Intel® Node Manager Programmer’s Reference Kit implementation guide

Contents

[1 Introduction 4](#_Toc323548869)

[2 Document edit history 4](#_Toc323548870)

[3 Programmer’s kit overview 5](#_Toc323548871)

[4 Simple Test Setup and requirements 6](#_Toc323548872)

[5 High level interface API 6](#_Toc323548873)

[5.1 Initialize/UnInitialize S/W sub-system 6](#_Toc323548874)

[5.2 Get Capability 7](#_Toc323548875)

[5.3 Get samples 7](#_Toc323548876)

[5.4 Reset Statistics 8](#_Toc323548877)

[5.5 Get Policy 9](#_Toc323548878)

[5.6 Set Policy 10](#_Toc323548879)

[5.7 Delete Policy 11](#_Toc323548880)

[5.8 Set policy status 12](#_Toc323548881)

[5.9 Register Alerts – Initiate call backs as needed 12](#_Toc323548882)

[6 High level APIs Data structure definitions 14](#_Toc323548883)

[6.1 Capabilities structure 14](#_Toc323548884)

[6.2 Policy structure 15](#_Toc323548885)

[6.3 Sample Power / Thermal Reading structure 16](#_Toc323548886)

[7 IPMI Interface API definition 17](#_Toc323548887)

[7.1 Connect To a Device 17](#_Toc323548888)

[7.2 Disconnect a Device 17](#_Toc323548889)

[7.3 Run Ipmi Command 18](#_Toc323548890)

[7.4 Initialize a System for local commands 19](#_Toc323548891)

[7.5 Get Sel Info 19](#_Toc323548892)

[7.6 Get Sel Record 20](#_Toc323548893)

[7.7 Delete Sel Record 21](#_Toc323548894)

[7.8 Clear the SEL 21](#_Toc323548895)

[7.9 Register Event – Initiate call backs as needed 22](#_Toc323548896)

[7.10 Unregister an event handler 23](#_Toc323548897)

[7.11 Get SDR Info 23](#_Toc323548898)

[7.12 Get Sdr Record 24](#_Toc323548899)

[7.13 Delete Sdr Record 24](#_Toc323548900)

[7.14 Clear the Sdr 25](#_Toc323548901)

[7.15 Get FRU Info 26](#_Toc323548902)

[7.16 Get Fru Data 26](#_Toc323548903)

[7.17 Set Fru Data 27](#_Toc323548904)

[7.18 Get Device ID 28](#_Toc323548905)

[7.19 Get Acpi Power State 28](#_Toc323548906)

[7.20 Set ACPI Power State 29](#_Toc323548907)

[8 IPMI APIs’ Data structure definition 29](#_Toc323548908)

[8.1 repoInfo\_t structure 29](#_Toc323548909)

[8.2 address\_t structure 30](#_Toc323548910)

[8.3 fruInfo\_t structure 30](#_Toc323548911)

[8.4 fruInfo\_t structure 30](#_Toc323548912)

[8.5 fruData\_t structure 31](#_Toc323548913)

[8.6 record\_t structure 31](#_Toc323548914)

[8.7 byte\_t definition 31](#_Toc323548915)

[8.8 deviceType\_t enum 31](#_Toc323548916)

[8.9 device class 32](#_Toc323548917)

[8.10 class nmprkExect 32](#_Toc323548918)

[8.11 commandReq\_t structure 33](#_Toc323548919)

[8.12 commandRsp\_t structure 33](#_Toc323548920)

[8.13 getDeviceIdRsp structure 33](#_Toc323548921)

[8.14 resetReq\_t enum 34](#_Toc323548922)

[8.15 acpiSystemPwrState\_t enum 34](#_Toc323548923)

[8.16 acpiDevicePwrState enum 34](#_Toc323548924)

[8.17 acpiPwrState\_t structure 35](#_Toc323548925)

[9 Sample code 36](#_Toc323548926)

[10 Error codes and descriptions 37](#_Toc323548927)

[11 Common issues and debugging tips 39](#_Toc323548928)

# Introduction

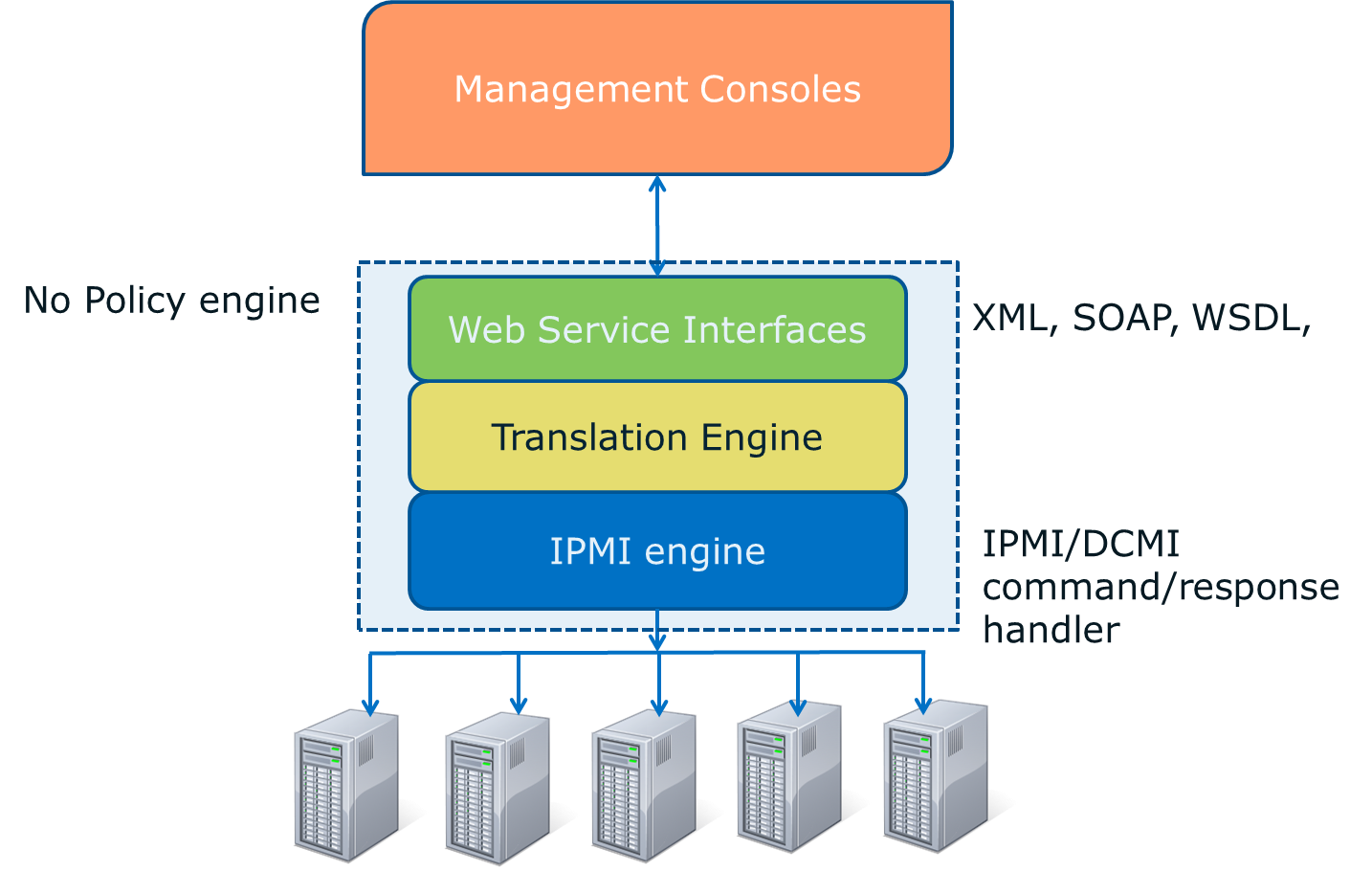
Intel® Node Manager Programmer’s Reference Kit implementation document discusses in detail about the various interfaces and associated APIs of various layers of kit. Document may refer to Intel® Node Manager Programmer’s Reference Kit as Programmer’s kit in various sections.

# 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) |

# 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 S/W eco-system ingredients to manage NM capable servers.



# 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 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 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 later version these can be extended to 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.

**nmprk::ipmi::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).

**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.

Note: DCMI doesn’t support the concept of resetting statistics. So calling this function for a dcmi machine will result in a exception with code of NMPRK\_CMD\_NOT\_SUPPORT\_CODE.

**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 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.

**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

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 it as the default policy. So calling in to a dcmi machine with 2 different policies, first policy id 2 then policy id 3, what ends up happening is policy id 3 is the only active policy or in case of this function, it returns only policy id 3

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.

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 it as the default policy. So calling in to a dcmi machine with 2 different policies, first policy id 2 then policy id 3, what ends up happening is policy id 3 is the only active 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.

Note: DCMI doesn’t have the concept of “delete a policy” much as it doesn’t have the concept of policies. Calling this function on a dcmi machine translates into calling setPolicyStatus and passing it a policyStatusType\_t of policyDisabled.

**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 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…

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.

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**

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)

## 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**

bool registerPolicyHandler (device\*,policy\_t\*, alertHandler\_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.

**alertHandler\_t :** Denotes the call back function for 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 via standard C++ methods (delete).

**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)

Call callback when alert of desired type happens on a system.

**Function prototype**

bool registerAlertHandler(device\*, alertType\_t, alertHandler\_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).

**alertType\_t:**  Denotes the alert type for the library to catch (ie… 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 alert of type alertType\_t is 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 via standard C++ methods (delete).

**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…

device 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 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 specific Sub System of the device. When no specific sub system 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 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 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;

## 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 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.

## 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 continueing 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)

## Delete Sel Record

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

**Function prototype:**

bool addSelRecord(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. 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 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)

## Clear the SEL

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

**Function prototype:**

bool clrSel(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)

## 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:**

bool registerEventHandler(device\*,sensorFilter\_t, eventFilter\_t,eventHandler\_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.

**sensorFilter\_t:** Denotes the type of sensor to watch for an event on (ie.. 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 then the eventFilter\_t the library will call the eventHandler\_t passing it the event info.

**Output**

**bool:** The output of this function is a bool that is used to indicate if the callback succesfully registered. 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)

## Unregister an event handler

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

**Function prototype:**

bool unregisterEventHandler(device\*, sensorFilter\_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.

**sensorFilter\_t:** Denotes the type of sensor to watch for an event on (ie.. System

Event, Temperature, Platform Alert, Power unit, Processor, Fan and Drive slot).

**Output**

**bool:** The output of this function is a bool that is used to indicate if the command was succesfully 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.

## 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.

## 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 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)

## 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 are wanting 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 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 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 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)

## 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:**

bool clrSel(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)

## 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:**

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**

**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 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.

## Get Fru Data

Returns data from the FRU. Just like the other getXXXRecord functions this one can be used to loop thru 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 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 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).

**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 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)

## Set Fru Data

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

**Function prototype:**

bool setFruData(device\*, address\_t\* , fruData\_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.

**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 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)

## 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.

## address\_t structure

typedef struct {

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

byte\_t msb; // most significant byte

}address\_t;

## fruInfo\_t structure

typedef struct {

byte\_t fruVersion;

unsigned int fruEntries;

unsigned int fruFreeSpace;

tm mostRecentAddTS;

tm mostRecentDelTS;

bool getAllocInfoSup;

bool reserveSup;

bool parAddSup;

bool delSup;

}fruInfo\_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;

## fruData\_t structure

The structure the 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;

## 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;

## 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.

# 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 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

// 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\_CMD\_CODE 0xc1

#define NMPRK\_INVALID\_CMD\_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\_TRIG\_CODE 0xd2

#define NMPRK\_NM\_INVALID\_POLICY\_TRIG\_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\_PWR\_LIMIT\_OUT\_RANGE\_CODE 0xd7

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

#define NMPRK\_NM\_COR\_TIME\_OUT\_RANGE\_CODE 0xd7

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

#define NMPRK\_NM\_POL\_TRIG\_OUT\_RANGE\_CODE 0xd7

#define NMPRK\_NM\_POL\_TRIG\_OUT\_RANGE\_MSG "Correction Time out of range"

#define NMPRK\_NM\_STAT\_REPORT\_OUT\_RANGE\_CODE 0xd8

#define NMPRK\_NM\_STAT\_REPORT\_OUT\_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\_POL\_CODE 0xda

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

#define NMPRK\_NOT\_IMPLEMNETED\_CODE 0xff

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

# 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. Because of this 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 only the most recent policy is the only one in effect.
2. 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.