;

This is used to specify a subsystem of the device. When no specific subsystem is needed, pass in componentSystem as the default.

typedef enum {
    componentSystem,
    componentCpu,
    componentMemory
} subSystemComponentType_t;
This is the structure that actually defines the policy. It is used to define not only what type of policy and the limit, but also the corrective actions to perform and other details that determine how the policy acts.

typedef struct {
    subSystemComponentType _t component;  // Used to Specify which part of the system this policy is for. In most cases, this will
    unsigned int policyId;                // The ID of this policy. If set to -1 then use //subSystemComponentType_t::componentSystem
    policyType_t policyType;              // Type of policy. Either thermal or power
    unsigned int policyLimit;             // The value to keep the system under.
                                      // If the system goes over this value for longer then
                                      // the correction time specified, then the following
                                      // actions will be performed if set to true
    bool sendAlert;                      // Action to perform if over limit for longer then
                                      // correction time
    bool shutdown;                       // Action to perform if over limit for longer then
                                      // correction time
    unsigned int correctionTime;         // How long a system can be over policyLimit
                                      // before action is taken by the system
    unsigned int statReportingPeriod;    // the length of time over which stats are averaged
    bool policyEnabled;                  // When adding the policy, determines if its enabled
                                      // or disabled
    bool perSubSystemCompentEnabled;     // sets per domain policy control
    bool globalPolicyControlEnabled;     // sets global policy control
} policy_t;

5.3 Sample Power / Thermal Reading Structure

This defines which type of reading to perform. Either Thermal (temp) or Power (watts).

typedef enum {
    samplePower,
    sampleThermal
} sampleType_t
This defines the actual sample reading that is returned by getSample.

```c
typedef struct {
    sampleType_t sampleType; // Which type of sample, thermal or power
    unsigned int cur;       // The current sample reading
    unsigned int min;       // the min reading seen over last
                            // statReportingPeriod
    unsigned int max;       // Max reading seen during past reporting
                            // period
    unsigned int avg;       // Average reading for the past reporting
                            // period
    tm timestamp;           // The time stamp for the reading
    unsigned int statReportingPeriod; // the length of the reporting period
}sample_t;
```

6  IPMI Interface API Definition

6.1  Connect To a Device

Make a basic connection to a device to allow the consumer to start running commands
against a device. Before any actions can be taken on a device, you first need to connect to
it. Trying to run commands against a device that isn’t connected will result in exceptions
being thrown.

**Function prototype:**

```c
bool connectDevice(device* d);
```

**Input Parameters**

_device*:_ Denotes which device to return the capabilities of. It is assumed that
device* is not equal to NULL and that the field members address and type are set
(when address denotes a remote host then user and password fields are also
assumed to be populated correctly, otherwise the value of address should be set to
“local”).
Output

**bool**: The output of this function is a bool used to indicate if the device was connected to successfully. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (that is, connection refused, bad authentication).

6.2 Disconnect a Device

Disconnect from a device. This function is important to call on remote devices because otherwise all the connections in the network stack will stay open and hang in a stale state.

Function prototype:

```c
bool disconnectDevice(device* d);
```

Input Parameters

**device***: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.

Output

**bool**: The output of this function is a bool used to indicate if the device was connected to successfully. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (that is, already disconnected).

6.3 Run Ipmi Command

Run a basic ipmi type command on a device. The most basic building block of the library, as it allows you to send any command to any device. Most users will not need to use this function, but when users identify additional functionality that is required that the library doesn’t currently provide, this function can be used to perform the command by allowing the user to specify the request bytes and then have them manually interpret the resulting response bytes.

Function Prototype:

```c
bool runIpmiCommand(device* d, commandReq_t*, commandRsp_t*)
```

Input Parameters
device*: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.

commandReq_t: Denotes the command to be run on the device. It is assumed commandReq_t* is not equal to NULL and that the structure members req is not equal to NULL and that len is correctly set to the number of bytes in the request. After this function returns, it is up to the consumer to de-allocate the member pointed to by commandReq_t->req through whatever means it was allocated. The library does not take responsibility for handling the de-allocation of this memory.

commandRsp_t*: Denotes the response from the command pointed to by commandReq_t*. It is assumed that commandRsp_t is not equal to NULL but that is all that is expected from the consumer. The library will allocate the memory holding the response (commandRsp_t->rsp) through standard C++ allocation and will set the value of commandRsp_t->len to the number of bytes in the response. After finishing with the commandRsp_t, it is the consumer’s responsibility to de-allocate the memory pointed to by commandRsp_t->rsp using a standard C++ delete[] call. The consumer can then de-allocate commandRsp_t through whatever means it was allocated.

Output

bool: The output of this function is a bool used to indicate if the command was successfully run on the device. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (device not valid, command not valid, incorrect length of command)

6.4 Initialize a System for Local Commands

This command will set up the local system for running commands in band. This usually involves making sure the correct drivers are loaded. For out of band communication this function is not required and will never be used. It’s only needed when the library is running on the system, wanting to be communicated with, and in band communication is needed. After making sure the required drivers are loaded, the library will then attempt to run a test command to make sure it has been set up correctly and is working.

Function Prototype:

bool initSystemForLocal();

Input Parameters

Output

bool: The output of this function is a bool that is used to indicate if the device
was set up successfully. It should be noted that this command will only return true after the library has been able to successfully run a command in band on the system (the library has a list of some basic commands to use to test in band communication). This function will never return a false because a return value of false will cause a nmprkExept to be thrown, which will include further error information of why it failed included in the exception (in band not supported (older supermicros), incorrect system authorization level (non root user for linux, non admin for windows)).

6.5 Get SEL Info

Return the info for the SEL (system event log) on the device. This info will be useful when looping through the SEL to read entries because it will let you determine how many entries there are.

Function Prototype:

```cpp
nmprk::ipmi::repoInfo_t* getSelInfo(device* d);
```

Input Parameters

- `device*`: Denotes which device to return the capabilities of. It is assumed that `device*` is not equal to NULL and that the field members address and type are set.

Output

- `repoInfo_t`: This function returns a structure holding all the current information and capabilities of the SEL repo. It should be noted that the library will allocate the memory for repoInfo_t through standard c++ methods, so when the consumer is finished, they can de-allocate with a c++ delete call. Please refer to section 7.1 to see the exact details of the repoInfo_t structure. It should be noted that record_t* will never return back NULL because a value of NULL will cause a nmprkExept to be thrown, which will include further error information about why it failed included in the exception.

6.6 Get SEL Record

Returns a record from the SEL. This is the function to use when wanting to loop through the SEL to read all the events that are currently happening on the device. This is a very simple task because all that’s required is to call to begin by calling into this with an address of 0,0 and then continuing to call this function, passing it the address of the next entry that is returned in the record_t* info. By doing this until the address of the next entry is FF,FF, you can guarantee to hit all the entries in the SEL.
## Function Prototype:

```c
nmprk::ipmi::record_t* getSelRecord(device* d, address_t*);
```

### Input Parameters

- **device***: Denotes which device to return the capabilities of. It is assumed that `device*` is not equal to NULL and that the field members address and type are set.

- **address_t***: Denotes the address in the SEL to read the record from. It is assumed, at this point, that `address_t` is not equal to NULL and that the field members lsb and msb are set correctly (to read from the whole SEL, start with lsb and msb of zero and then use the value of `record_t->nextRecord` as the value for the next call, which allows you to iterate through the SEL reading all the records).

### Output

- **record_t***: The output of this function returns a structure holding the record pointed to by `address_t`. The consumer can safely assume that `record_t` and `record_t->data` are not NULL and that `record_t->len` is set to the number of bytes pointed to by `record_t->data`. The library will allocate the memory for `record_t` and `record_t->data` through standard C++ methods, so when the consumer is finished they can de-allocate with a C++ delete call (delete[] `record_t->data`, delete `record_t`). In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (device not valid, command not valid, incorrect length of command).

### 6.7 Delete SEL Record

Delets a record from the SEL. Simple function to allow you to remove an entry from the SEL. This can be useful to remove events that you are aware of from the event log.

### Function Prototype:

```c
bool addSelRecord(device* , address_t*);
```

### Input Parameters

- **device***: Denotes which device to return the capabilities of. It is assumed that `device` is not equal to NULL and that the field members address and type are set.

- **address_t***: Denotes the address in the SEL to delete. It is assumed, at this point, that `address_t` is not equal to NULL and that the field members lsb and msb are set correctly.
Output

*bool:* The output of this function is a bool used to indicate if the record was successfully deleted from the SEL. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (device not valid, address_t not valid)

6.8 Clear the SEL

Clear all records in the SEL. Simply put, clear all events from the system and empty out the log.

**Function Prototype:**

```cpp
bool clrSel(nmprk::ipmi::device* d);
```

**Input Parameters**

device*: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.

Output

*bool:* The output of this function is a bool used to indicate if the SEL was successfully cleared. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (device not valid)

6.9 Register Event – Initiate call backs as needed

Register callback mechanism to trigger alerts for certain events. Call callback when a device has an alert/event. Exactly like the function listed above and provides all the same functionality. By having it here, as well, users that only want to use the ipmi engine of the library can still get the same access to events that is provided by the translation engine.

**Function Prototype:**

```cpp
bool registerEventHandler(device*,sensorFilter_t, eventFilter_t,eventHandler_t*)
```

**Input Parameters**

device*: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.
sensorFilter_t: Denotes the type of sensor to watch for an event on (System Event, Temperature, Platform Alert, Power unit, Processor, Fan and Drive slot).

eventFilter_t: Denotes a regular expression to use to filter out event messages. If this is set to anything besides "*", then the library will first make sure that the event happening matches the sensorFilter_t. It will then try to match eventFilter_t to the event. If there is a match, the library will call the eventHandler_t, passing it the event info.

Output

bool: The output of this function is a bool used to indicate if the callback successfully registered. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (device not valid).

6.10 Unregister an Event Handler

Remove an event handler so the consumer no longer gets notified of events of that type.

Function Prototype:

bool unregisterEventHandler(device*, sensorFilter_t)

Input Parameters

device*: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.

sensorFilter_t: Denotes the type of sensor to watch for an event on (System Event, Temperature, Platform Alert, Power unit, Processor, Fan and Drive slot).

Output

bool: The output of this function is a bool used to indicate if the command was successfully removed. At this point, if the function returns true, then the consumer will receive no more callbacks for events of type sensorFilter_t. This is also one of the few functions that will never throw an exception. A return value of false means there was no eventHandler registered for type sensorFilter_t.

6.11 Get SDR Info

Return the info for the SDR on the device. This can be used much the same way as the Get SEL info command.
Function Prototype:

```
nmprk::ipmi::repoInfo_t* getSdrInfo(device* d);
```

Input Parameters

- **device***: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.

Output

- **repoInfo_t**: This function returns a structure holding all the current information and capabilities of the Sdr repo. It should be noted that the library will allocate the memory for repoInfo_t thru standard c++ methods, so when the consumer is finished they can de-allocate with a c++ delete call. Please refer to section 7.1 to see the exact details of the repoInfo_t structure. It should be noted that record_t* will never return NULL because a value of NULL will cause a nmprkExecpt to be thrown, which will include further error information, of why it failed, included in the exception.

6.12 Get SDR Record

Returns a record from the SDR. Once again this function can be used just the same as the getSelRecord function to allow you to loop through and read all the records in the SDR. This is very useful because important information is stored in the SDR, including bridge and transport information for Node Manager devices.

Function Prototype:

```
nmprk::ipmi::record_t* getSdrRecord(device* d, address_t*)
```

Input Parameters

- **device***: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.

- **address_t***: Denotes the address in the SDR to read the record from. It is assumed, at this point, that address_t* is not equal to NULL and that the field members lsb and msb are set correctly (to read from the whole SDR, start with lsb and msb of zero and then use the value of record_t->nextRecord as the value for the next call, this will allow you to iterate through the SEL reading all the records).

Output
record_t*: The output of this function returns a structure holding the record pointed to by address_t*. The consumer can safely assume that record_t* and record_t->data are not NULL and that record_t->len is set to the number of bytes pointed to by record_t->data. The library will allocate the memory for record_t and record_t->data through standard c++ methods, so when the consumer is finished he can de-allocate with a c++ delete call (delete[] record_t->data, delete record_t). In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (device not valid, command not valid, incorrect length of command).

6.13 Delete SDR Record

Deletes a record from the SDR. Allows you to remove a record from the SDR. This is highly discouraged, unless you know what you are doing and why you want to do it. This is because important information is stored in the SDR (like the bridge and transport address of the ME for node manager devices) and parts of the functionality of the library require being able to read through the SDR and retrieve information it needs from it.

Function Prototype:

bool addSdrRecord(device* , address_t*);

Input Parameters

device*: Denotes the device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.

address_t*: Denotes the address in the SDR to delete. It is assumed, at this point, that address_t* is not equal to NULL and that the field members lsb and msb are set correctly.

Output

bool: The output of this function is a bool used to indicate if the record was successfully deleted from the SDR. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (device not valid, address_t not valid).

6.14 Clear the SDR

Clear all records in the SDR. Remove all the entries from the SDR. Just like above, this is only recommended if you know why you want to perform this operation.
Function Prototype:

```cpp
bool clrSel(nmprk::ipmi::device* d);
```

**Input Parameters**

- `device*`: Denotes which device to return the capabilities of. It is assumed that `device*` is not equal to NULL and that the field members address and type are set.

**Output**

- `bool`: The output of this function is a bool used to indicate if the Sdr was successfully cleared. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (device not valid)

### 6.15 Get FRU Info

Return the info for the FRU on the device. Can be used much the same way as the Get SEL info command.

**Function Prototype:**

```cpp
fruInfo_t* getFruInfo(device* d);
```

**Input Parameters**

- `device*`: Denotes which device to return the capabilities of. It is assumed that `device*` is not equal to NULL and that the field members address and type are set.

**Output**

- `fruInfo_t`: This function returns a structure holding all the current information and capabilities of the Fru. Please refer to section 7.10 for the full definition of the `fruInfo_t` structure. It should be noted that the library will allocate the memory for `fruInfo_t` thru standard c++ methods so when the consumer is finished they can de-allocate with a c++ delete call. Please refer to section 7.1 to see the exact details of the `repoInfo_t` structure.

### 6.16 Get Fru Data

Returns data from the FRU. Just like the other getXXXRecord functions, this one can be used to loop through and read all entries in the Fru. This can be useful when wanting to get all of the sensor and device info for the system.
Function Prototype:

fruData_t* getFruData(device* d, address_t*, unsigned int)

Input Parameters

device*: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.

address_t*: Denotes the address in the Fru to read the record from. It is assumed, at this point, that address_t* is not equal to NULL and that the field members lsb and msb are set correctly (to read from the whole Sdr, start with lsb and msb of zero and then use the value of record_t->nextRecord as the value for the next call, this will allow you to iterate through the SEL reading all the records).

Unsigned int: Denotes the amount of data to read from the Fru. This value is assumed to be > 0 but less than fruInfo_t->fruSize

Output

fruData_t*: The output of this function returns a structure holding the data pointed to by address_t*. Please see section 7.11 for the full definition of fruData_t. The consumer can safely assume that fruData_t* and fruData_t->data are not NULL and that fruData_t->len is set to the number of bytes pointed to by fruData_t->data. The library will allocate the memory for record_t and record_t->data through standard c++ methods, so when the consumer is finished they can de-allocate with a c++ delete call (delete[] record_t->data, delete record_t). In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (device not valid, command not valid, incorrect length of command)

6.17 Set Fru Data

Write data to the FRU. Use this function to add an entry into the fru.

Function Prototype:

bool setFruData(device*, address_t*, fruData_t*);

Input Parameters

device*: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.
address_t*: Denotes the address in the Fru to write to. It is assumed at this point that address_t* is not equal to NULL and that the field members lsb and msb are set correctly.

fruData_t*: Denotes the data to be written to the Fru at address_t. It is assumed both fruData_t and fruData_t->data are not equal to NULL and that fruData_t->len is equal to the number of bytes pointed to by fruData->data.

Output

bool: The output of this function is a bool used to indicate if the data was successfully written to the FRU. In most cases, this should return true because, most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (device not valid, address_t not valid)

6.18 Get Device ID

Return a devices ID. The most basic ipmi command. Perfect for use to test connection/response from a device. This function provides back the most basic info about a device including firmware version, what roles it provides (SEL, FRU, Bridge, etc), along with manufacturer info.

Function Prototype:

getDeviceIdRsp* getDeviceId(device*);

Input Parameters

device*: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.

Output

getDeviceIdRsp*: The output of this function is a structure that the device id for the device pointed to be device*. Refer to section 7.12 to see the full definition of the getDeviceIdRsp structure. At this point, the consumer can assume that getDeviceIdRsp is not equal to NULL and that all of its field members have been correctly set. It should be noted that the library allocates the memory for the getDeviceIdRsp using a standard c++ allocation so it is the consumer’s
responsibility when they are finished with it to de-allocate it using standard methods. It can also be noted that this function never returns NULL because a return value of NULL will result in an nmprkExept being thrown, which will include further error information of why it failed included in the exception (device not valid).

6.19 Get Acpi Power State

Returns the current acpi power state of the system.

Function Prototype:

acpiPwrState_t getAcpiPwrState(nmprk::ipmi::device*);

Input Parameters

device*: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.

Output

acpiPwrState: The output of this function is a enum used to indicate the current acpi status of the system. Please refer to section 7.16 for the full definition of acpiPwrState. The only cases where this function will throw a nmprkExept is in the case where the device pointed to by device* is not valid.

6.20 Set ACPI Power State

Set the acpi power state on the system.

Function Prototype:

bool setAcpiPwrState(nmprk::ipmi::device* d, acpiPwrState_t);

Input Parameters

device*: Denotes which device to return the capabilities of. It is assumed that device* is not equal to NULL and that the field members address and type are set.

acpiPwrState_t: Denotes the current acpi power state to set on the system. Refer to section 7.16 to see the full definition of the acpiPwrState_t structure.

Output

bool: The output of this function is a bool used to indicate if the acpi power state was correctly set on the system. In most cases, this should return true because,
most of the time, a result of false will cause an nmprkExept to be thrown, which will include further error information about why it failed to be included in the exception (device not valid, acpiPwrState_t not valid in current condition).

7 IPMI APIs’ Data Structure Definition

7.1 repoInfo_t Structure

This is the structure that holds all the info for the SEL. It’s useful to determine if certain functions are supported by a SEL, how many entries there are, and how much space is left.

typedef struct {
    byte_t selVersion;  // version of the SEL
    unsigned int selEntries;  // get total entries in the SEL
    unsigned int selFreeSpace;  // the amount of space left in the SEL
    tm mostRecentAddTS;  // timestamp from the last add command
    tm mostRecentDelTS;  // timestamp from the last delete command
    bool getAllocInfoSup;  // does this SEL support getAllocInfo command
    bool reserveSup;  // reserve SEL supported
    bool parAddSup;  // partial add SEL support
    bool delSup;  // SEL supports delete
} repoInfo_t;

This is the structure used to address a SEL/SDR/FRU entry. Typical usage for this is to start out with lsb and msb equal to zero and then set lsb/msb, based on values for the next record in the returned record info.

7.2 address_t Structure

typedef struct {
    byte_t lsb;  // least significant byte in addy
    byte_t msb;  // most significant byte
} address_t;

7.3 fruInfo_t Structure

typedef struct {
    byte_t fruVersion;
    unsigned int fruEntries;
    unsigned int fruFreeSpace;
} fruInfo_t;
tm     mostRecentAddTS;
tm     mostRecentDelTS;
bool    getAllocInfoSup;
bool    reserveSup;
bool    parAddSup;
bool    delSup;
}fruInfo_t;

7.4 fruInfo_t Structure

The structure that holds the info for the FRU. Provides the size of the fru and if access is by word or byte.

typedef struct {
    byte_t fruSize[2];  // total size of fru
    bool accessByWord;  // access by word or byte
}fruInfo_t;

7.5 fruData_t Structure

The structure that holds a fru entry info. This is the actual data in for that fru entry.

typedef struct {
    byte_t* data;  // data from the entry in the fru
    unsigned int len;  // size of data field
}fruData_t;

7.6 record_t Structure

The structure that holds SEL/SDR records. This is what is returned when calling the getXXXRecord commands. nextRecord holds the address of the next record and can be used to loop thru the SDR/SEL. The actual records data is stored in the data field and its length is denoted by the len field.

typedef struct {
    address_t nextRecord;  // address of next record
    byte_t* data;  // record data
    unsigned int len;  // length of data field
}record_t;
7.7 byte_t Definition

The definition of the basic data type for the library

typedef unsigned char byte_t;

7.8 deviceType_t enum

Type that is used to specify which type of platform a device is. When a user doesn’t know or wants the library to auto detect, then pass it a value of device_auto. Note that this will take longer then specifying the actual type. The recommended thing to do is to call in the first time with device_auto, and then take the returned info and use it to call back into the library with the exact type, so that following calls into the library are faster.

typedef enum {
    device_auto = 0,
    device_dcmi ,
    device_bmc ,
    device_nm ,
    device_dnm
}deviceType_t;

7.9 Device Class

The structure that holds the actual device information and is a required field in most commands. Fields required to be set for the library to work depend on the type of operation being performed. For inband, all that is needed is to set address to “local” and the bridge and transport. For Out of Band, the address will need to be set to the ip of the device, and the user and password fields will also need to be set along with the bridge and transport (bridge and transport should be set to nmprk::ipmi::default if bridging is not needed. This is the value they are set to by default during allocation).

class device {
    public:
        deviceType_t type;
        std::string address;
        std::string user;
        std::string password;
        byte_t bridge;
        byte_t transport;
        device(deviceType_t Type, std::string Address, std::string User, std::string Password, byte_t Bridge, byte_t Transport):
type(Type), address(Address), user(User), password(Password), bridge(Bridge), transport(Transport) {};

7.10 Class nmprkExcept

The class that is thrown when an exception happens in the library. All calls into the library should be surrounded with try { } catch(nmprkExcept ne) { } as all functions can throw exceptions in the library. Upon catching an exception, the fields errorCode are set to the value of the error, as defined in <nmprkExceptions.h>

class nmprkExcept {
public:
byte_t errorCode;
std::string errorMsg;
nmprkExcept(byte_t code,std::string msg):
    errorCode(code), errorMsg(msg) {}
};

7.11 commandReq_t Structure

The structure that holds the request (arguments) for an ipmi command. Note that the library depends on len being set correctly. If this is not, the library does not guarantee to correctly read all of the req bytes.

typedef struct {
  byte_t* req;
  unsigned int len;
}commandReq_t;

7.12 commandRsp_t Structure

The structure that holds the response bytes (return value) from running an ipmi command. The library guarantees that len will be set correctly.

typedef struct {
  byte_t* rsp;
  unsigned int len;
}commandRsp_t;

7.13 getDeviceIdRsp Structure

The structure that holds all the info about a device.
typedef struct {
  byte_t   deviceId;   // the id of this device
unsigned int deviceRev; // this devices revision
bool     deviceProvidesSdr; // does this device provide SDR access
unsigned int firmwareRev; // firmware revision
bool     devNormOp;      // device is in normal operation mode
unsigned int firmwareRev2; // firmware revision 2
bool     isSensorDev;    // this device is a sensor device
bool     isSdrRepoDev;   // is a sdr repo device
bool     isFruInvDev;    // is a fru
bool     isIpmbRevDev;
bool     isIpmiGenDev;   // is a ipmi device
bool     isBridgeDev;    // device acts as bridge
bool     isChassisDev;
std::string  manufId;    // manufacturer id
std::string  productId;  // product id
} getDeviceIdRsp;

7.14 resetReq_t Enum

the type of reset (reboot) operation to perform

typedef enum {
  resetCold = 0,
  resetWarm
} resetReq_t;

7.15 acpiSystemPwrState_t Enum

A way to denote all the different acpi power states.

typedef enum {
  stateS0G0 = 0,
  stateS1,
  stateS2,
  stateS3
  stateS4,
  stateS5G2,
  stateS4S5,
  stateG3,
  stateSleeping,
  stateG1Sleep,
  stateOverRide,
  stateLegacyOn = 0x20,
};
stateLegacyOff,
stateUnknown,
stateNoChange = 0x7f
}acpiSystemPwrState_t;

7.16 acpiDevicePwrState Enum

A way to denote all the different ACPI power states for a device.

typedef enum {
    stateD0 = 0x0,
    stateD1,
    stateD2,
    stateD3,
    stateUnknown = 0x2a,
    stateNoChange = 0x7f
}acpiDevicePwrState_t;

7.17 acpiPwrState_t Structure

A structure to hold a system's power state and device power state.

typedef struct {
    acpiSystemPwrState_t systemState;
    acpiDevicePwrState_t deviceState;
}acpiPwrState_t;
8 Sample Code

Please refer to usage documents.
# Error Codes and Descriptions

```c
#define NMPRK_NULL_CODE 0x00
#define NMPRK_NULL_MSG "Function passed a NULL reference"
#define NMPRK_FAILED_ALLOC_CODE 0x01
#define NMPRK_FAILED_ALLOC_MSG "Library failed to allocate the memory required for the return structure"
#define NMPRK_INVALID_DOMAIN_CODE 0x02
#define NMPRK_INVALID_DOMAIN_MSG "Specified domain is not valid for this platform"
#define NMPRK_CMD_FAILED_CODE 0x03
#define NMPRK_CMD_FAILED_MSG "Command failed with Unknown Error"
#define NMPRK_CMD_NOT_SUPPORT_CODE 0x04
#define NMPRK_CMD_NOT_SUPPORT_MSG "Command / Functionality is not supported on this device"
#define NMPRK_DCMI_NO_MEASUREMENT_CODE 0x05
#define NMPRK_DCMI_NO_MEASUREMENT_MSG "No Measurements Available"
#define NMPRK_NM_DCMI_NO_DEV_CODE 0x06
#define NMPRK_NM_DCMI_NO_DEV_MSG "Could not open device at /dev/ipmi0, /dev/ipmi/0 or /dev/ipmidev/0"
#define NMPRK_NM_DCMI_NO_EVENT_RCV_CODE 0x07
#define NMPRK_NM_DCMI_NO_EVENT_RCV_MSG "Could not enable event receiver"
#define NMPRK_NM_DCMI_NOSET_IPMB_ADDY_CODE 0x08
#define NMPRK_NM_DCMI_NOSET_IPMB_ADDY_MSG "Could not set IPMB address"
#define NMPRK_NM_DCMI_UNABLE_SND_CMD_CODE 0x09
#define NMPRK_NM_DCMI_UNABLE_SND_CMD_MSG "Unable to send command"
#define NMPRK_NM_DCMI_IO_ERR_SND_CMD_CODE 0x0A
#define NMPRK_NM_DCMI_IO_ERR_SND_CMD_MSG "I/O Error Getting CMD RSP"
#define NMPRK_NM_DCMI_NO_DATA_RSP_CODE 0x0B
#define NMPRK_NM_DCMI_NO_DATA_RSP_MSG "No data available while getting CMD RSP"
#define NMPRK_CNVRT_TS_FAILED_CODE 0x0C
#define NMPRK_CNVRT_TS_FAILED_MSG "Function to return struct TM from timestamp returned NULL. Is SEL time set correctly?"
#define NMPRK_CMD_RETURNED_NON_ZERO_CODE 0x0D
#define NMPRK_CMD_RETURNED_NON_ZERO_MSG "CMD Returned back a non zero completion code: "
#define NMPRK_REQ_NOT_ENOUGH_ARGS_CODE 0x10
#define NMPRK_REQ_NOT_ENOUGH_ARGS_MSG "nmprk::ipmi::commandReq_t* req did not hold enough arguments, at least 2 are required to specify Net Function and cmd"
#define NMPRK_FAILED_INIT_DLL_CODE 0x11
#define NMPRK_FAILED_INIT_DLL_MSG "Failed to initialize nmct code plug-in libraries"
#define NMPRK_FAILED_OPEN_KCS_CODE 0x12
#define NMPRK_FAILED_OPEN_KCS_MSG "KCS Host Interface connect returned an error"
#define NMPRK_FAILED_OPEN_REMOTE_CODE 0x13
#define NMPRK_FAILED_OPEN_REMOTE_MSG "RMCPP Host Interface connect returned an error"
#define NMPRK_DEV_NOT_CONNECTED_CODE 0x14
#define NMPRK_DEV_NOT_CONNECTED_MSG "Device was not connected before operations were performed"
#define NMPRK_LNX_LAN_NO_PING_CODE 0x15
#define NMPRK_LNX_LAN_NO_PING_MSG "Unable to send IPMI presence ping packet"
#define NMPRK_LNX_LAN_NO_LAN_INTF_CODE 0x16
#define NMPRK_LNX_LAN_NO_LAN_INTF_MSG "Failed to open LAN interface"
#define NMPRK_LNX_LAN_NO_RTME_RSP_CODE 0x17
#define NMPRK_LNX_LAN_NO_RTME_RSP_MSG "No response from remote controller"
#define NMPRK_LNX_LAN_INVLD_RSP_PKT_CODE 0x18
#define NMPRK_LNX_LAN_INVLD_RSP_PKT_MSG "Invalid response packet"
#define NMPRK_LNX_LAN_PKT_SEND_FAIL_CODE 0x19
#define NMPRK_LNX_LAN_PKT_SEND_FAIL_MSG "Packet send failed"
#define NMPRK_LNX_LAN_INTERNAL_ERR_CODE 0x1A
#define NMPRK_LNX_LAN_INTERNAL_ERR_MSG "Lnx Lan portion of library had a fatal error: "
#define NMPRK_LNX_LAN_UNABLE_TO_EST_CODE 0x1B
#define NMPRK_LNX_LAN_UNABLE_TO_EST_MSG "Unable to establish a LAN session"
#define NMPRK_LNX_LAN_CLOSE_SES_FAIL_CODE 0x1C
#define NMPRK_LNX_LAN_CLOSE_SES_FAIL_MSG "Close Session Command Failed "
#define NMPRK_LNX_LAN_CLOSE_SES_FAIL_BAD_SES_ID_CODE 0x1D
```
#define NMPRK_LN_LN_CLOSE_SESSION_FAIL_BAD_SESSION_ID_MSG "Failed to close session due to bad session id : "
// LAN / OOB General Errors
#define NMPRK_LAN_INVALID_USER_CODE 0x20
#define NMPRK_LAN_INVALID_USER_MSG "Invalid user name"
#define NMPRK_LAN_NO_NULL_USER_CODE 0x21
#define NMPRK_LAN_NO_NULL_USER_MSG "Null user name not enabled"
#define NMPRK_LAN_GET_SESSION_CHALLENGE_CMD_FAIL_CODE 0x22
#define NMPRK_LAN_GET_SESSION_CHALLENGE_CMD_FAIL_MSG "Get Session Challenge command failed:
#define NMPRK_LAN_NO_SESSION_SLOTS_CODE 0x23
#define NMPRK_LAN_NO_SESSION_SLOTS_MSG "No session slot available"
#define NMPRK_LAN_NO_USR_SESSION_SLOTS_CODE 0x24
#define NMPRK_LAN_NO_USR_SESSION_SLOTS_MSG "No slot available for given user - limit reached"
#define NMPRK_LAN_NO_USR_PRIV_SLOTS_CODE 0x25
#define NMPRK_LAN_NO_USR_PRIV_SLOTS_MSG "No slot available to support user due to maximum privilege capacity"
#define NMPRK_LAN_USR_REQUEST_PRIVILEGE_EXCEED_CODE 0x26
#define NMPRK_LAN_USR_REQUEST_PRIVILEGE_EXCEED_MSG "Requested privilege level exceeds limit"
#define NMPRK_LAN_USR_PRIVILEGE_INSUFFICIENT_CODE 0x27
#define NMPRK_LAN_USR_PRIVILEGE_INSUFFICIENT_MSG "Insufficient privilege level"
#define NMPRK_LAN_SET_SESSION_PRIVILEGE_LEVEL_FAIL_CODE 0x28
#define NMPRK_LAN_SET_SESSION_PRIVILEGE_LEVEL_FAIL_MSG "Set Session Privilege Level Failed"
#define NMPRK_LAN_SOCKET_CONNECT_FAIL_CODE 0x29
#define NMPRK_LAN_SOCKET_CONNECT_FAIL_MSG "Socket Connection Failed"
#define NMPRK_LAN_SOCKET_CREATE_FAILED_CODE 0x2a
#define NMPRK_LAN_SOCKET_CREATE_FAILED_MSG "Socket Creation Failed"
#define NMPRK_LAN_ADDY_LOOKUP_FAILED_CODE 0x2b
#define NMPRK_LAN_ADDY_LOOKUP_FAILED_MSG "Address lookup failed"
#define NMPRK_LAN_NO_ADDY_SPECIFIED_CODE 0x2c
#define NMPRK_LAN_NO_ADDY_SPECIFIED_MSG "No hostname specified"
#define NMPRK_INVALID_COMMAND_CODE 0xc1
#define NMPRK_INVALID_COMMAND_MSG "Invalid Command (if this is NM did you forget to set the bridge/transport)"
#define NMPRK_NM_INVALID_DOMAIN_CODE 0xd1
#define NMPRK_NM_INVALID_DOMAIN_MSG "Invalid Domain Specified"
#define NMPRK_NM_INVALID_POLICY_TRIGGER_CODE 0xd2
#define NMPRK_NM_INVALID_POLICY_TRIGGER_MSG "Invalid policy trigger specified"
#define NMPRK_NM_INVALID_POLICY_TYPE_CODE 0xd3
#define NMPRK_NM_INVALID_POLICY_TYPE_MSG "Invalid policy type specified"
#define NMPRK_NM_INVALID_POLICY_ID_CODE 0xd4
#define NMPRK_NM_INVALID_POLICY_ID_MSG "Invalid Policy ID"
#define NMPRK_NM_INVALID_MODE_CODE 0xd5
#define NMPRK_NM_INVALID_MODE_MSG "Invalid Mode"
#define NMPRK_NM_INVALID_POLICY_ACTION_CODE 0xd6
#define NMPRK_NM_INVALID_POLICY_ACTION_MSG "Unknown or Unsupported Policy Configuration Action"
#define NMPRK_NM_POWER_LIMIT_OUT_OF_RANGE_CODE 0xd7
#define NMPRK_NM_POWER_LIMIT_OUT_OF_RANGE_MSG "Power Limit out of range"
#define NMPRK_NM_CORRECTION_TIME_OUT_OF_RANGE_CODE 0xd7
#define NMPRK_NM_CORRECTION_TIME_OUT_OF_RANGE_MSG "Correction Time out of range"
#define NMPRK_NM_POLICY_PARAMETER_RANGE_CODE 0xd8
#define NMPRK_NM_POLICY_PARAMETER_RANGE_MSG "Correction Time out of range"
#define NMPRK_NM_STATISTICS_REPORT_PERIOD_OUT_OF_RANGE_CODE 0xd8
#define NMPRK_NM_STATISTICS_REPORT_PERIOD_OUT_OF_RANGE_MSG "Statistics Reporting Period out of range"
#define NMPRK_NM_POLICY_ID_ALREADY_EXIST_CODE 0xd9
#define NMPRK_NM_POLICY_ID_ALREADY_EXIST_MSG "Policy Could not be updated since Policy Id already exists and is enabled"
#define NMPRK_NM_UNABLE_DISABLE_POLICY_CODE 0xda
#define NMPRK_NM_UNABLE_DISABLE_POLICY_MSG "Unable to disable policy"
#define NMPRK_NOT_IMPLEMENTED_CODE 0xff
#define NMPRK_NOT_IMPLEMENTED_MSG "This Code and or Function is not currently implemented"
10 Common Issues and Debugging Tips

1. DCMI doesn’t support the concept of multiple policies, or policies in general for that matter. It just has a set of function to set 1 “policy” that it treats as a global one. As a result, calling into any function that requires a policy_t* as an argument on a dcmi machine causes the library to ignore the value of policy_t::policyId and just sets the “default policy”. So, while technically the library will allow you to use multiple policy ids, you’re actually just fooling yourself because the most recent policy is the only one in effect.

2. If your code is catching a lot of exceptions with errorCode == NMPRK_CNVRT_TS_FAILED_CODE (value of 0x0C) this usually means the time on the SEL is set incorrectly and so, when we pass the timestamp passed on the SEL time to the function to generate a struct tm, it can return NULL if the value wasn’t valid. The way to fix this is to correctly set the time on the SEL. You can do this by running ipmitool sel time set “00/00/00 00:00:00”, using military time for the hours.