SYSLIB Users Guide

Rev 0.4

Texas Instruments, Incorporated
20450 Century Boulevard
Germantown, MD 20874 USA

Document License

Revision Record	
Document Title:	**Software Design Specification**
Revision	Description of Change
0.1	Initial Draft
0.2	Worked on Architecture and Resource management sections
0.3	Worked on Root System and Name Space
0.4	Worked on DAT

Note: Be sure the Revision of this document matches the QRSA record Revision letter.

• Contents

1 References

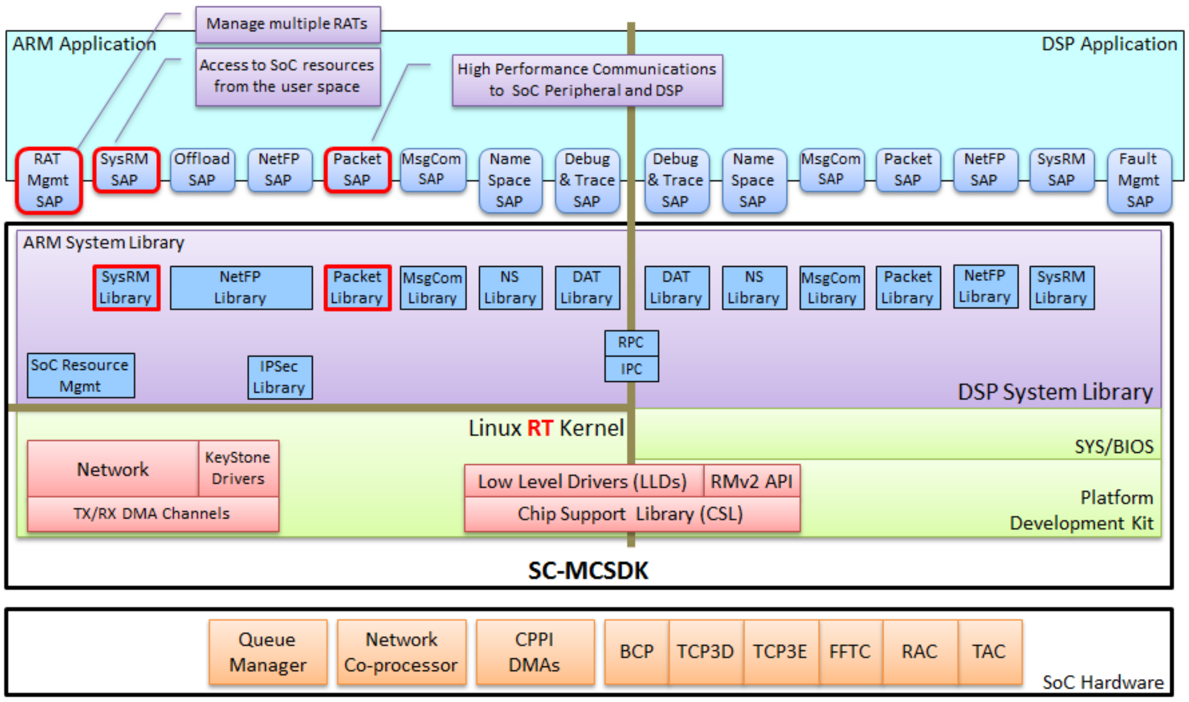
No	Referenced Document	Control Number	Description
1	RMv2 design documentation		MCSDK Documentation
2	RM API documentation		SYSLIB RM DOXY API [resmgr/docs/doxygen] documentation.
3	Name API Documentation		SYSLIB Name DOXY API [name/docs/doxygen] documentation.
4	PKTLIB design documentation		PKTLIB Software Design documentation pktlib/docs/PKTLIB_SDS.pdf

2 Introduction

2.1 Architecture

The SYSLIB package can be seen as a set of high level services available to an Application Software developer via Service Access Points (SAPs). Those SAPs are implemented by different SYSLIB modules with the goal of abstracting lower level hardware details, while at the same time providing all or most of the performance benefits of using the diverse HW acceleration capabilities of TI Keystone SoCs and ensures the application portability for SW migration from one device/architecture to another.

The following figure showcases the complete architecture building blocks which are provided by SYSLIB. In the red we highlighted new services available in SYSLIB 4.x



The SYSLIB services are now uniform between the DSP and ARM. The goal behind the architecture was to ensure that the module API between the DSP and ARM was as consistent as possible. Since there are still few fundamental differences between ARM and DSP, to clearly demarcate those differences, the concept of "**realm**" was introduced and is used throughout the documentation. Realm is an indication to the SYSLIB module whether it is executing in the DSP or ARM domain.

2.2 SYSLIB Services

Below is a brief description of each SAP provided by SYSLIB:

Resource Management SAP

This Service device tree based SoC resource management for both ARM (user space) and DSP applications. The service is facilitated by SYSLIB's Resource Management System.

RAT Management SAP

This Service allows the controlling application (typically, the OAM) to manage multiple instances of logically separate application objects (for example, LTE cell, WCDMA Cell, Radio system, etc). Each of those application objects can be deployed on a set of ARM processes and/or DSP cores. This service is available from ARM realm only, and is facilitated by the SYSLIB's Root System.

Offload SAP

Offload SAP allows the controlling application to provide policy-based control over transport services provided by NetFP SAP. This service is available from ARM realm only, and is facilitated by NetFP Proxy module.

NetFP SAP

This service provides application an access to the accelerated Transport services, including 3GPP specific FastPath management and generic socket levels of APIs.

MSGCOM SAP

This service provides an Application with the ability to manage efficient communication channels that use KeyStone HW capabilities, such as HW queues, DMAs, and Interrupts. Service is available on both ARM and DSP realms, and is facilitated by SYSLIB's msgcom libraries.

PKTLIB SAP

This service provides an Application with the ability to efficiently manipulate packets: allocate, merge/split/clone and free packets. Packets are used to directly communicate with HW accelerators. Service is available on both ARM and DSP realms, and is facilitated by SYSLIB's pktlib libraries.

Name Space SAP

This service provides an Application with mechanism to share the objects across domains and realms. APIs allows for efficient local search and capability to push/get an entry from/to remote data base. Service is available on both ARM and DSP realms, and is facilitated by SYSLIB’s name space libraries.

Debug and Trace SAP

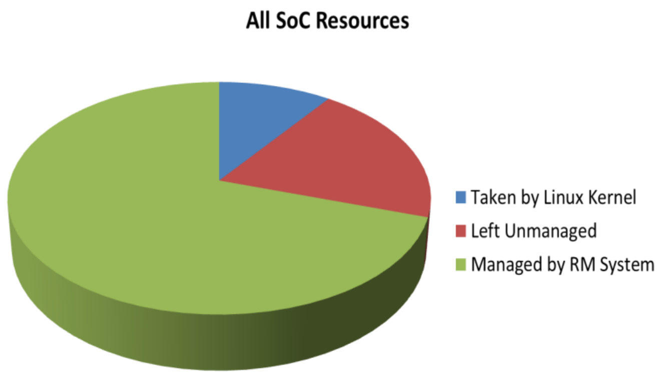
This service provides an Application with coherent mechanism to manage tracing and logging of multiple logical components. Service is available on both ARM and DSP realms, and is facilitated by SYSLIB’s dat libraries.

3 Resource Management System

SYSLIB Resource management system (RM System) provides a method to control SoC resource usage. It allows system integrator to define the scope of the resources that are under RM control, as well as admission policies towards those resources.

3.1 Resources under RM System control

The resources managed by the RM System can be seen as "all SoC resources minus the resources taken by Linux Kernel minus the resources left unmanaged":



The resources given to Linux Kernel are described by multiple kernel dts/dtsi files at compile time. Integrator may need to leave few resources unmanaged (hard coded), for example if the resource is unused, or there is a single component that is solely uses the resource. All the resources managed by RM system (the green ones), are expressed by the integrator in Resource management dts file.

The examples of resources that can be put under RM control are:

- Memory regions
- Hardware Queues
- Accumulator channels
- CPPI Flows and Channels
- NetCP LUTs
- HW Semaphores
- Interrupt controller inputs/outputs
- FFTC Accelerators
- EDMA Engines, etc

Below is an example of the resource description. The resource is represented their value and the number of resources.

```
INTC_QUEUE-qm1 {
            resource-range = <652 6>;
};
```

This indicates that there are 6 INTC (Direct Interrupt) queues starting from 652 onwards.

3.2 Admission Policies

The admission policies (who can take the resource under RM control) are specified in the policy file. The integrator may choose to allow all requesters to obtain the resource in an exclusive use, or to predefine the names of the requesters that will be granted the resource. The former policy will allow for more cooperative resource usage, while the later model may protect one set of application from another.

For example, consider the entry in the policy file:

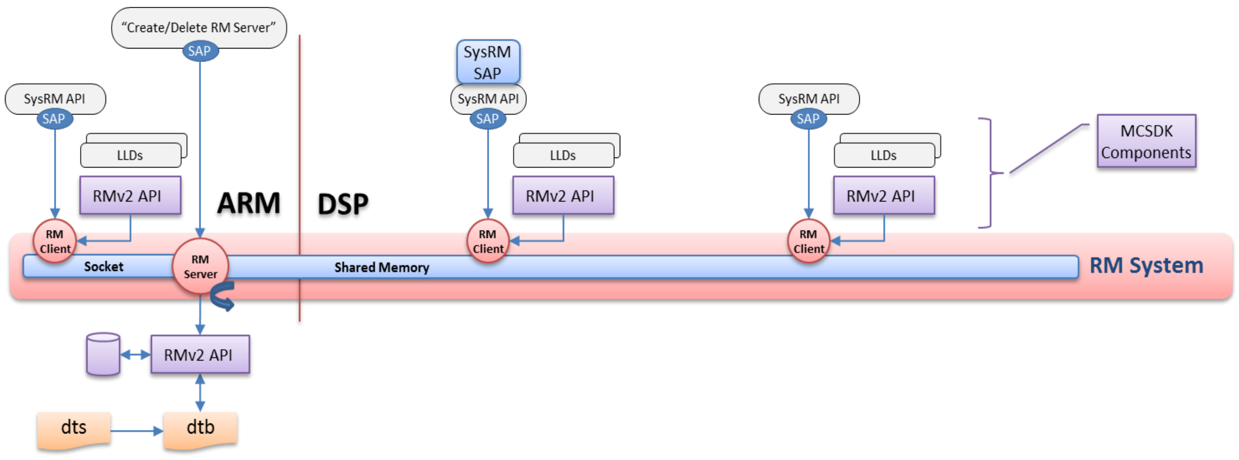
```
INTC_QUEUE-qm1 {
            assignments = <652 6>, "iu = (*)";
};
```

This indicates that the INTC queues from 652 onwards can be used by **any** RM client. Permissions can be modified to grant exclusive usage of a resource or divide it further among a subset of RM clients.

Please refer to the RMv2 MCSDK documentation for more details on the DTS file format specifications.

3.3 Resource Management Architecture

SYSLIB implements the RM System as server/client based



Resource Manager Server
The Resource manager server (RM Server) is a daemon which executes in the ARM User space process. The server maintains a list of all the resources and the allocation rules. Entities communicate with the RM server to request for resource allocation and release.

Resource Manager Clients
The Resource Manager Clients (RM Clients) are libraries linked with the application code and provide a communication pipe between the application and RM server.

3.4 Requesting/Releasing resources

RM Clients provide two mechanisms for resource allocation

- Using RMv2 aware LLDs. MCSDK LLDs: PA/SA/CPPI/QMSS already support resource management APIs. When Application requests a resource using those LLDs (it could be either "any" resource (ex. any general purpose queue), or particular resource (ex. queue 1234), the resource management API will make sure that resources are allocated according to the policies.
- Directly from Application level. For non-LLD managed objects (for example, the FFTC accelerators), integrator may create virtual FFTC channels, request the channel directly from the application, and use the accelerator resources described by this channel directly.

The RM client libraries exist on both the DSP and ARM cores and provide a standardized interface which allows the applications to request and release resources. The RM client libraries however use different transport mechanisms to communicate with the RM server.

- RM clients running on ARM communicate with the RM Server via the native *UNIX socket* mechanism, while
- DSP RM clients communicate with the RM Server using a *shared memory transport*. This mechanism was selected to simplify RM system start up. RM clients populate the RMv2 request in the shared memory transport and then they notify the RM Server via an IPC interrupt.

Please, note, that unlike the previous architecture where the request for all the resources was placed in a well-defined memory section; the current resource manager allows each processing entity (ARM process or DSP core) to request resources as and when needed. Each core is thus responsible **for** handling its own resource requests.

The RM module has also been extended to allow the release of allocated resources on cleanup. Please refer to the resource manager API [2] documentation.

3.5 Resource Management Dependencies

The RM server on **ARM** requires the following kernel modules to be available and loaded:
1. HPLIB Kernel module

The HPLIB Kernel module allows the ability to allocate and map contiguous physical memory to virtual memory and allows the user land applications to map and use the Navigator features.

2. <u>UIO Kernel module</u>
   The UIO kernel module allows the ability for user land applications to receive interrupts. This service is used by the RESMGR server to receive IPC notifications from the DSP cores.

3.6 Integration notes

The section documents some key integration notes which application developers need to account for while using the RESMGR module:

- **Resource Manager Shared Memory Transport address:**
  The RM server (ARM) and RM clients (DSP) use a shared memory transport to communicate with each other. This implies that the shared memory address should be carved out of the DSP DDR3 memory space and be marked as reserved in the platform memory map. This will ensure that no other DSP application can use this memory for additional purposes. This well known address should be configured in all the DSP cores and the RM server. Failure to do so will result in the cores getting stuck while requesting resources.

- **Setting up the SYSLIB RM DTS & DTB files**
  The SYSLIB RM server will need to be passed a list of all the resources which are available to the applications. The SYSLIB RM provides a set of default DTS files in the `resmgr/dts/<device>` directory. Please refer to the <u>getting started section</u> in the document for more details.

- **Modifying the default kernel DTB file**
  The MCSDK released kernel uses a DTS file which is modified by SYSLIB to initialize and setup additional features. Thus each release of SYSLIB releases a set of modified kernel DTS files which located in the `resmgr/dts/<device>` directory. Please refer to the SYSLIB Unit testing document on how to upgrade the kernel DTS file.

  NOTE: This step is a prerequisite; using the default kernel DTB file will lead to unpredictable results.

- **Always execute the RM Server**

Ensure that the resource manager server is executed on the ARM before executing any application (either on the DSP or ARM). The RM server requires the HPLIB kernel module to be loaded.

- **Debug the RM Server**

  The SYSLIB RM Server can display a list of all the resources which have been utilized by sending a User defined signal to the process

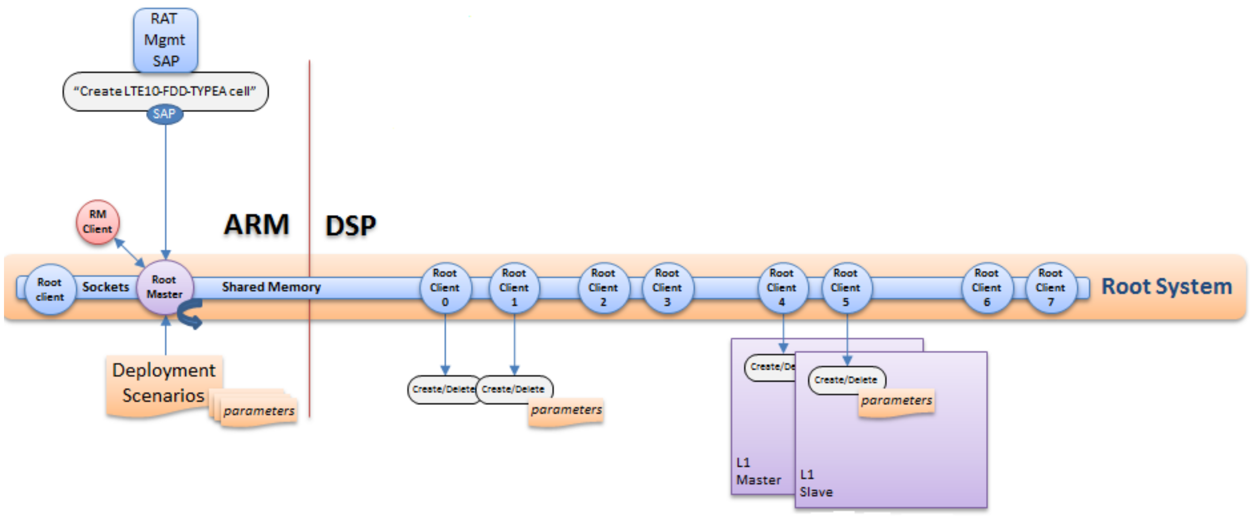
```
kill -10 <SYSLIB RM Server Process Id>
```

4 Root System

The Root System provides a method to manage applications deployments across multiple cores and processes.
For example, the OAM Application may deploy multiple instances of:

- Rel9 LTE Cells,
- Rel10 LTE Cells with multiple component carriers,
- WCDMA Cells,
- Radio System,
- Multi Standard Sniffers, etc.

4.1 Root System Architecture



Root system is built using master/slave concept. There is one and only one Root Master that runs in ARM realm. There can be multiple Root Clients on DSP cores (one per participating core, eg. up to 8 clients for K2H). There can be multiple Root clients on ARM realm (one per participating process).

Root system uses the Shared memory based transport. This is used for exchanging information between the root master and clients. The transport was selected because it needs to be up at the very early stages of application initialization sequence, when no resource management is available yet.

The Root clients/master should **not** consume any CPU cycles once the application has been initialized and is operational. This requires that the root master/clients are blocked on a semaphore. The semaphore is posted once an IPC interrupt is detected.

4.2 Root Master

The Root Master provides an API to OAM software to instantiate applications and to pass the initialization information to a set of Root Clients. The set of Root clients and client specific initialization configuration can be derived from the Deployment Scenarios data base.

The Client specific information includes two types of parameters:
- one describes the platform services required by an application instance (a Syslib configuration),
- while the other is passed transparently to an application domain as run time initialization parameters.

For example, the Syslib configuration passed to L2 core will request instantiation of Name Space, DAT, NetFP, Pktlib, and Msgcom Services, while Syslib configuration passed to L1 core may not include NetFP services.

OAM can register notification functions when deployed application reports a status change.

4.3 Root Clients

Root Clients in each execution realm (ARM or DSP) are initialized and start up and execute a high priority thread. The thread is blocked and the root client is waiting for control information to be passed from the Root Master.

Root System provides an API for Root clients to report their status to the controlling application. In fact, it is expected that after successful instantiation, the application deployed on a given core will report that "it is up".
The Root Client thread shall return back to the blocked state, monitoring the requests from the controlling application. There is a method available for controlling application to request a graceful shutdown.

Essentially, the Root clients provide applications with Enter and Exit points. Applications which utilize the root clients should perform their application initialization in the function `appInit`. Similarly the function `appDeinit` is invoked on application exit and applications should perform application cleanup code here.

4.4 Integration notes

The section documents some key integration notes which application developers need to account for while using the Root module:

- **Root shared memory address**

Please ensure that the root shared memory address is reserved in the DSP memory space and is well known between the DSP root clients and masters. Failure to do so will result in the loss of communication between the root master and DSP root slaves

- **Override "main" in root clients**
  The implementation in `main` should be as follows:-
    - Create a root slave
    - Launch the Root Slave thread of highest priority and execute the root slave API

- **Implement the appInit and appDeinit functions**
  Once the root slave has been initialized/deinitialized the function `appInit/appDeinit` are invoked. This function is called in the context of the highest priority slave thread. It is highly recommended that application perform minimum operations in these functions and instead launch a task to perform the rest of the operations. This is because the root slave thread is a control thread to communicate with the root master and if the thread dies the connectivity between the root master/slave is dead.

5 Name Space

Name Space System provides a mechanism to share the objects across multiple applications running on different ARM processes or DSP cores. The object (the Name Space *entry*) includes an ASCII Name and few parameters associated with it. The entries are stored in a Name Space database. Application can instantiate multiple databases. The entry name shall be unique within a database.

Name space system design is modular and allows for simple private name spaces (single core/single process), as well name spaces shared across multiple cores or ARM processes. The design had focused on search efficiency, where the search is always local and fast. The library provides an API to create/delete/find/modify local entries. The library also provides optional mechanisms for pushing the entry into remote data base and getting the entry from the remote database. Name space system consists of the following components:

- Name Space Library
- Name Space Database
- Name Space Proxy, and
- Name Space Client

5.1 Name Space Library

Name Space Library implements the Name Space API, and provides applications with the methods to create/delete modular Name Space components, methods to manage local database entries and to push/get the entries form the remote databases. Library is available to both ARM and DSP realms.

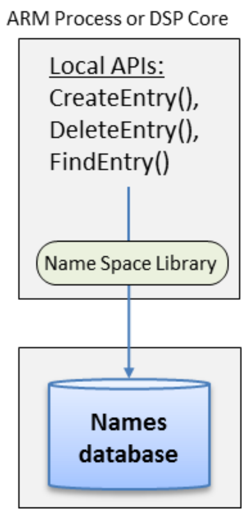
Please refer to the Name Space API documentation for more details.

5.2 Name Space Database

Name Space Library provides an API to create/delete the Name Space database. Application can create private databases, DSP multicore shared databases, and ARM multi-process shared databases.

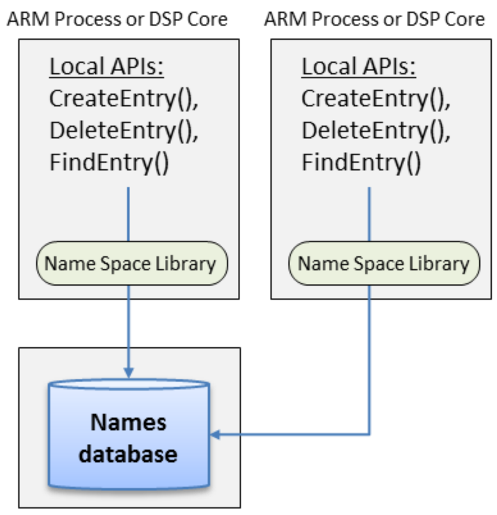
One database, `Name_ResourceBucket_INTERNAL_SYSLIB`, is used by the SYSLIB modules for their own internal purposes.

The simplest database is shown below:



Application creates a database and uses local APIs to create/delete/find entries. This is typical for single process/multithreaded applications, or DSP single core/multitasking applications.

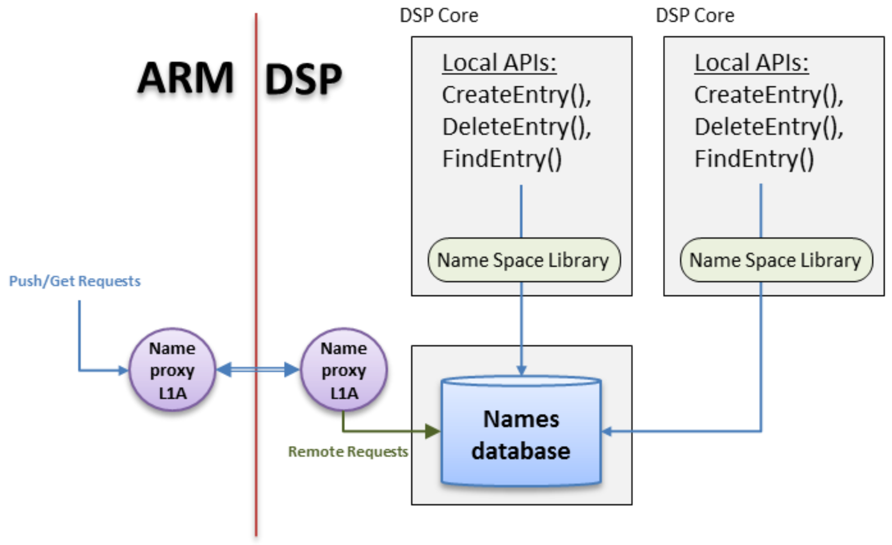
The next picture shows the database shared across multiple DSP cores or multiple ARM processes:



Application creates a multicore or multi-process save database on one process/core, and allows for efficient local API to be called from any of the processes/cores that got the handle to a shared database.

5.3 Name Space Proxy

When Application desires to allow "remote" (i.e. from another realm) access to the database, it creates a Name Space proxy:

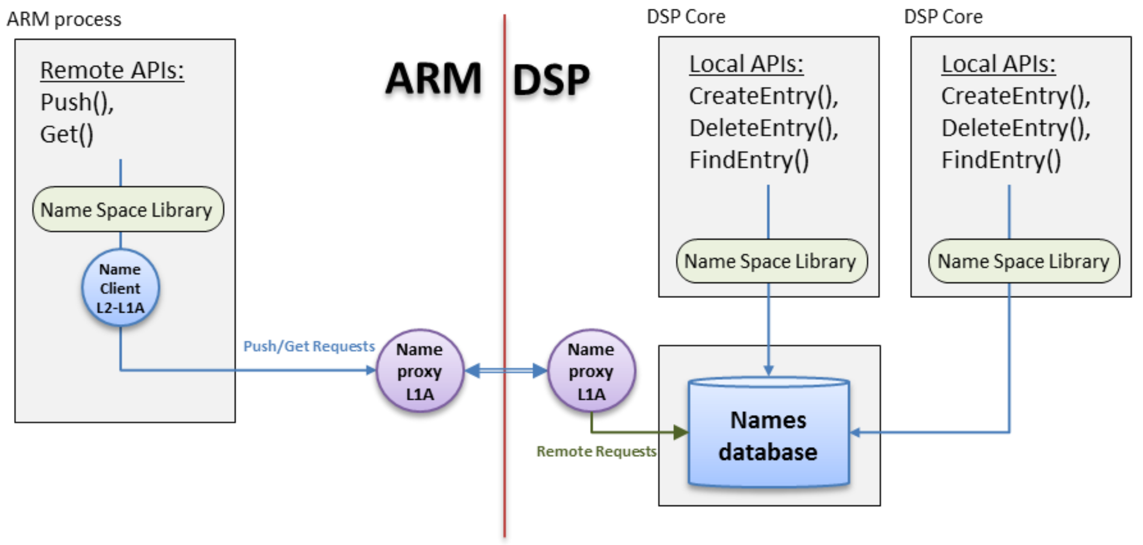


The picture above still only allows the local APIs on DSP cores to be used to manage the database entries; however with addition of Name Space Proxy objects we are allowing ARM side Application to either push and entry to the DSP side, or get an entry from the DSP side. Note that there is always one pair of Name Proxies per database. Also, both Name proxy objects are part of the DSP domain, meaning they are created/deleted when the Name Space service associated with the DSP database is created/deleted.

Name Proxies communicate with each other using the Navigator Infrastructure DMA channels. For ARM, SYSLIB release provides a Name Space Proxy application in the "apps" folder. Based on deployment configuration, the proxies will be launched during the Syslib init time. On DSP, name proxy needs to be created and an execution context needs to be provided. Please refer to the Name Proxy API documentation for more details.

5.4 Name Space Client

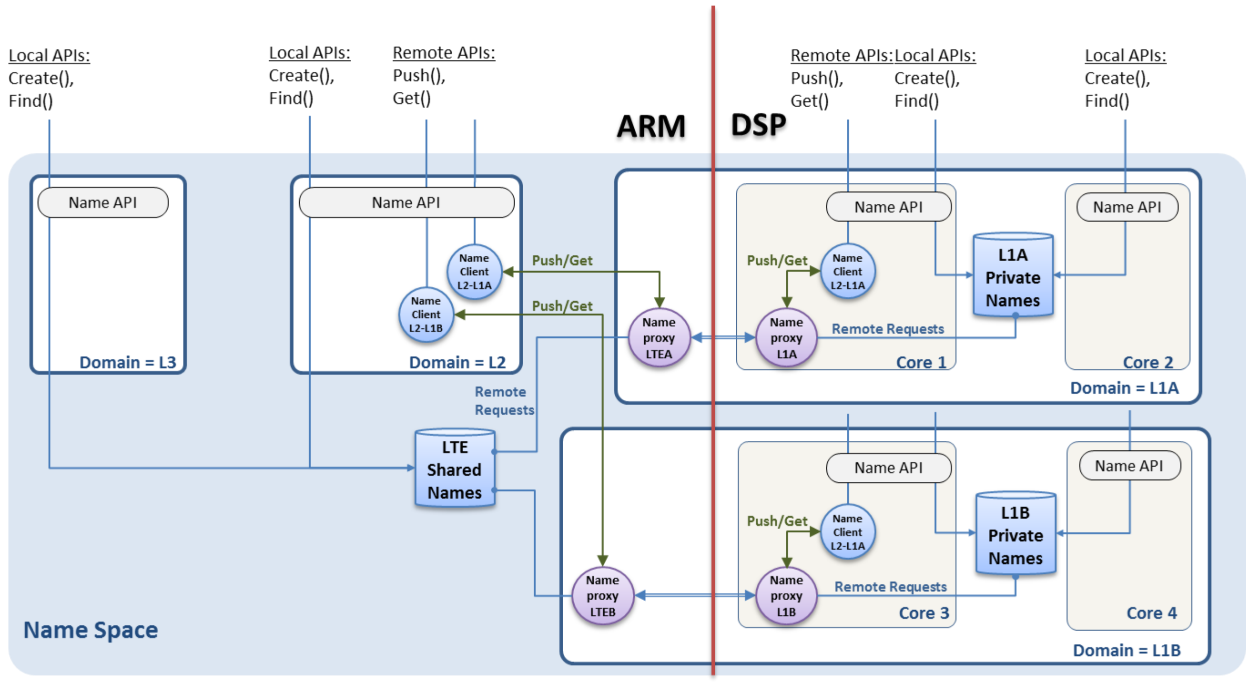
To access the "remote" database the Name Space Clients are created.



There can exist multiple name clients which attach themselves to a Name proxy; these clients send their requests to the name proxy which proxies their request to the remote realm for execution.

5.5 Implementation Example

The example below shows the Name Space system built for LTE Rel10 cell, where L1 component carriers are running on two DSP cores each, L3 and L2 applications are running on ARM separate ARM processes.



This example illustrates one shared (between L3 and L2 application processes) Name Space database (LTE Shared Names) instantiated on ARM realm. The term “shared” means that the handle to the database is known to both processes, and proper multi-process protection is being enforced. Each LTE component carrier Application that creates a database that is shared between the two DSP cores. In addition, each carrier allows remote access to the database, i.e. the L2 entity can push and get the entries form L1 databases (for that purpose, L1 Application creates set of Name Proxies. L2 Application also creates two Name Space clients, which allows the application to push and get the entries from the DSP side.

5.6 Integration notes

The section documents some key integration notes which application developers need to account for while using the Name Proxy module:

- **Name Database Address:**
  In the DSP realm the named resource database needs to be shared across all the DSP cores using the same name resource instance. It is recommended that this address be reserved in the platform memory map. This will ensure that no other DSP application can use this memory for additional purposes. This well known address should then be used across all the DSP cores. Failure to do so will result in the named resource module to fail across the DSP cores.

- **Hardware semaphore protection**
  Since the implementation of the database is a shared memory across multiple cores; the name database needs to be protected using a hardware semaphore. Please ensure that the hardware semaphore is the same across all cores.

- **Provisioning the Local & Remote Flow Id for the name proxy**

  The Name proxy communicates with its peer using the Navigator Infrastructure DMA channels. For this to operate properly the local & remote flow identifiers between the Name proxy peers needs to be configured correctly. Failure to do so will result in the peers unable to communicate.

- **Execution context for the name proxy on DSP**

  The name proxy requires an execution context which can be executed in a polled mode. Without an execution context any remote database API would hang indefinitely.

6 PKTLIB

PKTLIB is a library which is provided by SYSLIB which abstracts the TI Navigator data structures from the application providing a more generic data structure which is easily understood and used by application developers.

The PKTLIB module now allows multiple instances to be created. Heaps are created and registered with a specific PKTLIB instance.

6.1 Heaps

The PKTLIB module allows an application to create/delete heaps. The concept of heaps is similar to the heaps provided by a standard operating system i.e. a memory allocator from which a block of memory can be allocated or cleaned up. PKTLIB heaps are similar except they allow the allocation and cleanup of descriptor (TI Packets).

Each heap is associated with a unique name. The PKTLIB module using the named resource services to maintain the heap database. The following are the different heap types which are supported:-

- <u>Shared Heap</u>

  These heaps are visible across multiple cores and should only be used if packets from the heap are allocated on one core and are passed to another core where they could be cleaned up.

  a. Heaps can **<u>only</u>** be shared between the cores in the DSP realm
  b. Heaps cannot be shared between the ARM and DSP realm.
  c. Heaps cannot be shared between different ARM processes.

- <u>Local Heap</u>

  These heaps are visible only to a single core. This heap should be used if the packets are allocated and consumed on a single core itself.

- <u>Super Heap</u>

  These heaps combine multiple heaps and provide better memory usage. Multiple different size heaps can be combined together under a single super heap. Super heaps are again specific to a core. It is recommended that all the member heaps be of the same type (Shared vs. Local)

6.2 Heap Sizing

While integrating and creating heaps in the system sizing the heaps is an important consideration. The size of the data buffers is simple because in most cases it is known upfront the max size of the data which will be received or sent. However the number of packets to be present in each heap is difficult to estimate.

The heap configuration introduces a concept called "Threshold". Please review the PKTLIB Software Design Documentation [3] for more information. The concept basically allows during heap creation a specific "threshold" to be configured. Each time the queue depth drops below the specified threshold the Navigator records this. This counter can be retrieved through the PKTLIB get heap statistics API

Application developers can use this feature by setting different values of packets in the heap and by configuring the thresholds to different values. The data retrieved from these experiments can then be used to determine the size of the heap.

6.3 Garbage Collection

PKTLIB exposes a concept called "Garbage Collection". This concept is applicable under the following scenarios:

- The application uses the Clone Packet and/or Split Packet API
- Cloned and/or Split packets are being passed to an IP block for transmission

If the application is performing the above mentioned steps they will need to ensure that they call the Garbage Collection. Please read the PKTLIB Software Design Document [3] for more information.

The Garbage collection function needs to be executed else the heap will be exhausted and the allocations will fail. The periodicity depends upon a variety of factors such as the frequency of allocations and the size of the heap. The garbage collection needs to be executed on the following heaps:-

1. Heap from where the zero data buffer packet was allocated
2. Heap from where the actual data packet (which is being split/cloned) was allocated

The following pseudo code shows the usage:-

```
    /* Allocate data packet */
    ptrOrigPkt = Pktlib_allocPacket(appPktlibInstHandle, myHeap, 100);
    if (ptrOrigPkt == NULL)
        return -1;

   /* Populate the data packet. */
   ...
```

```
    /* Allocate Zero Buffer Packet */
    ptrClonePkt = Pktlib_allocPacket(appPktlibInstHandle, myHeap1, 0);
    if (ptrClonePkt == NULL)
        return -1;

    /* Now clone the original packet. */
    if (Pktlib_clonePacket(appPktlibInstHandle, ptrOrigPkt, ptrClonePkt) < 0)
        return -1;


    /* Pass the Clone packet to the hardware IP block. */
    ...

    /* Execute the Garbage Collection */
    Pktlib_garbageCollection(appPktlibInstHandle, myHeap);
    Pktlib_garbageCollection(appPktlibInstHandle, myHeap1);
```

There are 2 heaps which are being used `myHeap` and `myHeap1`. The data packets are allocated from `myHeap` while the zero buffer packets are allocated from `myHeap1`.

The cloned or original packet could be passed to an IP block for CPDMA transmission. Depending upon the application use case this can be done in either of the following ways:

1. Application pushes the packet to a CPDMA hardware queue.
2. Application uses the NETFP module to send the packet to the network
3. Application use the MSGCOM Queue-DMA to send the packet to another entity

In either of the above cases applications are responsible for executing the garbage collection to ensure that the packets are recycled from the garbage queue back into the free queue.

6.4 Heap Deletion

The PKTLIB provides an ability to delete the heap. However there are certain rules which need to be followed while calling this API:-

- The application is responsible for ensuring that the heap handle being deleted is NOT being used to invoke other PKTLIB API. This is because there is no runtime checking support added in PKTLIB API to determine if the heap is valid or not since this will cause a performance penalty.
- If a super heap is being deleted the super heap is removed but the member heaps which comprise the super heap are left intact.
- If local and shared heaps are being deleted; then all data & zero buffer packets specified during heap creation be present and if this is not the case the heap deletion will fail with an appropriate error code. This will ensure that there are no missing descriptors & memory leaks.

6.5 Integration notes

The section documents some key integration notes which application developers need to account for while using the PKTLIB module:

1. **Use local heaps and avoid using shared heaps**

   Shared heaps should be avoided as much as possible. This is for the following reasons:

   - **Garbage Collection**

     Garbage collection on shared heaps can cause CPDMA lockups. This is because garbage collection modifies the reference counters inside the packets and this can cause the CPDMA if the packet is being transmitted when the modification is done.

   - **Handling heap deletion**

     There is no run time checking done in the PKTLIB API to validate if the heap is active or not (for performance). For shared heaps if the entity which created the heap; deletes it then it can cause another entity which uses the shared heap to continue using it which could lead to intermittent failures.

2. **Descriptor Leakage**

   During initial development it is possible that packets from a local heap might inadvertently leak to another core. This could be because of a programming error. It is highly recommended that application use the following checking in the following OSAL call table:-

```
void Pktlib_osalBeginPktAccess
(
Pktlib_HeapHandle heapHandle,
Ti_Pkt*           ptrPkt,
uint32_t          size
)
{
#ifdef __DEBUG__
     if ((heapHandle != myHeap) &&
         (heapHandle != myHeap1))
    {
        /* FATAL ERROR: Leaked descriptor. */
    }
#endif
   // Invalidate the cache
   ...
}

void Pktlib_osalEndPktAccess
(
Pktlib_HeapHandle heapHandle,
Ti_Pkt*           ptrPkt,
uint32_t          size
)
{
#ifdef __DEBUG__
     if ((heapHandle != myHeap) &&
         (heapHandle != myHeap1))
    {
        /* FATAL ERROR: Leaked descriptor. */
    }
#endif
   // Writeback the cache
   ...
}
```

3. **Descriptor Size**

   The PKTLIB module requires 16 bytes of “module private information” to be stored at the end of the descriptor. Application developers should account for this while allocating descriptor sizes. Please ensure that this memory is not overwritten else this will result in descriptor leakage.

4. **ARM Data buffer allocations:**

   Please ensure that the PKTLIB data buffer allocation implementations on ARM use the HPLIB memory allocation functions. PKTLIB data buffers are used by the hardware to DMA data; HPLIB allocated data buffers are accessible by the DMA.

7 MSGCOM

MSGCOM is an IPC library which is provided by SYSLIB which allows message communication between the DSP cores and also between the ARM and DSP. Please refer to the MSGCOM Software design document for more information.

In order to include MSGCOM into an application please ensure the following line is added to the application RTSC configuration file:-

```
var Msgcom = xdc.useModule('ti.runtime.msgcom.Settings');
```

7.1 Channels

MSGCOM introduces a concept called “channels” which are endpoints identified with a unique name which can be used to send and receive messages. Channels are unidirectional which are referred to as “reader channels” if the channel is capable of receiving messages. “Writer channels” are channels which are only capable of sending messages.

Reader channels are also associated with a blocking mode. The mode implies the operation of reader channel when there is no message available.

The MSGCOM module uses the named resource module to store channel name and associated information.

7.2 Select Functionality

In certain cases applications would need to wait on multiple channels for a message to arrive. This can be implementing by performing the following:-

1. Ensure that the channel is created in “non-blocking” mode but the interrupt mode is either **accumulated or direct interrupt**
2. Register a callback function while creating the channel. The callback function once invoked will post a common semaphore or event
3. The application will be waiting on this common semaphore.

The following code snapshot explains how this done:

```
/* Populate the channel configuration. */
cfg.mode                        = Msgcom_ChannelMode_NON_BLOCKING;
cfg.appCallBack                 = myCallback1;
cfg.u.queueCfg.interruptMode    = Msgcom_QueueInterruptMode_DIRECT_INTERRUPT;
...
/* Create the Message communicator channel. */
ch1 = Msgcom_create ("C1", &cfg, ...);

/* Populate the channel configuration. */
cfg.mode                        = Msgcom_ChannelMode_NON_BLOCKING;
cfg.appCallBack                 = myCallback2;
cfg.u.queueCfg.interruptMode    = Msgcom_QueueInterruptMode_DIRECT_INTERRUPT;

/* Create the Message communicator channel. */
ch2 = Msgcom_create ("C2", &cfg, ...);
...
```

There are 2 channels ch1 and ch2 created with direct interrupt support and there are 2 call back functions registered. The callback functions will be invoked as soon as an interrupt is detected. The below pseudo code is a sample implementation of these functions:-

```
static void myCallback1 (MsgCom_ChHandle chHandle)
{
    Event_post(myEventObj, Event_Id_00);
    return;
}
static void myCallback2 (MsgCom_ChHandle chHandle)
{
    Event_post(myEventObj, Event_Id_01);
    return;
}
```

The callback functions use the OS notification semantics. In the example above; the BIOS Event object is used to post an event. This could also be replaced with a semaphore post. The application thread in the meantime would have the following pseudo code:

```
/* Wait for an event to occur in the system. */
events = Event_pend(myEventObj,
                    Event_Id_NONE,
                    Event_Id_00 + Event_Id_01,
                    BIOS_WAIT_FOREVER);

/* Process the events: */
if (events & Event_Id_00)
{
     while (1)
     {
         /* Process all messages on Channel 1 */
         Msgcom_getMessage (ch1, (MsgCom_Buffer**)&ptrMsg);
         ...
     }
}

/* Process the events: */
if (events & Event_Id_01)
{
     while (1)
     {
         /* Process all messages on Channel 2 */
         Msgcom_getMessage (ch2, (MsgCom_Buffer**)&ptrMsg);
         ...
     }
}
```

The application thread is pending on the event object or semaphore and will be woken up as soon as a message is received on one of the channels. Once the thread is woken up it can read all the messages from the channel.

7.3 Integration Notes

The section documents some key integration notes which application developers need to account for while using the MSGCOM module:

1. **Reader & Writer Channel are unique**

   In the *older architecture*; the reader & writer channels shared the same memory block. This is no longer the case. Internally a MSGCOM reader channel only has the "reader" part of the data structure populated while the writer channel block will have only the "writer" part of the data structure populated. This implies that using a reader channel to send a message **OR** using a writer channel to receive a message is an error.

2. **Writer Channel need to be deleted**

   In the older architecture a writer channel should NOT be deleted. Only a reader channel was required to be deleted. This is no longer the case and now it is required to delete both the reader & writer channel.

3. **Finding a channel creates a writer channel**

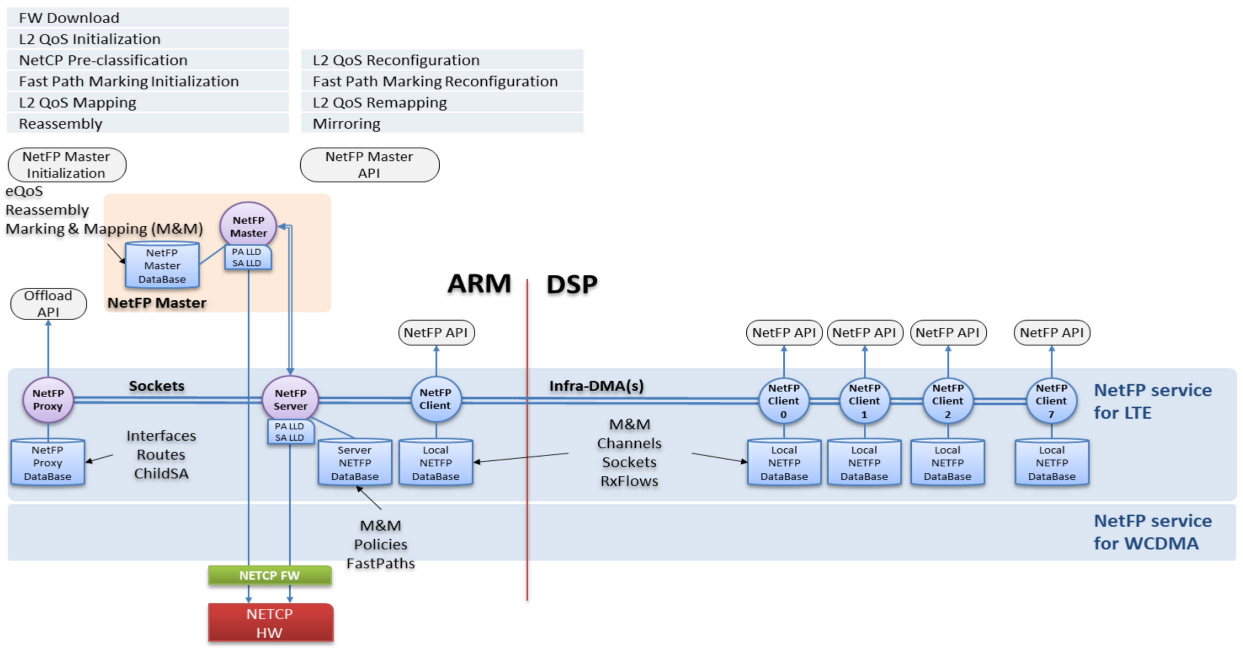
   In the older architecture `Msgcom_find` could be done any number of times. This is no longer the case and doing a find would now cause a MSGCOM writer channel to be created and memory to be allocated. Please be aware that this could lead to memory leaks.

8 NETFP

The NETFP (Network Fast Path) module was designed to provide application developers with high level API abstractions to use HW accelerators (Network Coprocessor (NETCP) and traffic shapers) to perform transport and 3GPP specific functions:

- Efficient communication with network core over UDP/GTPU packets
- Fully offloaded IPSec processing
- 3GPP Air ciphering
- Accelerated Reassembly and Fragmentation
- HW egress shaping
- Packet forwarding

NETFP module system architecture is presented on a picture below.



There is one and only one NetFP Master and there can be multiple (per RAT) NetFP Services.

8.1 NetFP Master

NetFP Master performs tasks that are common for all RATs.

NetFP Master functionality includes:

1. One-time NETCP Initializations
    a. Firmware download,
    b. L2 QoS initialization
    c. NetCP Pre-classification initialization,
    d. Fast Path Marking initialization (per physical interface inner to outer DSCP/pBit translation)
    e. L2 QoS Mapping (per interface mapping of outer DSCP/pBit to a QoS channel/flow)
    f. Reassembly process initialization
2. Provides set of messaging APIs for NetFP reconfiguration
3. Registration service for multiple NetFP servers
4. Physical Interface information exchange with NetFP servers

8.2 NetFP Service

Each NetFP service includes one NetFP proxy, one NetFP Server, and multiple NetFP clients. NetFP service provides following functionality

- Efficient communication with network core over UDP/GTPU packets,

- Fully offloaded IPSec processing, and
- 3GPP Air ciphering

8.2.1 NetFP proxy

NetFP proxy provides an API for policy offloading.
It monitors networking events and events from IPSec manager. It notifies NetFP server on relevant changes.

8.2.2 NetFP server

NetFP server maintains the marking and QoS mapping database, Security Policy, and FastPaths database.
NetFP Server provides Clients with relevant egress information after route resolution.
NetFP Server notifies Clients on relevant change events.

8.2.3 NetFP client

NetFP client is a library that provides an application with an APIs for fast path and channel/socket management, notifies sockets on modifications, and collects relevant statistics.

The NETFP module provides the following services:

1. **NETFP server**
   The NETFP server is a standard daemon (provided as a part of the `apps` folder) which executes in the context of the ARM realm. The NETFP server is responsible for the following services:
   - Configuration of the NETCP subsystem (PA and SA)
   - Responsible for the management of any entity which could be shared among the multiple NETFP clients.
     - Networking Interface
     - IPv4 & IPv6 Routing Table
     - Secure & Non Secure Fast Paths
     - Security Policy and IPSEC security channel configuration
     - Air Ciphering security channel configuration
   - The NETFP Server is configurable using a configuration file. Please refer to an example of the configuration file format in the `ti/apps/netfp_server` folder.

2. **NETFP Client Libraries**
   The NETFP clients execute in the context of the application and provide a well-defined interface which allows the clients to communicate with the NETFP server for configuration. The NETFP client is also responsible for the management of entities which are client specific and cannot be shared across clients:
   - NETFP Sockets
   - NETFP 3GPP Channels
   - NETFP Reassembly

NETFP clients executing on the DSP core communicate with the NETFP server via MSGCOM Queue DMA channels. For the NETFP clients executing on the DSP; since the NETFP Server executes in the ARM realm a name client & proxy is also required to exchange information with the server. This is not required for the ARM NETFP clients.

8.3 Integration notes

The section documents some key integration notes which application developers need to account for while using the NETFP module:

- **NETFP Client Thread**
  Each NETFP client module needs to be provided an execution context. This is the context in which the NETFP client responses will be processed. Failure to create a NETFP client thread will result in an NETFP client call getting blocked forever.

- **NETFP Reassembly Service**
  There should be 1 NETFP client which has been registered to handle the reassembly services. Please ensure that the Reassembly servicing function is invoked and the timeout function is also invoked periodically. Failure to do will result in the reassembly operations to not work properly. The Reassembly (Servicing and Timeout) API should always be executed in the same context. Due to performance considerations the API do not use any critical sections.

  Current version of packet accelerator (PA) module, inside network coprocessor (NETCP) hardware accelerator, does not support IP reassembly, so all the IP fragments are detected and forwarded to and reassembled at host. This reassembly service is considered as software extension of the PA. Integrator needs to guarantee CPU cycles for the reassembly operations so that there are no drops of packets between PA and reassembly service. Dropped packet will result mismatch of packet count inside PA and assigned traffic flow will not be freed, causing every (fragmented or not) packet to go through reassembly service permanently.

9 DAT

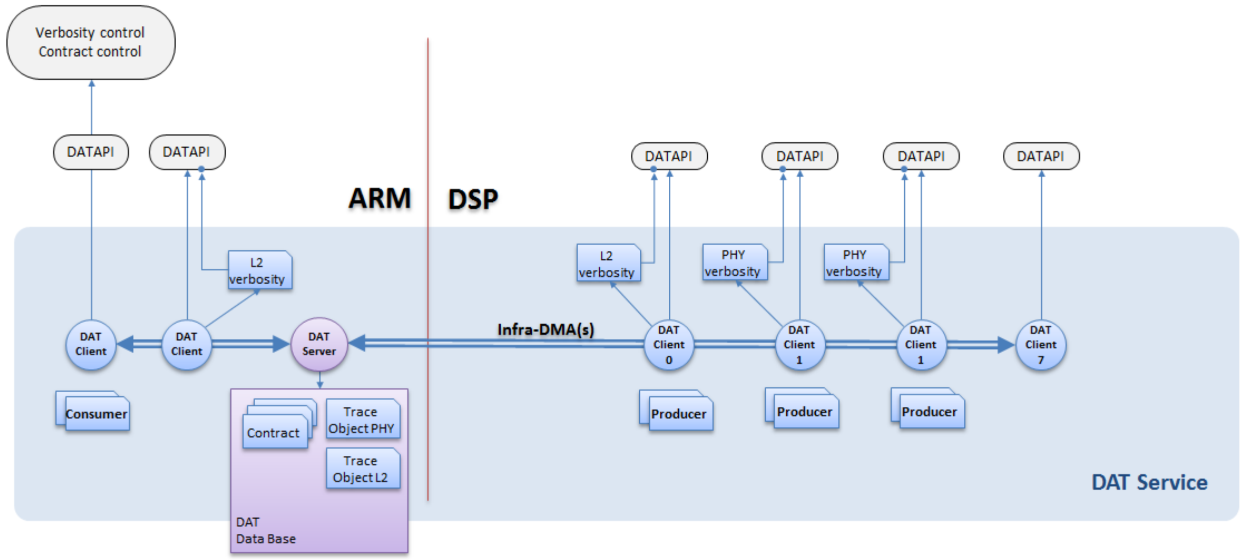
The DAT (Debug and Trace) module was designed to provide application developers with APIs to have coherent debug and trace system on both DSP and ARM. It utilizes TI UIA (Unified Instrumentation Architecture) and cUIA (C-style UIA) modules to provide efficient event or memory snapshot logging.

Trace Verbosity control of DAT modules gives the application an unified way to control log levels from both ARM and DSP.

In summary, DAT module supports the following features:

- Logging through UIA/cUIA, log buffers are managed by Producer and Consumer
- Log shipment to Network/Memory/Consumer
- Trace verbosity control to set log levels

The following figure showcases the NETFP system architecture:



The DAT module provides the following services:

1. **DAT server**

   The DAT server is a standard daemon (provided as a part of the `apps` folder) which executes in the context of the ARM realm. The DAT server communicates with DAT clients and is responsible for the following services:
   - Responsible for the management of producers and consumer from DAT clients
   - Responsible for relaying the actions such as connect/disconnect DAT consumers, trace verbosity modification
   - Responsible for the management of trace objects from DAT clients
   - It maintains the Trace Object database and is responsible for queries from DAT clients.
   - The DAT Server is configurable using a configuration file. Please refer to an example of the configuration file format in the `ti/apps/dat_server` folder.

2. **DAT Client Libraries**

The DAT clients execute in the context of the application and provide a well-defined interface which allows the clients to communicate with DAT server and other DAT clients through DAT server for the following activities:

- Producer/Consumer creation and deletion
- Consumer connect/disconnect
- Trace Object creation/deletion
- Trace verbosity query/modification
- Producer Memory logging
- UIA/cUIA loggerStreamer instance creation/deletion/management
- UIA/cUIA LogSync events management

DAT clients executing on both DSP and ARM cores communicate with the DAT server via MSGCOM Queue DMA channels.

9.1 DAT Entities

9.1.1 DAT Producer

DAT producers are entities created to generate logs and manage log buffers. Producer can ship the log buffers through following ways:

- To Network through NETFP servers
- To Memory through Memory logging (MSGCOM Queue DMA channels)
- To Consumers through MSGCOM Queue DMA channels

The selection for Network and/or Memory is done at producer creation time.

9.1.2 DAT Consumer

DAT consumer is responsible for receiving the log from producer and passes it application.

9.1.3 DAT Trace Object

DAT Trace Object is an entity to maintain trace verbosity levels for components in the trace object. It is created on a DAT client and trace verbosity levels database is maintained on DAT server. DAT server handles queries for verbosity levels. Trace verbosity level modification request is also handled by DAT server and pushed to all the DAT clients that created instances for the trace object.

9.2 UIA/cUIA Log Analysis

Logs generated through UIA/cUIA events can be analyzed through System Analyzer in CCS. Please refer to Appendix A for details of creating UIA config file.

10 UINTC

The UINTC (User space interrupt controller) module is only applicable for the ARM. The UINTC module allows the applications to receive interrupts in their user land applications. The UINTC module uses the services provided by the UIO kernel module.

The SYSLIB modifies the default MCSDK kernel DTS file (as explained above in the RESMGR section). One of the modification's which have been done is the registration of the interrupts for Direct Interrupt queues with the kernel UIO module.

The UINTC module thus provides the following features:-

1. **Registering & Deregistering ISR**
   The UINTC module allows applications to register their ISR function to be invoked on the reception of an interrupt. The module allows application to deregister their ISR.

2. **Enabling/Disabling interrupts**
   The UINTC module allows application to enable/disable registered interrupts.

3. **Flexible execution mode**
   The UINTC module allows application to handle interrupts in multiple methods:
   - **UINTC managed Mode**
     In this mode; the UINTC executes as a thread of highest priority in the system. Once an interrupt is raised; the UINTC thread is woken up which decodes the interrupt and invokes the application register ISR function. The mode is easy to use but comes at the cost of an additional task switch.

   - **Application Managed Mode**
     In this mode; the application is responsible for waiting for interrupts to be invoked in the system. Once an interrupt is registered; the UINTC returns a file descriptor. Applications can use the descriptor and perform a select in their context to wait for interrupts. The mode is slightly more complicated to use but can increase performance since it reduces task switching

10.1 Integration notes

The section documents some key integration notes which application developers need to account for while using the UINTC module:

- **Ensure the kernel DTS file is upgraded**
  Please ensure that the kernel DTS file is upgraded as specified in the RESMGR section. Failure to do so will result in the kernel UIO module failing to register interrupts and the UIO ISR registration will fail.

- **Application managed mode: Perform a read else select will not block**
  The UIO kernel module requires a read to be performed on the file descriptor (returned after the ISR has been registered). Failure to do so will result in the select to never block. The read is an acknowledgment to the kernel module that the interrupt has been acknowledged by the user space application.

- **Enabling interrupt**
  Once an interrupt is detected; the kernel UIO module will disable the interrupt. Application needs to enable the interrupt once all the interrupts have been processed. **NOTE:** In the case of MSGCOM channels the MSGCOM module enables the interrupt once it detects that the queue is empty. However if the UINTC module is used for other interrupt sources this behavior should be understood by the application. Failure to account for this will result in only a single interrupt being detected.

10.2 Using UINTC module in Application Managed mode with interrupt driven MSGCOM channels

Below is a pseudo code to exemplify the usage of Syslib UINTC module in Application Managed mode.

10.2.1 Create additional UINTC module instances

In addition to UINTC managed mode, create an instance of UINTC in Application Managed mode:

```
    uintcConfig.mode = Uintc_Mode_APPLICATION_MANAGED;
    gAppManagedHandle = Uintc_init (&uintcConfig, &errCode);

    uintcConfig.mode = Uintc_Mode_UINTC_MANAGED;
    gUintcManagedHandle = Uintc_init (&uintcConfig, &errCode);
```

10.2.2 ISR registration in MSGCOM OSAL

Track all the fd's which need to be selected directly by application. You can use the channel name to determine which channel was being registered. Hook them into the appropriate UINTC module.

```
int32_t Msgcom_osalRegisterIsr
(
    const char*          channelName,
    Qmss_Queue           queueInfo,
    MsgCom_Isr           isr,
    MsgCom_ChHandle      chHandle,
    MsgCom_Interrupt*    ptrInterruptInfo
)
{
   ...
   if (strcmp (channelName, "SFI_Channel1") == 0)
   {
     fd = Uintc_registerIsr(gAppManagedHandle, ptrInterruptInfo->hostInterrupt,
  (UintcIsrHandler)isr, chHandle, &errCode);
      /* Track all the UINTC file descriptors which are being registered */
      gApplicationISRTracker[0] = fd;
   }
   else if (strcmp (channelName, "SFI_Channel2") == 0)
   {
     fd = Uintc_registerIsr(gAppManagedHandle, ptrInterruptInfo->hostInterrupt,
(UintcIsrHandler)isr, chHandle, &errCode);
      /* Track all the UINTC file descriptors which are being registered */
      gApplicationISRTracker[1] = fd;
   }
   else
   {
     fd = Uintc_registerIsr(gUintcManagedHandle,
ptrInterruptInfo->hostInterrupt, (UintcIsrHandler)isr, chHandle, &errCode);
   }
   ...
}
```

This shows that channels name SFI_Channel1 and SFI_Channel2 are hooked up in the Application managed mode. All other channels (DAT Client, NETFP Clients etc.) are hooked in the UINTC managed mode.

10.2.3 File Descriptor initialization and blocking on Select

For the application managed mode; we will now have two threads, each of those threads is blocked waiting on the appropriate channels.

```
/* thread SCHED-RR/FIFO prio 90 for carrier A*/
while (1)
{
        FD_ZERO (&fds);
        FD_SET (gApplicationISRTracker[0], &fds);

        /* Wait for the data to arrive */
        errCode = select (maxFd + 1, &fds, NULL, NULL, &timeout);
        ...
        if (FD_ISSET(gApplicationISRTracker[0], &fds) != 0)
        {
              /* MSGCOM SFI_Channel1 got something: */
        }
 }
```

```
/* thread SCHED-RR/FIFO prio 90 for carrier B*/
while (1)
{
        FD_ZERO (&fds);
        FD_SET (gApplicationISRTracker[1], &fds);

        /* Wait for the data to arrive */
        errCode = select (maxFd + 1, &fds, NULL, NULL, &timeout);
        ...
        if (FD_ISSET(gApplicationISRTracker[1], &fds) != 0)
        {
              /* MSGCOM SFI_Channel2 got something: */
        }
 }
```

11 Domain

The domain is a helper module which has been designed to help in the initialization and deinitialization of the SYSLIB services. In the older releases the SYSLIB start up sequence has been complicated and customers have faced issues in the correct initialization order.

As mentioned earlier; the SYSLIB services are available on both the DSP and ARM. One of the design goals was to keep the API between the realms to be as consistent as possible; however in certain cases the API compatibility could not be maintained because of the different operating systems.

The domain module abstracts the SYSLIB initialization and cleanup from the application accounting for the differences between the realms. Applications pass to the domain the configuration of the SYSLIB services which are required and the domain module will instantiate the SYSLIB services on the behalf of the application.

During domain initialization; the following components are initialized and instantiated:

1. **Resource Manager Initialization:** This initializes the Navigator subsystem and initializes the RM client subsystem which allows the client to communicate with the Resource manager server.

2. **Name Database:** The function creates and initializes the name database. The name database is used by all the other SYSLIB components to store meta-information.

3. **PKTLIB Instance:** The PKTLIB instance is instantiated with its set of OSAL functions. PKTLIB heaps are created and mapped to the PKTLIB instance.

4. **MSGCOM Instance:** The MSGCOM instance is instantiated with its set of OSAL functions. MSGCOM channels are created and mapped to the MSGCOM instance.

5. **Name Client and Proxy:** This is an optional instantiation which is done and is needed by an application if they wish to communicate with another realm to exchange “names”. There exist multiple name clients in a “realm” but there is always one “proxy” for each realm. The name proxy is instantiated to execute in a polled thread context.

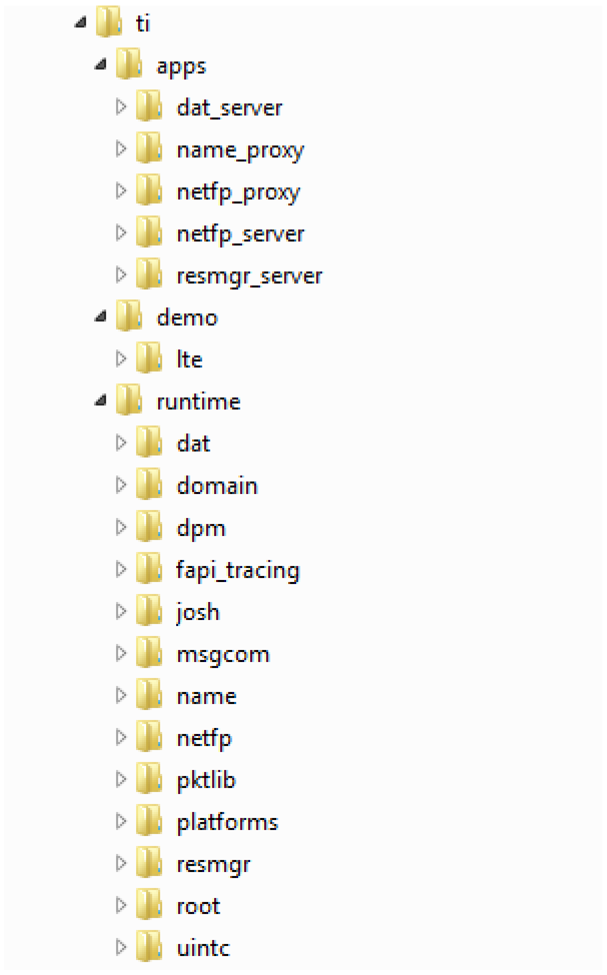
6. **NETFP Client:** This is an optional instantiation which is done and is needed by an application if it uses the NETFP services. The NETFP clients are initialized and synchronized with the server. The NETFP clients are executed in a blocked thread context. The NETFP client thread is used for processing messages from the NETFP server.

7. **DAT Client:** This is an optional instantiation which is done and is needed by an application if it uses the DAT services. The DAT clients are initialized and synchronized with the server. The DAT clients are executed in a polled thread context.

SYSLIB service handles for individual modules (DAT, NETFP clients etc.) can be requested by using the Domain API. These handles can then be passed to the SYSLIB module API to use the requested service.

12 Release structure

The figure below illustrates the release directory structure.



```
ti
  apps
    dat_server
    name_proxy
    netfp_proxy
    netfp_server
    resmgr_server
  demo
    lte
  runtime
    dat
    domain
    dpm
    fapi_tracing
    josh
    msgcom
    name
    netfp
    pktlib
    platforms
    resmgr
    root
    uintc
```

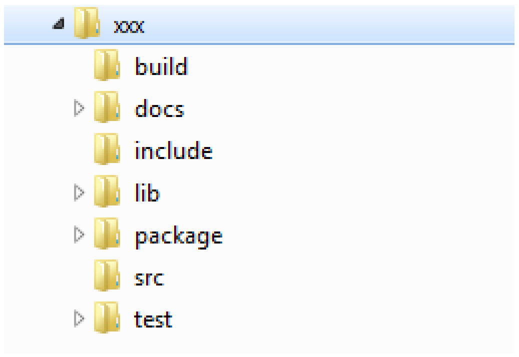
Directory Name	Description
apps	The “apps” directory has all the SYSLIB provided applications which execute in ARM.
demo	This is the LTE Demo application which is provided as a sample to application developers to illustrate how to use the SYSLIB modules to

	design a basic LTE cell. The L1 provided in the demo is a simulated PHY. The demo illustrates the Release9 cell with the L2 executing on the DSP and also a Release10 cell with the L2 executing on ARM
runtime	These are all the SYSLIB modules. The directory structure for each module is explained below:

The SYSLIB release also provides a sample LTE demo application. The demo application showcases the following:

- **Release 9 Cell**: The L2 and simulated L1 application execute on the DSP cores.
- **Release 10 Cell**: The L2 executes on the ARM while the simulated L1 application executes on the DSP cores.

Each module has the following directory structure:-



Directory Name	Description
build	The “build” directory has internal build related files. This is not used by the application developers.
docs	This directory contains the DOXYGEN output related to the module. The API documentation is provided as a part of this directory. Please refer to this folder for the latest API documentation.
include	These are the ***internal*** module header files. These should NOT be included by the application. Application exported header files are exported in the top level module directory.
lib	The library folder has a list of all the libraries for the module. The folder will contain the DSP and ARM libraries. NOTE: Some modules might be DSP (only) or ARM (only). Please refer to the module

	documentation for more details: DSP libraries (if applicable) are present in the lib/<device> directory ARM libraries (if applicable) are present in the lib/ directory. The library name is post fixed with the device name to indicate which device the library supports.
package	Internal ***package*** files which are not used by the application developers.
Src	Source code for the module
test	Unit test code for the module. It is **highly** recommended that application developers browse and understand the test code as it depicts API usage.

12.1 Developing with SYSLIB

SYSLIB releases are device specific and so application developers need to define the following compilation flag. This is required for all applications irrespective of where they execute (DSP or ARM)

Please ensure that the following device specific compilation flags are defined:

Device	Compilation Flag
tmdxevm6638lxe (Hawking)	-DEVICE_K2H

Since SYSLIB supports both the DSP and ARM. The next section explains how to integrate the SYSLIB module with the customer applications depending upon which **realm** they execute in

12.1.1 DSP

Application developers developing an application on DSP integrate the SYSLIB module using the RTSC configuration files. In order to use the SYSLIB module please use the following syntax in the application configuration file:

```
var Netfp  = xdc.useModule('ti.runtime.netfp.Settings');
Netfp.deviceType   = "k2h";
```

This indicates that the application is using the SYSLIB NETFP module. The device type will ensure that the correct device library is used for linking the application. For a list of supported devices please refer to the release notes.

12.1.2 ARM

Application developers developing an application on ARM integrate the SYSLIB modules using makefile. Please ensure that the following are accounted for:-

- **Include directory**
  Ensure that the SYSLIB INSTALL PATH is in the include path for the compiler. This is the path to the top level SYSLIB directory. **For example:** /home/user/ti/syslib_xxx.
  NOTE: Please ensure that the SYSLIB INSTALL PATH is placed in the makefile before the MCSDK devkit include paths.

- **Library**
  Ensure that the library path in the makefile is set to point to the "`lib`" folder for each SYSLIB module being used. For example:-

```
APP_LIBS = -lnetfp_k2h ...
STD_LIBS = -L$(SYSLIB_INSTALL_PATH)/ti/runtime/netfp/lib
```

  The application is using the NETFP library which is always specified with the device name and it includes in the library path the name of directory where the library is located.

- **Compilation Flags**
  All ARM applications need to be built with the following additional compilation flags

```
-D__ARMv7 -D_GNU_SOURCE -D_VIRTUAL_ADDR_SUPPORT
```

12.2 Instantiation

The SYSLIB release now provides support for both the DSP and ARM libraries. The release also provides the following user space daemons on ARM:-

- **Resource Manager Server**: The SYSLIB RM server is responsible for allocating resources on the behalf of different RM clients. RM clients can execute either in the DSP or ARM. The SYSLIB RM server ensures that there are no resource conflicts between the allocations and kernel usage. These are specified and passed to the server through the DTB files.

- **NETFP Server**: The SYSLIB NETFP server is responsible for the management of the NETCP subsystem. Multiple NETFP clients (executing on the DSP/ARM) register themselves with the NETFP server. The NETFP clients use the NETFP server to program the NETCP subsystem. The NETFP Server is configurable using a configuration file. Sample NETFP configuration files are located in the `ti/apps/netfp_server` directory.

- **NETFP Proxy**: The NETFP proxy is an application is a NETFP client which is responsible for interfacing the kernel networking subsystem with the NETFP server. This is used to synchronize the Linux networking stack with the NETFP server.

- **DAT Server**: The DAT Server allows the connection of DAT producers with consumers. Producers/Consumers could be executing on the DSP or ARM. The DAT Server is configurable using a configuration file. Sample DAT configuration files are located in the `ti/apps/dat_server` directory.

Each of the SYSLIB modules now allows for multiple instances to be created. Each instance is provided with its own set of OSAL call functions. This is different from the older SYSLIB architecture. Since there are multiple instances which can exist in the system; there are no more SYSLIB global variables and no more memory placement/cache issues.

12.3 Getting started

The section documents the basic steps which need to be done before starting any development work using SYSLIB.

The SYSLIB supports only the RT kernel from the MCSDK release. Similarly please use the RT DEVKIT for the development of user space applications.

12.3.1 RM Server DTB file

As already mentioned the system resources need to be divided up and managed between all the components in the system. These include the Linux kernel and the remaining application components running on the DSP cores or as ARM processes.

The SYSLIB RM server will need to be passed a list of all the resources which are available to the applications. The SYSLIB RM provides a set of default DTS files in the `runtime/resmgr/dts/<device>` directory. Please refer to the getting started section in the document for more details.

SYSLIB provides a set of default DTS files which divide the resources between the kernel and remaining application components. These files are called:

a. **global-resource-list.dts**

   This is a list of all the resources which are available for management to the SYSLIB RM server.

b. **policy_dsp_arm_syslib.dts**

   This is a list of all the resources along with a list of permissions on which all RM clients are allowed to use the resource.

Please use the following steps:

- Transfer the DTS files to the target.
- Convert the DTS files to DTB files using the on target DTS compiler

```
dtc -I dts -O dtb global-resource-list.dts > global-resource-list.dtb
dtc -I dts -O dtb policy_dsp_arm_syslib.dts >policy_dsp_arm_syslib.dtb
```

NOTE: Applications might decide to modify the default DTS files as per their own requirements or because of customized policies. If so be the case the above steps to create the DTB files need to be redone.

12.3.2 Kernel DTB file update

The default kernel DTB file is modified for the following reasons:

- Free up certain GIC interrupt queues which were not being used by the kernel. These queues are made available to the ARM user space applications.
- Cleaned up the UDMA based resources; since SYSLIB 4.x no longer uses UDMA for transfer between the DSP and ARM.
- Create interrupt mappings using UIO for the GIC, Accumulated and IPC interrupts

The updated kernel DTS files are present in the `runtime/resmgr/dts/<device>` directory. These files are listed below:

```
SYSLIB/ti/runtime/resmgr/dts/<device>/skeleton.dtsi
SYSLIB/ti/runtime/resmgr/dts/<device>/<device>.dtsi
```

```
SYSLIB/ti/runtime/resmgr/dts/<device>/<device>-clocks.dtsi
SYSLIB/ti/runtime/resmgr/dts/<device>/<device>.dts
SYSLIB/ti/runtime/resmgr/dts/<device>/<device>-net.dtsi
SYSLIB/ti/runtime/resmgr/dts/<device>/keystone.dtsi
SYSLIB/ti/runtime/resmgr/dts/<device>/keystone-clocks.dtsi
SYSLIB/ti/runtime/resmgr/dts/<device>/keystone-qostree.dtsi
```

Please use the following steps to update the default kernel DTS file:

- Transfer the above DTS file(s) to the target.
- Convert the DTS file to a DTB file using the on target DTS compiler

```
dtc -I dts -O dtb <device>-evm.dts > uImage-<device>-evm.dtb
```

- **First time only;** create a temporary directory

```
mkdir /mnt/boot
```

- Now; use the following commands to update the kernel DTB file

```
mount -t ubifs ubi0_0 /mnt/boot

rm /mnt/boot/uImage-<device>-evm.dtb

cp uImage-<device>-evm.dtb /mnt/boot/uImage-<device>-evm.dtb

umount /mnt/boot
```

NOTE: The above scripts are provided only as an example and are subject to change. Please refer to the MCSDK UG for the recommended sequences.

- Reboot the kernel.

Appendix A: UIA config file for System Analyzer

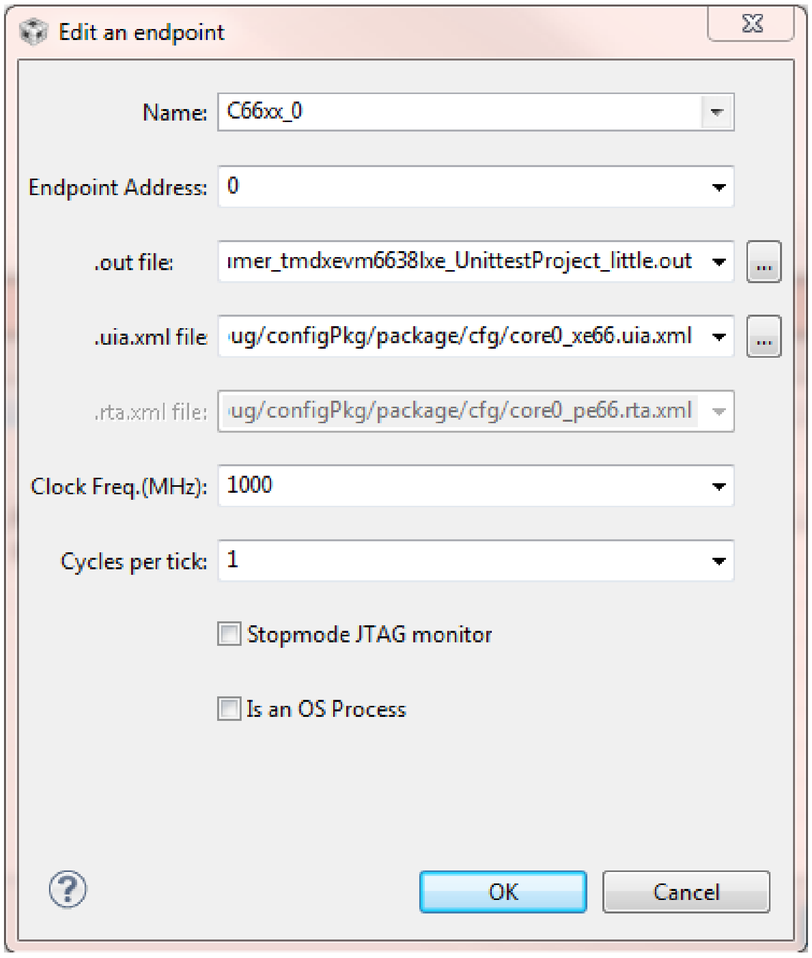
Introduction

The section describes how to setup UIA config file to analyze UIA logs. System Analyzer can be started from CCS Tools menu -> System Analyzer. Please refer to SystemAnalyzer_Getting_Started_Guide.pdf from UIA package for more information.

This procedure is the same for both live capture and binary files. UIA config file can be created through System Analyzer menu->UIA config.

A1.1 Creating UIA config file for DSP producer

Endpoint configuration is shown below. The endpoint address is DAT client id configured at DAT client initialization time.



Name:	C66xx_0
Endpoint Address:	0
.out file:	ımer_tmdxevm6638lxe_UnittestProject_little.out
.uia.xml file	ug/configPkg/package/cfg/core0_xe66.uia.xml
.rta.xml file:	ug/configPkg/package/cfg/core0_pe66.rta.xml
Clock Freq.(MHz):	1000
Cycles per tick:	1

☐ Stopmode JTAG monitor

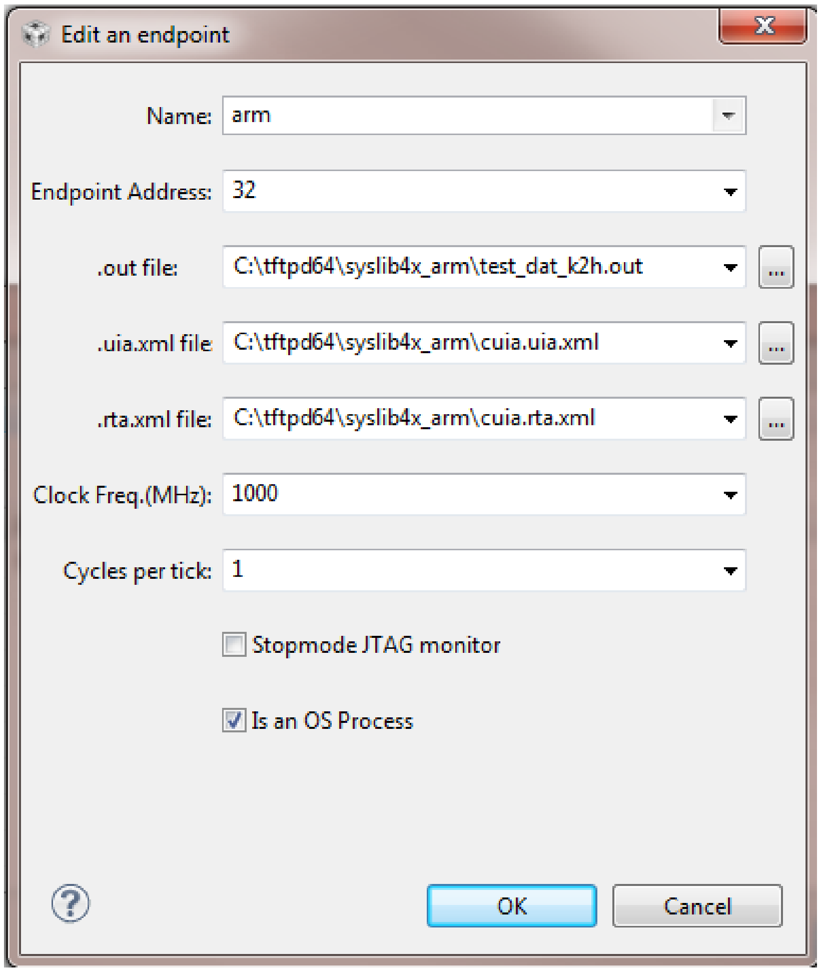
☐ Is an OS Process

Notes:

1. If loggers are created dynamically, the logger needs to be set in *.rta.xml file. This is file is generated when compiling the DSP projects. Dynamic logger configuration is available form ti\runtime\dat\test\metadataFiles\uiadynamicLogger.rta.xml. uiadynamicLogger.rta.xml needs to be merged into *.rta.xml file generated from DSP builds.

A1.2 Creating UIA config file for ARM Producer

Please refer to cUIA_Linux_Example_Program_User_Guide.html from cUIA package section “Configuring and starting System Analyzer” on the steps to create UIA config file. An example is shown below:



Name:	arm
Endpoint Address:	32
.out file:	C:\tftpd64\syslib4x_arm\test_dat_k2h.out
.uia.xml file	C:\tftpd64\syslib4x_arm\cuia.uia.xml
.rta.xml file:	C:\tftpd64\syslib4x_arm\cuia.rta.xml
Clock Freq.(MHz):	1000
Cycles per tick:	1

☐ Stopmode JTAG monitor

☑ Is an OS Process

Notes:

1. Please use “32” as endpoint address for ARM producers.
2. *.uia.xml and *.rta.xml files for ARM producers can be found ti\runtime\dat\test\metadataFiles\cuia.
3. The out file name should match with the application name used to generate the CRC value when creating the ARM producer. Producers from the same application should use the same crc value.