Intel® Node Manager Programmer’s Reference Kit implementation guide

Contents

[1 Introduction 4](#_Toc335907137)

[2 Document edit history 4](#_Toc335907138)

[3 Programmer’s kit overview 5](#_Toc335907139)

[4 Simple Test Setup and requirements 6](#_Toc335907140)

[5 High level interface API 7](#_Toc335907141)

[5.1 Initialize/UnInitialize S/W sub-system 7](#_Toc335907142)

[5.2 Get Capability 8](#_Toc335907143)

[5.3 Get Samples 8](#_Toc335907144)

[5.4 Reset Statistics 9](#_Toc335907145)

[5.5 Get Policy 10](#_Toc335907146)

[5.6 Set Policy 11](#_Toc335907147)

[5.7 Delete Policy 11](#_Toc335907148)

[5.8 Set policy status 12](#_Toc335907149)

[6 High level APIs Data structure definitions 14](#_Toc335907150)

[6.1 Capabilities structure 14](#_Toc335907151)

[6.2 Policy structures 14](#_Toc335907152)

[6.3 Sample Power / Thermal Reading structure 15](#_Toc335907153)

[7 IPMI Interface API definition 17](#_Toc335907154)

[7.1 Connect To a Device 17](#_Toc335907155)

[7.2 Disconnect a Device 17](#_Toc335907156)

[7.3 Run IPMI Command 18](#_Toc335907157)

[7.4 Initialize a System for local commands 19](#_Toc335907158)

[7.5 Get NM and IPMI Versions 19](#_Toc335907159)

[7.6 Get SEL Info 20](#_Toc335907160)

[7.7 Get SEL Record 20](#_Toc335907161)

[7.8 Delete SEL Record 21](#_Toc335907162)

[7.9 Clear the SEL 22](#_Toc335907163)

[7.10 Get SDR Info 22](#_Toc335907164)

[7.11 Get an SDR 23](#_Toc335907165)

[7.12 Add an SDR 24](#_Toc335907166)

[7.13 Delete an SDR 24](#_Toc335907167)

[7.14 Clear the SDR Repository 25](#_Toc335907168)

[7.15 Get FRU Info 26](#_Toc335907169)

[7.16 Get FRU Data 26](#_Toc335907170)

[7.17 Get FRU Repository 27](#_Toc335907171)

[7.18 Set FRU Data 28](#_Toc335907172)

[7.19 Get Device ID 28](#_Toc335907173)

[7.20 Get ACPI Power State 29](#_Toc335907174)

[7.21 Set ACPI Power State 29](#_Toc335907175)

[8 IPMI APIs’ Data structure definition 31](#_Toc335907176)

[8.1 repoInfo\_t structure 31](#_Toc335907177)

[8.2 nmVersion\_t structure 31](#_Toc335907178)

[8.3 address\_t structure 31](#_Toc335907179)

[8.4 fruInfo\_t structure 32](#_Toc335907180)

[8.5 record\_t structure 32](#_Toc335907181)

[8.6 byte\_t definition 32](#_Toc335907182)

[8.7 deviceType\_t enum 32](#_Toc335907183)

[8.8 device class 33](#_Toc335907184)

[8.9 class nmprkExect 33](#_Toc335907185)

[8.10 commandReq\_t structure 33](#_Toc335907186)

[8.11 commandRsp\_t structure 34](#_Toc335907187)

[8.12 getDeviceIdRsp structure 34](#_Toc335907188)

[8.13 resetReq\_t enum 34](#_Toc335907189)

[8.14 acpiSystemPwrState\_t enum 35](#_Toc335907190)

[8.15 acpiDevicePwrState enum 35](#_Toc335907191)

[8.16 acpiPwrState\_t structure 35](#_Toc335907192)

[9 Sample code 37](#_Toc335907193)

[10 Error codes and descriptions 38](#_Toc335907194)

[11 Common issues and debugging tips 41](#_Toc335907195)

# Introduction

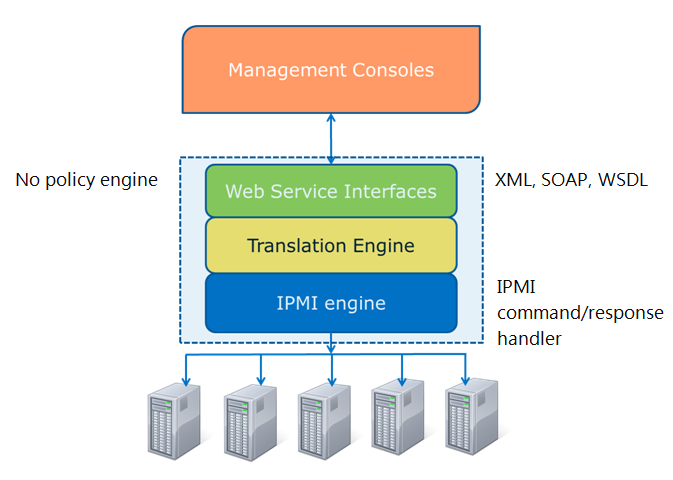
Intel® Node Manager Programmer’s Reference Kit implementation document discusses in detail about the various interfaces and associated APIs of various layers of kit. Document may refer to Intel® Node Manager Programmer’s Reference Kit as Programmer’s kit in various sections.

# Document edit history

|  |  |  |
| --- | --- | --- |
| Version | Author | Comments |
| First version | Simer P Singh | Create document added information based on the PPT data |
| 1.1 | Stewart Dale | Starting definitions of all functions and data structures(1/11/2012) |
|  | Simer Singh | Updated the overall architectural diagram and minor formatting (1/19/2012) |
|  | Simer Singh | Added section on Test setup, Error codes, common issues. (1/26/2012) |
|  | Stewart Dale | Updated definitions and explained data structures further  (2/1/12) |
| 1.2 | James Massoni | Updated definitions and explanations. Removed DCMI specific definitions. Updated messages, error codes and debugging information. Added/removed and corrected function calls and structures. |

# Programmer’s kit overview

Intel® Node Manager Programmer’s Reference Kit is a software package that contains all the required source code, test applications, 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 S/W eco-system ingredients to manage NM capable servers. Currently the Translation Engine and IPMI Engine are part of the NMPRK release. The Web Service Interface will be completed in a future release of the NMPRK.



# 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 then 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 provided for the targeted NM platforms and also one for the IPMI library as a whole. Each unit test is very simple and is either run in In-band or Out-of-Band mode. The unit test starts out by 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 exits.

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**

# High level interface API

These API will be available in the form a library in the first version and in later versions these will be extended to support a Web interface.

## Initialize/UnInitialize S/W sub-system

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:**

bool swSubSystemSetup (initType\_t, nmprk::ipmi::device\*);

**Input Parameters:**

**initType\_t :** Denotes the type of setup taking place, includes device init, device uninit, library init, library uninit. This is used by the library to determine what type of action to perform.

**device\*:** Denotes the device the init/uninit is supposed to take place 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 are set (when address denotes a remote host then user and password fields are also assumed to be populated). The actual memory that the pointer (device->intf) points to is allocated by the library using standard C++ memory allocation techniques (new) so once the calling application is finished with this structure it is the responsibility of the calling application to appropriately de-allocate this memory using a standard call delete call. This structure is used in the policy functions and should be deleted only after they are finished.

**Output:**

**bool:** The output of this function is a bool that is used to indicate if the init/uninit

was successful. In most cases this should return back true as most times a result

of false will result in an nmprkExept being throw which will include further error

information of why it failed included in the exception.

## Get Capability

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

**Function Prototype:**

capabilities\_t\* getCapabilities(device\*);

**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

(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) 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 5.1 to see the full definition of the capabilities\_t variable.

## Get Samples

Get power, thermal and other supported types of samples from 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**

sample\_t\* getSample(device\*, sampleType\_t, subSystemComponentType\_t, policy\_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

(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 . Since not all platforms implement or support taking thermal readings calling for a thermal reading on said platforms will cause in nmprkExcept being 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 either the entire system, cpu sub system or memory subsystem. If the library consumer asks for the sample reading of a sub system 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 sub system 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 then consumer should set policy\_t equal to NULL. If policy\_t is not equal to NULL then it is assumed that field members compenent and policyId 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.

## Reset Statistics

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

**Function prototype:**

bool resetStatistics (device\*, subSystemComponentType\_t, policy\_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

(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 either the entire system, CPU subsystem 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 then consumer should set policy\_t equal to NULL.

**Output:**

**bool:** The output of this function is a bool that is used to indicate if the statistics

reset was successful. In most cases this should return back true as most times a

result of false will result in an nmprkExept being throw which will include further

error information of why it failed included in the exception (ie…

subSystemCompent\_t not supported, policy\_t not valid).

## Get Policy

Get a 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 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

(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 have been correctly populated.

**Output:**

**policy\_t\* :** This function returns back the policy\_t\* that was passed to it as an argument once it has populated all the field members. Since 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 via which ever means the consumer allocated the memory. Refer to section 5.2 to see the full definition of the policy\_t variable.

## Set Policy

Set a 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.

**Function prototype:**

bool setPolicy(device\*,policy\_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

(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, 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) 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 that is used to indicate if the policy

was set successfuly. In most cases this should return back true as most times a

result of false will result in an nmprkExept being throw which will include further

error information of why it failed included in the exception (ie…

subSystemCompent\_t not supported, policy\_t not valid)

## Delete Policy

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

**Function prototype**

bool delPolicy(device\*,policy\_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

(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 have been correctly populated.

**Output**

**bool:** The output of this function is a bool that is used to indicate if the policy

was deleted successfully. In most cases this should return back true as most times

a result of false will result in an nmprkExept being throw which will include

further error information of why it failed included in the exception (ie…

policy\_t not valid)

## Set policy status

Set a policy status to Enable or Disable. This function allows you do 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 know you will use it again later and don’t want to delete.

**Function prototype**

bool setPolicyStatus(device\*,policy\_t\*,policyStatusType\_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

(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 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 that is used to indicate if the policy

status was successfully set. In most cases this should return back true as most

times a result of false will result in an nmprkExept being throw which will

include further error information of why it failed included in the exception (ie…

policy\_t not valid)

# High level APIs Data structure definitions

## 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;

## Policy structures

Used to define the type of policy trigger, which are: noPolicyTrigger, inletTempTrigger, missingPowerTimeout, afterHostResetTrigger and bootTimePolicy. In some cases you need to setup additional bytes in order for these flags to work. Please see the Intel® Intelligent Power Node Manager 2.0 specification for more details how to use these flags with the Set Policy command. Additionally, some servers only implement the use of triggers on lower policies, for example the inletTempTrigger may work with Policy 1, but not Policy 3.

Note: Previously the flags policyPower and policyThermal were defined here. Those names were inappropriate for these flags. Those names may still be used for backwards compatibility reasons, but in the future you should use the names defined below.

typedef enum {

noPolicyTrigger,

inletTempTrigger,

missingPowerTimeout,

afterHostResetTrigger,

bootTimePolicy

}policyType\_t;

This structure is used to specify a specific subsystem of the device. The subsystem componentSystem actually means a platform wide policy and is the default.

typedef enum {

domainSystem,

domainCpu,

domainMemory

}subSystemComponentType\_t;

This is the structure that actually defines the policy. It is used to define not only what type of policy and 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

//subSystemComponentType\_t::componentSystem

unsigned int policyId; // The ID of this policy. If set to -1 then use

// the first free policy otherwise policy #policyId

policyType\_t policyType; // Type of trigger policy to use.

unsigned int policyTriggerLimit; // 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

aggessiveCorrect\_t aggressiveCorrect;// used to define how aggressively to set the policy

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;

## 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.

typdef 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;

# IPMI Interface API definition

## 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:**

bool connectDevice(device\* d);

**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

(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 that is used to indicate if the device

was connected to successfully. In most cases this should return back true as most

times a result of false will result in an nmprkExept being throw which will include

further error information of why it failed included in the exception (ie…

connection refused, bad authentication).

## Disconnect a Device

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

**Function prototype:**

bool disconnectDevice(device\* d);

**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.

**Output**

**bool:** The output of this function is a bool that is used to indicate if the device

was connected to successfully. In most cases this should return back true as most

times a result of false will result in an nmprkExept being throw which will include

further error information of why it failed included in the exception (ie…

already disconnected)

## 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. Must 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 still perform the command by allowing the user to specify the request bytes and then have them manually interpret the resulting response bytes.

**Function prototype:**

bool runIpmiCommand(device\* d, commandReq\_t\*, commandRsp\_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.

**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 thru 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->len to the number of bytes

in the response. After finishing with the commandRsp\_t it is the consumers

responsibility to de-allocate the memory pointed to by commandRsp->rsp via

a standard c++ delete[] call. The consumer can then de-allocate commandRsp\_t

via whatever means it was allocated.

**Output**

**bool:** The output of this function is a bool that is used to indicate if the command was succesfully run on the device. In most cases this should return back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid, command not valid, incorrect length of command)

## Initialize a System for local commands

This command will setup the local system for running commands in band. This usually involves making sure the correct drivers are loaded. For consumers that will only band communication this function is not required and will never be used. Only needed when 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 setup 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 as a return value of false will cause a nmprkExept being throw 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)).

## Get NM and IPMI Versions

This command will get the NM and IPMI versions that are currently supported on the target system. This information may be used to determine which functions and parameters that are used later in the application, as different versions of Node Manager and IPMI support different features.

**Function prototype:**

nmVersion\_t\* getNMVersion(d);

**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.

**Output**

**nmVersion\_t\*:** The output of this function is a structure that contains flags indicating whether IPMI version 1.0 or IPMI version 2.0 is supported, and flags that indicate whether NM version 1.5 or NM version 2.0 is supported. Look to the nmVersion\_t\* definition to see the detailed description of the flags defined.

## Get SEL Info

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

**Function prototype:**

nmprk::ipmi::repoInfo\_t\* getSelInfo(device\* d);

**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.

**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 thru standard c++ methods so when the consumer is finished he 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 notes that record\_t\* will never return back NULL as a value of NULL will cause a nmprkExecpt to being throw which will include further error information of why it failed included in the exception. Standard IPMI defined error codes will be returned through this mechanism.

## Get SEL Record

Returns a record from the SEL. This is the function to use when wanting to loop thru the SEL to read all the events that are currently happening on the device. This is a very simple task as all that’s required is to call to begin by calling into this with 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:**

nmprk::ipmi::record\_t\* getSelRecord(device\* d, 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 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, this will allow you to iterate thru 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 thru 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 back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid, command not valid, incorrect length of command). Standard IPMI defined error codes will be returned through this mechanism.

## Delete SEL Record

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

**Function prototype:**

bool deleteSelRecord(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 SEL to delete. The only allowed values

are either 0x0000 (the beginning) or 0xFFFF (the end). It is assumed that address\_t\* is not equal to NULL and that the field members lsb and msb are set

correctly to either 0x0000 or 0xFFFF.

**Output**

**bool:** The output of this function is a bool that is used to indicate if the record was successfully deleted from the SEL. In most cases this should return back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid, address\_t not valid). Standard IPMI defined error codes will be returned through this mechanism.

## Clear the SEL

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

**Function prototype:**

bool clearSel(nmprk::ipmi::device\* d);

**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.

**Output**

**bool:** The output of this function is a bool that is used to indicate if the SEL was successfully cleared. In most cases this should return back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid). Standard IPMI defined error codes will be returned through this mechanism.

## Get SDR Info

Return the info for the SDR on the device. 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 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.

**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 he 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 notes that record\_t\* will never return back NULL as a value of NULL will cause a nmprkExecpt to being throw which will include further error information of why it failed included in the exception. Standard IPMI defined error codes will be returned through this mechanism.

## Get an SDR

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 thru and read all the records in the SDR. This is very useful as 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 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 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 thru 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 thru 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 back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid, command not valid, incorrect length of command). Standard IPMI defined error codes will be returned through this mechanism.

## Add an SDR

Allows you to add a record to the SDR repository. This is highly discouraged unless you know what you are doing and why you are doing 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 requires being able to read thru the SDR repository and retrieve needed information from it.

**Function prototype:**

bool addSdrRecord(device\* , address\_t\* record\_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 repository to add the SDR. 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.

**record\_t\*:** Denotes the SDR to add. It is assumed at this point that record\_t\*

is not equal to NULL and that the data area contains valid information.

**Output**

**bool:** The output of this function is a bool that is used to indicate if the record was successfully added to the SDR. In most cases this should return back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid, address\_t not valid). Standard IPMI defined error codes will be returned through this mechanism.

## Delete an SDR

Allows you to remove a record from the SDR repository. This is highly discouraged unless you know what you are doing and why you are doing 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 a parts of the functionality of the library require being able to read thru the SDR and retrieve information it needs from it.

**Function prototype:**

bool delSdrRecord(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 that is used to indicate if the record was successfully deleted from the SDR. In most cases this should return back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid, address\_t not valid). Standard IPMI defined error codes will be returned through this mechanism.

## Clear the SDR Repository

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:**

bool clearSdr(nmprk::ipmi::device\* d);

**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.

**Output**

**bool:** The output of this function is a bool that is used to indicate if the SDR was successfully cleared. In most cases this should return back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid). Standard IPMI defined error codes will be returned through this mechanism.

## Get FRU Info

Return the info for the FRU on the device.

**Function prototype:**

fruInfo\_t\* getFruInfo(device\* d);

**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.

**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 thru 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 back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid, command not valid, incorrect length of command). Standard IPMI defined error codes will be returned through this mechanism.

## Get FRU Data

Returns all data from the FRU. This can be useful when wanting to get all of the FRU information for the system.

**Function prototype:**

record\_t\* getFruData(device\* d, int offset, int length)

**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.

**int:** Denotes the offset in the FRU Repository from where to read the data. This

value is assumed to be > 0 but less than fruInfo\_t->fruSize

**int:** Denotes the amount of data to read from the Fru. This value is should be

> 0 but less than 0x20. If larger that 0x20 bytes, then it will be truncated to 0x20

bytes

**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 thru 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 back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid, command not valid, incorrect length of command). Standard IPMI defined error codes will be returned through this mechanism.

## Get FRU Repository

Returns all data from the FRU Repository. This can be useful when wanting to get all of the FRU information for the system.

**Function prototype:**

record\_t\* getFruRpry(device\* d, 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.

**record\_t\*:** Denotes the FRU data to add. It is assumed at this point that

record\_t\* is not equal to NULL and that the data area contains valid information.

**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 thru 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 back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid, command not valid, incorrect length of command). Standard IPMI defined error codes will be returned through this mechanism.

## 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\* , record\_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 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.

**record\_t\*:** Denotes the data to be written to the FRU at address\_t. It is assumed

both record\_t and record\_t->data are not equal to NULL, and that record\_t->len is

equal to the number of bytes pointed to by record\_t->data.

**Output**

**bool:** The output of this function is a bool that is used to indicate if the data was successfully written to the FRU. In most cases this should return back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid, address\_t not valid). Standard IPMI defined error codes will be returned through this mechanism.

## 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 manufacture info.

**Function prototype:**

getDeviceIdRsp\* getDeviceId(device\*);

**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.

**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 via a standard c++ allocation so it is the consumer’s responsibility when he is finished with it to del-allocate it via standard methods. It can also be noted that this function never return NULL as a return value of NULL will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid)

## 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 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.

**Output**

**acpiPwrState:** The output of this function is a enum that is 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.

## 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 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.

**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 that is used to indicate if the acpi power state was correctly set on the system. In most cases this should return back true as most times a result of false will result in an nmprkExept being throw which will include further error information of why it failed included in the exception (device not valid,acpiPwrState\_t not valid in current condition)

# IPMI APIs’ Data structure definition

## repoInfo\_t structure

This is the structure that holds all the info for the SEL. 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 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 next record in the returned record info.

## nmVersion\_t structure

typedef struct {

bool nmVersion\_1pt5; // The value indicates what node manager version

bool nmVersion\_2pt0; // the server is

bool ipmiVersion\_1pt0; // The value indicates what ipmi version the

bool ipmiVersion\_2pt0; // server is.

}nmVersion\_t;

## address\_t structure

typedef struct {

byte\_t lsb; // least significant byte in address

byte\_t msb; // most significant byte

}address\_t;

## 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;

## record\_t structure

The structure that holds SEL/SDR/FRU 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/FRU. 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[1024]; // record data

unsigned int len; // length of data field

}record\_t;

## byte\_t definition

The definition of the basic data type for the library

typedef unsigned char byte\_t;

## 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;

## 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 then 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) {}

};

## class nmprkExect

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 a exception the fields errorCode are set to the value of the error as defined in <nmprk\_Excpetions.h>

class nmprkExcept {

public:

byte\_t errorCode;

std::string errorMsg;

nmprkExcept(byte\_t code,std::string msg):

errorCode(code), errorMsg(msg) {}

};

## commandReq\_t structure

The structure that holds the request (arguments) for a 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;

## commandRsp\_t structure

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

typedef struct {

byte\_t\* rsp;

unsigned int len;

}commandRsp\_t;

## 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; // manfuctor id

std::string productId; // product id

}getDeviceIdRsp;

## resetReq\_t enum

the type of reset (reboot) operation to perform

typedef enum {

resetCold = 0,

resetWarm

}resetReq\_t;

## 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;

## acpiDevicePwrState enum

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

typdef enum {

stateD0 = 0x0,

stateD1,

stateD2,

stateD3,

stateUnknown = 0x2a,

stateNoChange = 0x7f

}acpiDevicePwrState\_t;

## acpiPwrState\_t structure

a structure to hold a systems power state and device power state.

typedef struct {

acpiSystemPwrState\_t systemState;

acpiDevicePwrState\_t deviceState;

}acpiPwrState\_t;

# Sample code

Please refer to usage documents. Additionally, there are two test programs supplied with the source code, testIpmi.cpp and testTranslation,cpp. These are good examples on how to develop applications that are linked to the NMPRK libraries.

The test applications show the order in how to use the functions calls defined in the NMPRK. Furthermore they show how to set, modify and delete policies. How to monitor a server by polling it for power and temperature information. Also on how to get and clear the SEL, get, clear and program the SDR repository, and how to get and program the FRU repository.

Using the test applications should enable a programmer to quickly develop applications that are useful doing complicated tasks in monitoring and maintaining servers in a datacenter. The code is provided as example code and it is expected to be modified to be more robust and to a more complex environment.

# Error codes and descriptions

#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 allocat 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

// THE FOLLOWING IS USED

// to dynamically generated to include the

// non-zero exit code

#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"

// LINUX LAN SPECIFIC ERRORS

#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\_RMTE\_RSP\_CODE 0x17

#define NMPRK\_LNX\_LAN\_NO\_RMTE\_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\_SND\_FAIL\_CODE 0x19

#define NMPRK\_LNX\_LAN\_PKT\_SND\_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\_CLSE\_SES\_FAIL\_CODE 0x1c

#define NMPRK\_LNX\_LAN\_CLSE\_SES\_FAIL\_MSG "Close Session Command Failed "

#define NMPRK\_LNX\_LAN\_CLSE\_SES\_FAIL\_BAD\_SES\_ID\_CODE 0x1d

#define NMPRK\_LNX\_LAN\_CLSE\_SES\_FAIL\_BAD\_SES\_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\_SES\_CHAL\_CMD\_FAIL\_CODE 0x22

#define NMPRK\_LAN\_GET\_SES\_CHAL\_CMD\_FAIL\_MSG "Get Session Challenge command failed:"

#define NMPRK\_LAN\_NO\_SES\_SLOTS\_CODE 0x23

#define NMPRK\_LAN\_NO\_SES\_SLOTS\_MSG "No session slot available"

#define NMPRK\_LAN\_NO\_USR\_SES\_SLOTS\_CODE 0x24

#define NMPRK\_LAN\_NO\_USR\_SES\_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\_REQ\_PRIV\_EXCEED\_CODE 0x26

#define NMPRK\_LAN\_USR\_REQ\_PRIV\_EXCEED\_MSG "Requested privilege level exceeds limit"

#define NMPRK\_LAN\_USR\_PRIV\_INSUFF\_CODE 0x27

#define NMPRK\_LAN\_USR\_PRIV\_INSUFF\_MSG "Insufficient privilege level"

#define NMPRK\_LAN\_SET\_SES\_PRIV\_LVL\_FAIL\_CODE 0x28

#define NMPRK\_LAN\_SET\_SES\_PRIV\_LVL\_FAIL\_MSG "Set Session Privilege Level Failed"

#define NMPRK\_LAN\_SCKT\_CONNECT\_FAIL\_CODE 0x29

#define NMPRK\_LAN\_SCKT\_CONNECT\_FAIL\_MSG "Socket Connection Failed"

#define NMPRK\_LAN\_SCKT\_CREATE\_FAILED\_CODE 0x2a

#define NMPRK\_LAN\_SCKT\_CREATE\_FAILED\_MSG "Socket Creation Failed"

#define NMPRK\_LAN\_ADDY\_LOOKUP\_FAILED\_CODE 0x2b

#define NMPRK\_LAN\_ADDY\_LOOKUP\_FAILED\_MSG "Address look up failed"

#define NMPRK\_LAN\_NO\_ADDY\_SPECIFIED\_CODE 0x2c

#define NMPRK\_LAN\_NO\_ADDY\_SPECIFIED\_MSG "No hostname specified"

#define NMPRK\_INVALID\_CHECKSUM\_CODE 0x2d

#define NMPRK\_INVALID\_CHECKSUM\_MSG "Invalid checksum found in the FRU common area"

#define NMPRK\_NM\_INVALID\_POLICY\_ID 0x80

#define NMPRK\_NM\_INVALID\_POLICY\_ID\_MSG "Invalid policy ID"

#define NMPRK\_NM\_INVALID\_DOMAIN 0x81

#define NMPRK\_NM\_INVALID\_DOMAIN\_MSG "Invalid domain specified"

#define NMPRK\_NM\_INVALID\_POLICY\_TRIG 0x82

#define NMPRK\_NM\_INVALID\_POLICY\_TRIG\_MSG "Unsupported policy trigger specified"

#define NMPRK\_NM\_UNKNOWN\_POLICY\_TYPE 0x83

#define NMPRK\_NM\_UNKNOWN\_POLICY\_TYPE\_MSG "Unknown policy type specified"

#define NMPRK\_NM\_PWR\_LIMIT\_OUT\_RANGE 0x84

#define NMPRK\_NM\_PWR\_LIMIT\_OUT\_RANGE\_MSG "Power Limit out of range"

#define NMPRK\_NM\_COR\_TIME\_OUT\_RANGE 0x85

#define NMPRK\_NM\_COR\_TIME\_OUT\_RANGE\_MSG "Correction Time out of range"

#define NMPRK\_NM\_POL\_TRIG\_OUT\_RANGE 0x86

#define NMPRK\_NM\_POL\_TRIG\_OUT\_RANGE\_MSG "Policy trigger value out of range"

#define NMPRK\_NM\_INVALID\_MODE 0x88

#define NMPRK\_NM\_INVALID\_MODE\_MSG "Invalid mode"

#define NMPRK\_NM\_STAT\_REPORT\_OUT\_RANGE 0x89

#define NMPRK\_NM\_STAT\_REPORT\_OUT\_RANGE\_MSG "Statistics reporting period out of range"

#define NMPRK\_NM\_INVALID\_AGGRESSIVE\_BIT 0x8B

#define NMPRK\_NM\_INVALID\_AGGRESSIVE\_BIT\_MSG "Invalid aggressive power correction field"

// Define Standard IPMI return codes and messages

#define IPMI\_NODE\_BUSY 0xC0

#define IPMI\_NODE\_BUSY\_MSG "Node busy, could not process command"

#define IPMI\_INVALID\_COMMAND 0xC1

#define IPMI\_INVALID\_COMMAND\_MSG "Invalid or unsupported command"

#define IPMI\_INVALID\_LUN 0xC2

#define IPMI\_INVALID\_LUN\_MSG "Invalid command for specified LUN"

#define IPMI\_TIMEOUT\_PROCESS 0xC3

#define IPMI\_TIMEOUT\_PROCESS\_MSG "Timeout while processing command"

#define IPMI\_LACK\_OF\_STORAGE 0xC4

#define IPMI\_LACK\_OF\_STORAGE\_MSG "Command could not be processed due to lack of storage space"

#define IPMI\_RESERVATION\_CANCELED 0xC5

#define IPMI\_RESERVATION\_CANCELED\_MSG "Reservation canceled or invalid reservation ID"

#define IPMI\_REQ\_DATA\_TRUNCATED 0xC6

#define IPMI\_REQ\_DATA\_TRUNCATED\_MSG "Request data truncated"

#define IPMI\_DATA\_LENGTH\_INVALID 0xC7

#define IPMI\_DATA\_LENGTH\_INVALID\_MSG "Request data length invalid"

#define IPMI\_DATA\_LENGTH\_EXCEEDED 0xC8

#define IPMI\_DATA\_LENGTH\_EXCEEDED\_MSG "Request data field length limit exceeded"

#define IPMI\_PARAMETER\_OUT\_OF\_RANGE 0xC9

#define IPMI\_PARAMETER\_OUT\_OF\_RANGE\_MSG "Parameter out of range"

#define IPMI\_CANNOT\_RETURN\_DATA 0xCA

#define IPMI\_CANNOT\_RETURN\_DATA\_MSG "Cannot return number of requested data bytes"

#define IPMI\_SENSOR\_NOT\_PRESENT 0xCB

#define IPMI\_SENSOR\_NOT\_PRESENT\_MSG "Requested sensor, data or record not present"

#define IPMI\_INVALID\_DATA\_FIELD 0xCC

#define IPMI\_INVALID\_DATA\_FIELD\_MSG "Invalid data field in request"

#define IPMI\_ILLEGAL\_FOR\_SENSOR 0xCD

#define IPMI\_ILLEGAL\_FOR\_SENSOR\_MSG "Command illegal for specified sensor or record type"

#define IPMI\_COULD\_NOT\_BE\_PROVIDED 0xCE

#define IPMI\_COULD\_NOT\_BE\_PROVIDED\_MSG "Command response could not be provided"

#define IPMI\_DUPLICATE\_REQUEST 0xCF

#define IPMI\_DUPLICATE\_REQUEST\_MSG "Cannot execute duplicated request"

#define IPMI\_SDR\_IN\_UPDATE\_MODE 0xD0

#define IPMI\_SDR\_IN\_UPDATE\_MODE\_MSG "Response could not be provided, SDR repository in update mode"

#define IPMI\_FW\_IN\_UPDATE\_MODE 0xD1

#define IPMI\_FW\_IN\_UPDATE\_MODE\_MSG "Response could not be provided, device in firmware update mode"

#define IPMI\_BMC\_INITIALIZATION 0xD2

#define IPMI\_BMC\_INITIALIZATION\_MSG "Response could not be provided, BMC initialization in progress"

#define IPMI\_DESTINATION\_UNAVAIL 0xD3

#define IPMI\_DESTINATION\_UNAVAIL\_MSG "Cannot deliver request to selected destination"

#define IPMI\_INSUFFICIENT\_PRIV 0xD4

#define IPMI\_INSUFFICIENT\_PRIV\_MSG "Insufficient privilege level to execute command"

#define IPMI\_UNSUPPORT\_PRES\_STATE 0xD5

#define IPMI\_UNSUPPORT\_PRES\_STATE\_MSG "Command not supported in present state"

#define IPMI\_SUB\_FUNC\_DISABLED 0xD6

#define IPMI\_SUB\_FUNC\_DISABLED\_MSG "Parameter is illegal because sub-function is disabled"

#define NMPRK\_NM\_UNABLE\_DISABLE\_POL\_CODE 0xda

#define NMPRK\_NM\_UNABLE\_DISABLE\_POL\_MSG "Unable to disable policy"

#define NMPRK\_BRIDGE\_IPMIRECV\_ERR\_CODE 0xe0

#define NMPRK\_BRIDGE\_IPMIRECV\_ERR\_MSG "AUTO GENERATED BY CODE. THIS IS NOT USED"

#define NMPRK\_BRIDGE\_RECV\_TIMEOUT\_CODE 0xe1

#define NMPRK\_BRIDGE\_RECV\_TIMEOUT\_MSG "No data received after 5 seconds, aborting!"

#define NMPRK\_BRIDGE\_RECV\_DATA\_ERR\_CODE 0xe2

#define NMPRK\_BRIDGE\_RECV\_DATA\_ERR\_MSG "AUTO GENERATED BY CODE. THIS IS NOT USED"

#define NMPRK\_NOT\_IMPLEMNETED\_CODE 0xff

#define NMPRK\_NOT\_IMPLEMNETED\_MSG "This Code and or Function is not currently implemented"

# Common issues and debugging tips

1. The NMPRK is a reference kit and not production code. It is the responsibility of the calling program to verify all input data prior to making a call, and to ensure all functions are called in the correct order. The developer using the NMPRK is welcomed to modify the code for their needs and add additional error checking as they see fit. Issues may be entered into the issue database located at: <https://github.com/01org/NMPRK/issues>
2. DCMI functions are no longer supported by the NMPRK.
3. For most functions, if the IPMI call returns an IPMI defined error code, then that error code is returned to the calling application. In these cases the function returns back a result of false, and an nmprkExept is thrown which will include further error information being returned. Standard IPMI defined error codes will be returned through this mechanism, and may be tested for by checking “e->errorCode”. The error message may be viewed by printing “e->errorMsg”.
4. If you 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.
5. When testing with DCM on the network. If DCM is connected with a Server under test, this can make it difficult, to use certain NMPRK functions. DCM by design will not allow its policies to be modified or removed, thus it will block the NMPRK from changing them.
6. Calling setPolicy() to modify a policy. Node Manager will not allow a policy to be modified if it is enabled, therefore you must first disable the policy first, modify it using setPolicy(), and then enable the policy. This will allow you to do things such as modify the limit value after the policy has been set.
7. How to restore you SDR or FRU repository. SDRs are created specifically for each server. You must get the firmware package for your server and restore the SDR for you server that way. The SDR effects how your server functions, so just putting a generic SDR on your system won’t work. You can also restore the FRU using the firmware package for your server.
8. FRU call reports invalid checksum. The first 8 bytes of the FRU repository have a checksum. If that checksum is invalid, then the FRU is invalid and the test ends with the message “invalid checksum”, this may be because the FRU was never programmed in the first place. Unlike the SDR, the FRU is informational and doesn’t really affect the performance of the server. You can restore the FRU using the firmware package for your server.
9. Destructive FRU & SDR Tests. I suggest you run the IPMITEST first without the destructive tests enabled, and then save the SDR and FRU files. Rename the files so you know which server they go to. Only then enable the destructive tests and check the clearing and programming of the FRU and SDRs. You will then be able to use the saved SDR and FRU files to restore your system if the test fails. Note: There is no default SEL event log, so there is nothing to restore it with. Plus the firmware does not allow the SEL to be restored; it only allows new events to be entered.