Systune: A System Resource Provisioning Framework

Table of Contents

Introduction

Gaining control over system resources such as the CPU, caches, and GPU is a powerful capability in any developer's toolkit. By fine-tuning these components, developers can optimize the system's operating point to make more efficient use of hardware resources and significantly enhance the user experience.

For example, increasing the CPU's Dynamic Clock and Voltage Scaling (DCVS) minimum frequency to 1 GHz can boost performance during demanding tasks. Conversely, capping the maximum frequency at 1.5 GHz can help conserve power during less intensive operations.

The Systune framework supports `Signals` which is dynamic provisioning of system resources in response to specific signals—such as app launches or installations—based on configurations defined in YAML. It allows business units (BUs) to register extensions and add custom functionality tailored to their specific needs.

Systune Key Points

- Systune exposes a Variery of APIs for Resource Provisioning. These APIs can be directly used by the End-Client.
- Using these APIs the Client can Tune any System Resource Parameter, like cpu, dcvs, min / max frequencies etc.
- To provide a Convenient and Transparent Method for Clients to interact with the Systune Server, a Client Library is Provided, which takes care of Encoding and Sending the Request Message across to the Server for further Processing.
- A Request in this context, is a Group of Resources which need to Tuned for a certain (or possibly infinite) Duration.
- Systune also provides a Signal Framework which is useful for identifying Use Cases and Provisioning according to the Use Case.
- The Client is returned a Handle, a 64-bit Integer, which uniquely Identifies the Request.
- The Extension Interface Provides a way to Customize Systune Behaviour, by Specifying Custom Resources, Custom Signals and Features.
- Systune uses YAML based Config files, for fetching Information relating to Resources / Signals and Properties.

Systune Features

Design - Syslocks
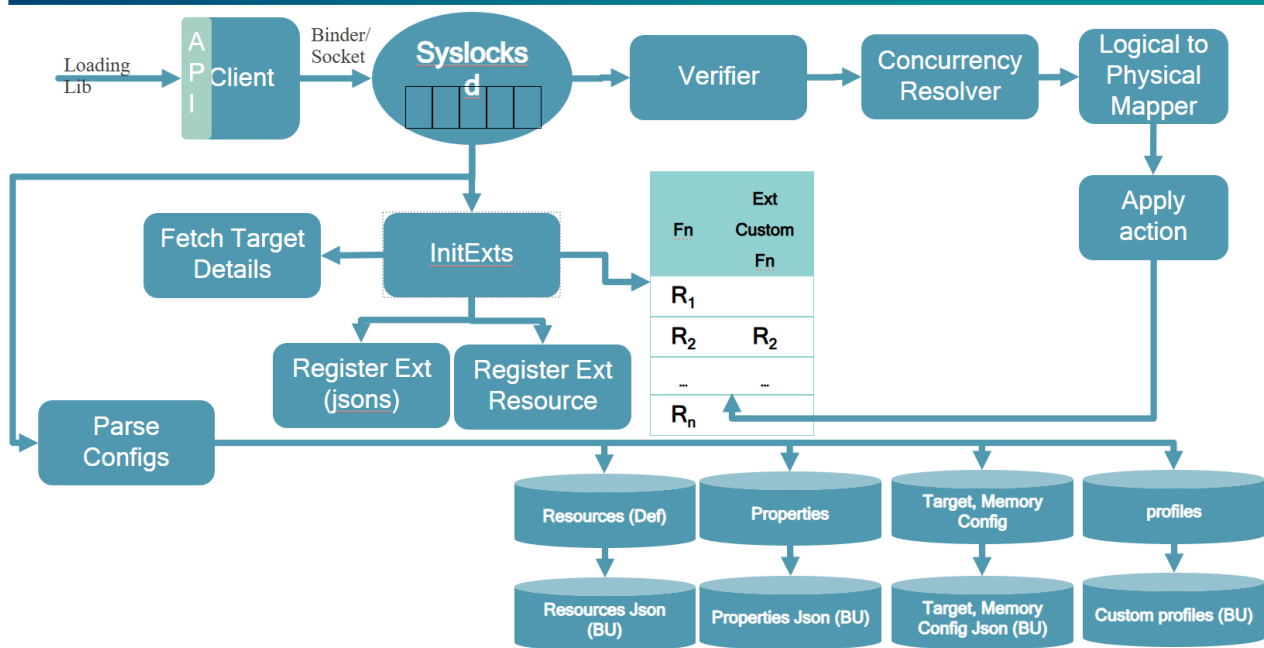
Systune Architecture is captured above.

Initialization

- During the Server Initialization Phase, the YAML Config Files are Read to build up the Resource Registry, Property Store etc.
- If the BU has Registered any Custom Resources, Signals or Custom YAML files via the Extension Interface, then these changes are detected during this Phase itself to build up a Consolidated System view, before it can start serving Requests.
- During the Initialization Phase, Memory is Pre-Allocated for Commonly used types (via Memory Pool), and Worker (Thread) capacity is reserved in advance via the ThreadPool, to avoid any delays during the Request Processing Phase.
- Systune will also Fetch the Target Details, like target Name, total Number of Cores, Logical to Physical Cluster / Core Mapping in this phase.
- If the Signals Module is Plugged In, it will be initialized as well and the Signal Configs will be Parsed similarly to Resource Configs.
- Once all the Initialization is completed, the Server is Ready to Serve Requests, a new Listener Thread is created for Handling Requests.

Request Processing

- The Client Can use the Systune Client Library to Send their Requests.
- Systune Supports Sockets and Binders for Client-Server Communication.
- As soon as the Request is received on the Server end, a Handle is generated and returned to the Client. This handle uniquely identifies the Request and can be used for subsequent Retune (retuneResources) or Untune (untuneResources) API calls.
- The Request is submitted to the ThreadPool for async Processing.
- When the Request is Picked up by a Worker (from the ThreadPool), it will first Decode the Request Message and then Validate the Request.
- The Request Verifier, will run a series of Checks on the Request like Permission Checks, and on the Resources part of the Request, like Config Value Bounds Check.
- Once Request is verified, a Duplicate Check is Performed, to verify if the Client has already submitted the same Request before. This is done so as to the improve System Efficiency and Performace.
- Next the Request is added to an Queue, which is essentially PriorityQueue, which orders Requests based on their Priorities (for more details on Priority Levels, refer the next Section). This is done so that the Request with the highest Priority is always served first.
- To Handle Concurrent Requests for the same Resource, we maintain Resource Level Linked Lists of Pending Requests, which are ordered according to the Request Priority and Resource Policy. This ensures that the Request with the higher Priority will always be applied first. For 2 Requests with the same Priority, the application Order will depend on Resource Policy. For example, in case of Resource with "Higher is Better" Policy, the Request with a higher Configuration Value for the Resource shall take effect first.
- Once a Request reaches the head of the Resource Level Linked List, it is applied, i.e. the Config Value specified by this Request for the Resource takes effect on the corresponding Sysfs Node.
- A timer is created and used to keep track of a Request, i.e. check if it has expired. Once it is detected that the Request has expired an Untune Request for the same Handle as this Request, is automatically generated and submitted, it will take care of Resetting the effected Resource Nodes to their Original Values.
- BUs can Provide their own Custom Appliers for any Resource. The Default Action provided by Systune is writing to the Resource Sysfs Node.

Here is a more detailed explanation of the key features discussed above:

1. Permissions

Certain resources can be tuned only by system clients and some which have no such restrictions and can be tuned even by third party clients. The Client permissions are dynamically determined, the first time it makes a Request. If a client with Third Party Permissions tries to tune a Resource, which allows only clients with System Permissions to tune it, then the Request shall be dropped.

2. Policies

To ensure efficient and predictable handling of concurrent requests, each system resource is governed by one of four predefined policies. Selecting the appropriate policy helps maintain system stability, optimize performance, and align resource behavior with application requirements.

- Instant Apply (or Always Apply): This policy is for resources where the latest request needs to be honored. This is kept as the default policy.
- Higher is better: This policy honors the request writing the highest value to the node. One of the cases where this makes sense is for resources that describe the upper bound value. By applying the higher-valued request, the lower-valued request is implicitly honored.
- Lower is better: Self-explanatory. Works exactly opposite of the higher is better policy.
- Lazy Apply: Sometimes, you want the resources to apply requests in a first-in-first-out manner.

3. Priorities

As part of the tuneResources API call, the Client is allowed to specify a desired Priority Level for the Request. Systune supports 4 priority levels:

- System High [SH]
- System Low [SL]
- Third Party High (or Regular High) [TPH]
- Third Party Low (or Regular Low) [TPL]

Requests with a higher Priority will always be prioritized, over another Request with a lower priority. Note, the Request Priorities are related to the Client Permissions. A client with System Permission is allowed to acquire any priority Level it wants, however a Client with Third Party Permissions can only acquire either Third Party High (TPH) or Third Party Low (TPL) level of Priorities. If a Client with Third Party Permissions tries to acquire a System High or System Low level of Priority, then the Request will not be honoured.

4. Detection of Dead Clients and Subsequent Cleanup

To improve efficiency and conserve Memory, it is essential to Regularly Check for Dead Clients and Free up any System Resources associated to them. This includes, Untuning all (if any) Ongoing Tune Request issued by this Client and Freeing up the Memory used to store Client Specific Data (Example: Client's List of Requests (Handles), Health, Permissions, Threads Associated with the Client etc). Systune Ensures that such clients are detected and Cleaned Up within 90 seconds of the Client Terminating.

Systune performs these actions by making use of two components:

- Pulse Monitor: Pulse Monitor scans the list of the Active Clients, and checks if any of the Client (PID) is dead (It does by checking if an entry for that PID exisits in /proc/pid/). If it finds a Dead Client, it schedules the Client for Cleanup by adding this PID to a Queue (called the GC Queue).
- Client Garbage Collector: When the Garbage Collector runs it iterates over the GC Queue and Performs the Cleanup.

Both Pulse Monitor and Client Garbage Collector run as Daemon Threads.

5. Preventing System Abuse

Systune has a built in RateLimiter component that prevents abuse of the system by limiting the number of requests a client can make within a given time frame. This helps to prevent clients from overwhelming the system with requests and ensures that the system remains responsive and efficient. RateLimiter works on a Reward / Punishment methodology. Whenever a Client enters the System for the first time, it is assigned a "Health" of 100. A Punishment is incurred if a Client makes subsequent Requests in a very short Time Interval (called Delta, say 5 ms). A Reward results in increasing the health of a Client (not above 100), while a Punishment involves decreasing the health of the Client. If at any point this value of Health reaches Zero then any further Requests from this Client wil be dropped. Note the Exact value of Delta, Punishment and Rewards are BU-configurable.

6. Duplicate Checking

Systune's RequestManager component is Responsible for detecting any duplicate Requests issued by a Client, and dropping them. This is done by maintaining a List of all the Requests issued by a Client. Whenever a new Request is received, it is checked against this List to see if it is a duplicate. If it is, then the Request is dropped. If it is not, then the Request is added to this List and processed. Duplicate Checking helps to improve System Efficiency, by saving wasteful CPU time on processing Duplicates.

7. Logical to Physical Mapping

Logical to Physical Core / Cluster Mapping helps us to achieve achieve decoupling on the Client side, as the Client does not need to be aware of the Physical Topology of the Target to issue Resource Tuning Requests. Instead the Client can specify Logical values for Core and Cluster. Systune will translate these values to their physical counterparts and apply the Request accordingly. Logical to Physical mapping in essence like System Independent Layer makes the same client code interchangable across different Targets, and Systune will take care of the mapping.

8. Display-Aware Operational Modes

The system's operational modes are influenced by the state of the device's display. To conserve power, certain system resources are optimized only when the display is active. However, for critical components that require consistent performance—such as during background processing or time-sensitive tasks, resource tuning can still be applied even when the display is off, including during low-power states like Doze mode. This ensures that essential operations maintain responsiveness without compromising overall energy efficiency.

9. Crash Recovery

In case of Server Crash, Systune ensures that all the Resource Sysfs Nodes are restored to a Sane State, i.e. they are reset to their Original Values. This is done by maintaining a List of all the Resource Sysfs Nodes and their Original Values, before any modification was made on behalf of the Clients by Systune. In the event of Server crash, this File is read and all Sysfs Nodes are reset to their Original Values.

10. Flexible Packaging

The Users are free to pick and Choose the Systune Modules they want for their use-case and which fit their constraints. The Framework Module is the core / central module, however if the Users choose they can add on top of it other Modules: Signals and Profiles.

11. Pre-Allocate Capacity for efficiency

Systune provides a MemoryPool component, which allows for pre-allocation of memory for certain commonly used type at the time of Server initialization. This is done to improve the efficiency of the system, by reducing the number of memory allocations and deallocations that are required during the processing of Requests. The allocated memory is managed as a series of blocks which can be recycled without any system call overhead. This reduces the overhead of memory allocation and deallocation, and improves the performance of the system.

Further, a ThreadPool component is provided to pre-allocate processing capacity. This is done to improve the efficiency of the system, by reducing the number of thread creation and destruction required during the processing of Requests, further ThreadPool allows for the Threads to be repeatedly reused for processing different tasks.

Config Files Format

Systune utilises YAML files for configuration. This includes the Resources, Signal Config Files. The BUs can provide their own Config Files, which are specific to their use-case through the Extension Interface

1. Resource Configs

Tunable Resources are specified via the resourceConfigs.yaml file. Each Resource is defined with the following fields:

Fields Description

Field	Type	Description	Default Value
ResID	string (Mandatory)	16-bit Resource Identifier, unique within the Resource Type.	Not Applicable
ResType	string (Mandatory)	8-bit Type of the Resource, for example: cpu / dcvs	Not Applicable
Name	string (Optional)	Path to the system sysfs node.	Empty String
Supported	boolean (Optional)	Indicates if the Resource is Eligible for Provisioning.	False
HighThreshold	integer (int32_t) (Mandatory)	Upper threshold value for the resource.	Not Applicable
LowThreshold	integer (int32_t) (Mandatory)	Lower threshold value for the resource.	Not Applicable
Permissions	string (Optional)	Type of client allowed to Provision this Resource (system or third_party).	third_party
Modes	array (Optional)	Display modes applicable ("display_on", "display_off", "doze").	display_on
Policy	string(Optional)	Concurrency policy ("higher_is_better", "lower_is_better", "instant_apply", "lazy_apply").	lazy_apply
CoreLevelConflict	boolean (Optional)	Indicates if the resource can have different values, across different cores.	False

Example

```
ResourceConfigs:
  - ResType: "0x1"
    ResID: "0x0"
    Name: "/proc/sys/kernel/sched_util_clamp_min"
    Supported: true
    HighThreshold: 1024
    LowThreshold: 0
    Permissions: "third_party"
    Modes: ["display_on", "doze"]
    Policy: "higher_is_better"
    CoreLevelConflict: false

  - ResType: "0x1"
    ResID: "0x1"
    Name: "/proc/sys/kernel/sched_util_clamp_max"
    Supported: true
    HighThreshold: 1024
    LowThreshold: 0
    Permissions: "third_party"
    Modes: ["display_on", "doze"]
    Policy: "lower_is_better"
    CoreLevelConflict: false
```

2. Properties Config

This targetPropertiesConfigs.yaml file stores various properties which are used by the Systune Modules internally (for example, to allocate sufficient amount of Memory for different Types, or to determine the Pulse Monitor Duration) as well as by the End Client.

Field Descriptions

Field	Type	Description	Default Value
Name	string (Mandatory)	Unique name of the parameter	Not Applicable
Value	integer (Mandatory)	The value for the parameter.	Not Applicable

Example

```
PropertyConfigs:
  - Name: systune.maximum.concurrent.requests
    Value: "60"
  - Name: systune.maximum.resources.per.request
    Value: "64"
  - Name: systune.listening.port
    Value: "12000"
  - Name: systune.pulse.duration
    Value: "60000"
```

3. Signal Configs

The file signalConfigs.yaml defines the Signal Configs.

Field Descriptions

Field	Type	Description	Default Value
SigId	string (Mandatory)	Signal Identifier	Not Applicable
Category	string (Mandatory)	Category of the Signal, for example: Generic, App Lifecycle.	Not Applicable
Name	string (Optional)		Empty String
Enable	boolean (Optional)	Indicates if the Signal is Eligible for Provisioning.	False
TargetsEnabled	array (Optional)	List of Targets on which this Signal can be Acquired	Empty List
TargetsEnabled	array (Optional)	List of Targets on which this Signal cannot be Acquired	Empty List
Permissions	array (Optional)	List of acceptable Client Level Permissions for tuning this Signal	third_party
Timeout	integer (Optional)	Default Signal Acquire Duration to be used in case the Client specifies a value of 0 for duration in the tuneSignal API call.	1000
Resources	array (Mandatory)	List of Resources.	Not Applicable

Example

```
SignalConfigs:
  - SigId: "0x0"
    Category: "0x1"
    Name: INSTALL
    Enable: true
    TargetsEnabled: ["sun", "moon"]
    Permissions: ["system", "third_party"]
    Derivatives: ["solar"]
    Timeout: 4000
    Resources:
      - {ResId: "0x0", ResType: "0x1", OpInfo: 0, Values: [700]}

  - SigId: "0x1"
    Category: "0x1"
    Name: EARLY_WAKEUP
    Enable: true
    TargetsDisabled: ["sun"]
    Permissions: ["system"]
    Derivatives: ["solar"]
    Timeout: 5000
    Resources:
      - {ResId: "0", ResType: "0x1", OpInfo: 0, Values: [300, 400]}
      - {ResId: "1", ResType: "0x1", OpInfo: 1024, Values: [12, 45]}
      - {ResId: "2", ResType: "0x2", OpInfo: 32, Values: [5]}
      - {ResId: "3", ResType: "0x4", OpInfo: 256, Values: [23, 90]}
      - {ResId: "4", ResType: "0x1", OpInfo: 512, Values: [87]}
```

4. (Optional) Target Configs

The file targetConfigs.yaml defines the Target Configs, not this an Optional Config, i.e. this file need not necessarily be provided. Systune can dynamically fetch system info, like Target Name, Logical to Physical Core / Cluster Mapping, number of cores etc. Use this file, if you want to provide this information explicitly. If the targetConfigs.yaml is provided, Systune will always Prioritize and use it. Also note, there are no field-level default values available if the targetConfigs.yaml is provided. Hence if you wish to provide this file, then you'll need to exhaustivly provide all the required information.

Field Descriptions

Field	Type	Description	Default Value
TargetName	string (Mandatory)	Target Identifier	Not Applicable
ClusterInfo	array (Mandatory)	Cluster ID to Type Mapping	Not Applicable
ClusterSpread	array (Mandatory)	Cluster ID to Per Cluster Core Count Mapping	Not Applicable
TotalCoreCount	integer (Mandatory)	Total Number of Cores available.	Not Applicable

Example

```
TargetConfig:
  - TargetName: qli
    ClusterInfo:
      - Id: 0
        Type: big
      - Id: 1
        Type: little
      - Id: 2
        Type: prime
      - Id: 3
        Type: titanium
    ClusterSpread:
      - Id: 0
        NumCores: 4
      - Id: 1
        NumCores: 4
      - Id: 2
        NumCores: 4
      - Id: 3
        NumCores: 4
    TotalCoreCount: 16
```

Systune APIs

This API suite allows you to manage system resource provisioning through tuning requests. You can issue, modify, or withdraw resource tuning requests with specified durations and priorities.

tuneResources

Description: Issues a Resource provisioning (or Tuning) request for a finite or infinite duration.

Function Signature:

```cpp
int64_t tuneResources(int64_t duration,
                      int32_t prio,
                      int32_t numRes,
                      std::vector<Resource*>* res);
```

Parameters:

- `duration` (`int64_t`): Duration in milliseconds for which the Resource(s) should be Provisioned. Use `-1` for an infinite duration.

- `prio` (`int32_t`): Priority level of the request.

- `numRes` (`int32_t`): Number of resources to be tuned as part of the Request.

- `res` (`std::vector<Resource*>*`): Pointer to a list of resources to be provisioned. Details about the resource format are provided below (Refer section "Resource Format").

Returns: `int64_t`

- **A positive unique handle** identifying the issued request (used for future `retune` or `untune` operations)
- `-1` otherwise.

retuneResources

Description: Modifies the duration of an existing Tune request.

Function Signature:

```
int8_t retuneResources(int64_t handle,
                       int64_t duration);
```

Parameters:

- handle (int64_t): Handle of the original request, returned by the call to tuneResources.

- duration (int64_t): New duration in milliseconds. Use -1 for an infinite duration.

Returns: int8_t

- 0 if the request was successfully submitted.
- -1 otherwise.

---

untuneResources

Description: Withdraws a previously issued resource provisioning (or Tune) request.

Function Signature:

```
int8_t untuneResources(int64_t handle);
```

Parameters:

- handle (int64_t): Handle of the original request, returned by the call to tuneResources.

Returns: int8_t

- 0 if the request was successfully submitted.
- -1 otherwise.

---

getprop

Description: Gets a property from the Config Store

Function Signature:

```
int8_t getprop(const char* prop,
               char* buffer,
               size_t buffer_size,
               const char* def_value);
```

Parameters:

- `prop` (`const char*`): Name of the Property to be fetched.
- `buffer` (`char*`): Pointer to a buffer to hold the result, i.e. the property value corresponding to the specified name.
- `buffer_size` (`size_t`): Size of the buffer.
- `def_value` (`const char*`): Value to be written to the buffer in case a property with the specified Name is not found in the Config Store

Returns: `int8_t`

- `0` If the Property was found in the store, and successfully fetched
- `-1` otherwise.

setprop

Description: Modifies an already existing property in the Config Store.

Function Signature:

```
int8_t setprop(const char* prop,
               const char* value);
```

Parameters:

- prop (const char*): Name of the Property to be fetched.
- value (const char*): A buffer holding the new the property value.

Returns: int8_t

- 0 If the Property with the specified name was found in the store, and was updated successfully.
- -1 otherwise.

Resource Format

As part of the tuneResources APIs, the resources (which need to be provisioned) are specified by using a List of Resource structures. The format of the Resource structure is as follows:

```
typedef struct Resource {
    uint32_t OpId;
    uint32_t OpInfo;
    uint32_t OptionalInfo;
    uint16_t NumValues;
    union {
        int32_t Value;
        int32_t *Values;
    };
} Resource;
```

OpId: An unsigned 32-bit unique identifier for the resource. It encodes essential information that is useful in abstracting away the system specific details.
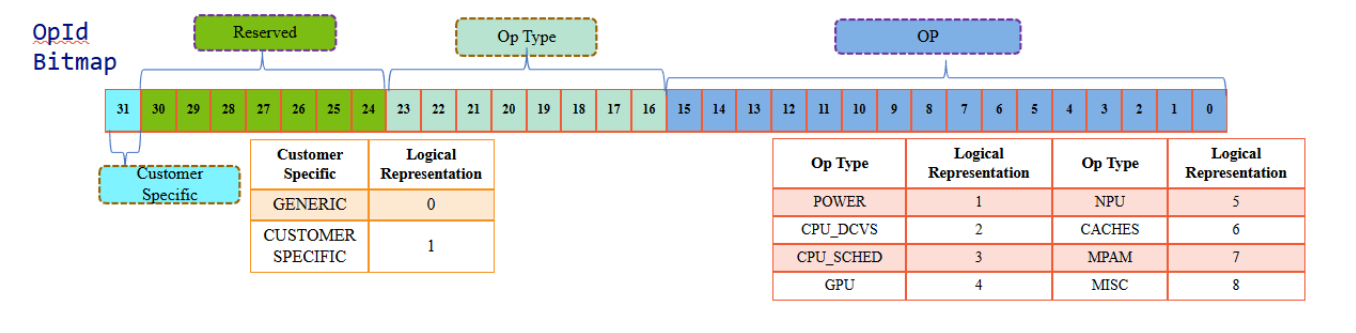
OpInfo: Encodes operation-specific information such as the Logical cluster and core IDs, and MPAM part ID.

OptionalInfo: Additional optional metadata, useful for custom or extended resource configurations.

NumValues: Number of values associated with the resource. If multiple values are needed, this must be set accordingly.

Value / Values: It is a single value when the resource requires a single value or a pointer to an array of values for multi-value configurations.

Notes on Resource Opcode

As mentioned above, the Resource OpCode is an unsigned 32 bit integer. This section describes how this OpCode can be generated. Systune implements a System Independent Layer (SIL) which Provides a Transparent and Consistent way for Indexing Resources. This makes it easy for the Clients to Identify the Resource they want to provision, without needing to worry about Compatability Issues across Targets or about the Order in which the Resources are defined in the YAML files.

Essentially, the Resource Opcode (unsigned 32 bit) is composed of two fields:

- ResID (last 16 bits, 17 - 32)
- ResType (next 8 bits, 9 - 16)
- [Additionally if the BU is providing it's own Custom Resource Config Files, then the MSB must be set to "1", Indicating this is a Custom Resource else it shall be treated as a Default Resource].

These fields can uniquely identify a Resource across targets, hence making the code operating on these Resources interchangable. In Essence, we ensure that the Resource with OpCode "x", refers to the same Tunable Resource across different Targets.

Examples:

- The Resource OpCode: 65536 [00000000 00000001 00000000 00000000], Refers to the Default Resource with ResID 0 and ResType 1.
- The Resource OpCode: 2147549185 [10000000 00000001 00000000 00000001], Refers to the Custom Resource with ResID 1 and ResType 1.

List Of Resource Types (Use this table to get the value of ResType for a Resource)

Name	ResType	Examples
POWER	1	
CPU_DCVS	2	
CPU_SCHED	3	`/proc/sys/kernel/sched_util_clamp_min` `/proc/sys/kernel/sched_util_clamp_max`
CPU_FREQ	4	`/sys/devices/system/cpu/cpufreq/policy<>/scaling_min_freq` `/sys/devices/system/cpu/cpufreq/policy<>/scaling_max_freq`
GPU	5	
NPU	6	
CACHES	7	
MPAM	8	
MISC	9	

Example Usage of Systune APIs

tuneResources

Note the following code snippets showcase the use of Systune APIs. For more in-depth examples refer "link to Examples dir"

This example demonstrates the use of tuneResources API for Resource Provisioning.

```cpp
#include <iostream>
#include <Systune/SystuneAPIs.h>

void sendRequest() {
    // Define resources
    SysResource* resourceList = new SysResource[1];
    resourceList[0].mOpCode = 65536;
    resourceList[0].mNumValues = 1;
    resourceList[0].mConfigValue.singleValue = 980;

    // Issue the Tune Request
    int64_t handle = tuneResources(5000, 0, 1, resourceList);

    if(handle < 0) {
        std::cerr<<"Failed to issue tuning request."<<std::endl;
    } else {
        std::cout<<"Tuning request issued. Handle: "<<handle<<std::endl;
    }
}
```

The memory allocated for the resourceList will be freed by the tuneResources API. The user of this API should not free this memory.

retuneResources

The below example demonstrates the use of the retuneResources API for modifying a Request's duration.

```cpp
void sendRequest() {
    // Modify the duration of a previously issued Tune Request to 20
seconds
    // Let's say we stored the handle returned by the tuneResources API in
    // a variable called "handle". Then the retuneResources API can be
simply called like:
    if(retuneResources(20000, handle) < 0) {
        std::cerr<<"Failed to Send retune request to Systune Server"
<<std::endl;
    }
}
```

untuneResources

The below example demonstrates the use of the untuneResources API for untuning a previously issued
Tune Request.

```cpp
void sendRequest() {
    // Withdraw a Previously issued tuning request
    if(untuneResources(handle) == -1) {
        std::cerr<<"Failed to Send untune request to Systune Server"
<<std::endl;
    }
}
```

Extension Interface

The Systune framework allows business units (BUs) to extend its functionality and customize it to their use-case. Extension Interface essentially provides a series of hooks to the BUs to add their own custom behaviour. This is achieved through a lightweight extension interface using macros. This happens in the initialisation phase before the service is ready for requests.

Specifically the Extension Interface provides the following capabilities:

- Registering custom resource handlers
- Registering Custom Configuration Files (This includes Resource Configs, Signal Configs and Property Configs). This allows, for example the specification of Custom Resources.

Macros

URM_REGISTER_RESOURCE

Registers a custom resource handler with the system. This allows the framework to invoke a user-defined callback when a specific resource opcode is encountered. A function pointer to the callback is to be registered. Now, instead of the normal resource handler, this callback function will be called when a Resource Provisioning Request for this particular resource opcode arrives.

Usage Example

```c
int32_t applyCustomCpuFreqCustom(Resource* res) {
    // Custom logic to apply CPU frequency
    return 0;
}

URM_REGISTER_RESOURCE(0x00010001, applyCustomCpuFreqCustom);
```

URM_REGISTER_CONFIG

Registers a custom configuration YAML file. This enables the BU to provide their own Config Files, i.e. allowing them to provide their Own Custom Resources for Example.

Usage Example

```c
URM_REGISTER_CONFIG(RESOURCE_CONFIG,
"/etc/bin/targetResourceConfigCustom.yaml");
```

The above line of code, will indicate to Systune to Read the Resource Configs from the file "/etc/bin/targetResourceConfigCustom.yaml" instead of the Default File. Note, the BUs must honour the

structure of the YAML files, for them to be read and registered successfully.

Custom Signal Config File can be specified similarly:

Usage Example

```
URM_REGISTER_CONFIG(SIGNALS_CONFIG,
"/etc/bin/targetSignalConfigCustom.yaml");
```

Server CLI

The **Systune Server** runs as a background service, initializing configurations and registering extensions to handle incoming requests.

Commands

- `start` Launches the server, loads all configuration files, and prepares for request handling.

- `exit` Gracefully shuts down the server.

- `dump` Displays all currently active requests in the system.

Client CLI

REDO: NEED TO REWRITE TO MATCH CURRENT IMPLEMENTATION

The **Systune Client** sends tuning-related requests to the server via command-line interface.

Usage Examples

1. Send Tune Requests from YAML File

```
./client_ex -j
```

- Reads requests from `SampleRequests.yaml`. TODO:show example.

2. Send Tune Request via CLI

```
./client_ex -i -v 0:567 -d 5000 -p 1
```

- `-i` : Initiates a tune request
- `-v` : Opcode:Value pairs (comma-separated, no spaces)
- `-d` : Duration in milliseconds
- `-p` : Priority level

3. Send Untune Request

```
./client_ex -u -h 1
```

- `-u` : Untune request

- `-h` : Handle ID

4. Send Retune Request

```
./client_ex -r -h 1 -d 8000
```

- `-r` : Retune request
- `-h` : Handle ID
- `-d` : New duration in milliseconds