

SysLink 02.00.00.68 beta1 UserGuide

SysLink User Guide

Introduction

This document is the User Guide for SYS/Link. It gives information about the SYS/Link architecture, modules, along with their features, concepts and programming tips.

SYS/Link, also known as SysLink, is runtime software and an associated porting kit that simplifies the development of embedded applications in which either General-Purpose microprocessors (GPP) or DSPs communicate with each other. The SYS/Link product provides software connectivity between multiple processors. Each processor may run either an HLOS such as Linux, WinCE, etc. or an operating system such as SYS/BIOS™ or PrOS. In addition, a processor may also be designated as the master for another slave processor, and may be responsible for controlling the slave processor's execution (including boot-loading the slave).

The SYS/Link product provides the following services to frameworks and applications:

- Processor Manager
- Inter-Processor Communication
- Utility modules

Terms and Abbreviations

Abbreviation	Description
HLOS	Higher Level Operating System
RTOS	Real Time Operating System
CCS	Code Composer Studio
IPC	Inter-Processor Communication
GPP	General Purpose Processor e.g. ARM
DSP	Digital Signal Processor e.g. C64X
CGTools	Code Gen Tools, e.g. Compiler, Linker, Archiver
SysLink	SYS/Link

This bullet indicates important information. Please read such text carefully.

Features

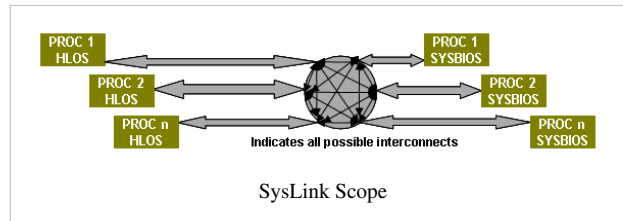
SysLink provides communication and control infrastructure between multiple processors present on the target platform and is aimed at traditional embedded applications. SysLink comprises of the following sub-components:

- System Manager
- Processor Manager
- Inter-Processor Communication (IPC)
- Utility modules

The target configurations addressed by the SysLink architecture are:

- Intra-Processor
 - Inter-Processor
-

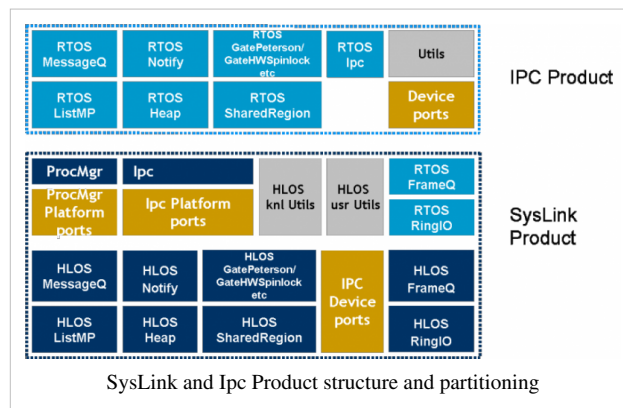
- Processor may be GPP or DSP, running HLOS or SYS/Bios



The SysLink product consists of a Porting Kit product as well as support for a set of specific target devices.

Product structure

The SysLink product is used in conjunction with the IPC product. The product structuring is as shown below:



- The common header files for IPC implemented by both the IPC product (on RTOS-side) and the SysLink product (on HLOS-side) are available within ti/ipc package.
- The SysLink product, in addition to implementing the HLOS-side of all IPC modules available within the RTOS IPC product, adds the following:
 - ProcMgr
 - FrameQ (HLOS & RTOS)
 - RingIO (HLOS & RTOS)

System Purpose

System Interface

The SysLink product provides software connectivity between multiple processors. Each processor may run either an HLOS such as Linux, WinCE, Symbian etc. or an RTOS such as SYS/Bios or PrOS. Based on specific characteristics of the Operating system running on the processor, the software architecture of each module shall differ.

The SysLink product provides three types of services to frameworks and applications:

- **System Manager:** The System Manager manages system level resources and overall initialization/finalization for the system. It provides a mechanism of integrating and providing a uniform view for the various sub-components in the system. In addition to overall system management, it also includes the System Memory Manager sub-component.
- **Processor Manager:** The Processor Manager on a master processor provides control functionality for a slave device.
- **IPC:** IPC modules are peer-to-peer protocols providing logical software connectivity between the processors.

- Utility modules: Utility modules provide utility services utilized by the IPCs and ProcMgr.

Architecture goals

The architecture aims to provide reusable & portable modules for the sub-components within SysLink. The architecture is portable to multiple Operating Systems such as:

- SYS/Bios 6
- Linux
- WinCE

The porting kit architecture has a generic code base with maximum reuse across OS's and devices.

The SysLink architecture is modularized in a way to enable clear interfaces and ownership for module definition.

Identical SysLink APIs are provided across all processors and Operating Systems.

Architecture Overview

Dependencies

The SysLink product has dependencies on other several other components: On HLOS side:

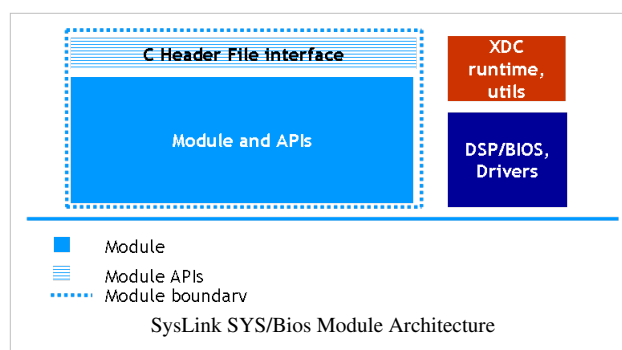
1. Base port: SysLink has a dependency on the base port to the target device.
2. OSAL: In user-space, SysLink utilizes the Operating System Adaptation Layer
3. Kernel utils: In kernel-space, the SysLink product utilizes the services available within the kernel utils. This includes OSAL services as well as generic utilities such as trace, string functions etc.
4. Types: SysLink uses types matching those defined within xdc/std.h available with the xdctools product.
5. Configuration: SysLink provides dynamic configuration for all modules in both user-side and kernel-side. For the Linux product, kernel configuration mechanisms shall be used.

On SYS/Bios-side:

1. SYS/Bios: SysLink depends on SYS/Bios port available on the target device.
2. Build: The SysLink product uses xdc build mechanism
3. Configuration: SysLink uses XDC configuration.
4. XDC runtime: SysLink uses services available within XDC runtime such as trace, asserts etc.

SysLink SYS/Bios Module Architecture

The architecture of each module on SYS/Bios shall follow the following structure:



Each module (System Manager/Processor Manager/IPC) on SYS/Bios is a RTSC module. It uses the XDC configure and build mechanism to ensure that the advantages of whole-program optimization are fully utilized. It also makes use of XDC runtime utils, asserts and tracing mechanisms.

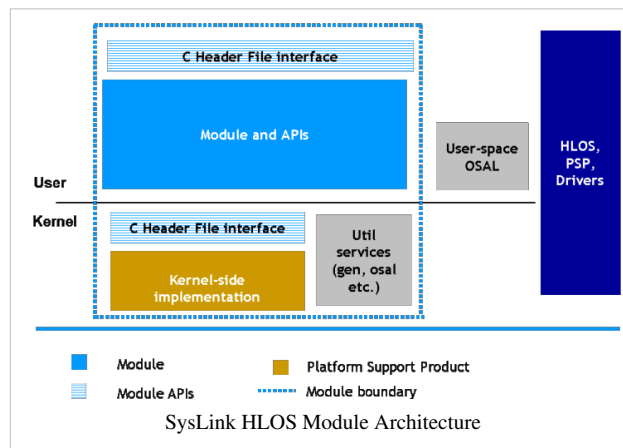
Each module implements the common C Header interface APIs defined for SysLink.

The module additionally defines internal interfaces as required to ensure plug-and-play of different types of implementations of the physical drivers.

In SYS/Bios, all module initialization get plugged into the startup sequence of the Operating System itself. Due to this, users do not need to call specific initialization functions for the modules.

SysLink HLOS Module Architecture

The architecture of each module on the HLOS shall follow the following structure:



On the HLOS, the module configuration is done from the kernel-side, by the system integrator at startup. More details on this are given in a later section on Configuration.

The HLOS modules use a make-file based build system. They utilize an OSAL on user-side for abstracting OS-specifics from the implementation.

In HLOS, the modules are implemented as drivers within the HLOS. The drivers may be statically or dynamically loaded. Each module has one top-level initialization function that must be called only once. This happens as part of the driver loading. SysLink supports multiple applications/clients using the modules simultaneously, and hence internally has all the checks required to ensure that multiple applications attempting system-level actions are controlled. For example, multiple clients attempting to change the state of the co-processor shall not be allowed to disrupt the co-processor state machine. Only the first client to initiate each state change shall be complied with, and other requests ignored. For example, only the first client to call ProcMgr_start shall actually result in starting the co-processor. Other clients making this call for the same co-processor shall be ignored unless the co-processor had reached the appropriate state from which the 'start' state change can be initiated. The build system supports optionally building all modules into a single driver or having separate drivers for each module.

- If all modules are integrated into a single driver, it provides better ease-of-use to customers who do not wish to load separate one or more drivers for each module. For this use-case, the static configuration can be used to decide which modules are included within the build.
- If each module is provided as one or more separate drivers, it enables easier plug-ability and allows users to dynamically add new drivers for different hardware/usage without having to modify existing modules.

User-kernel Separated Operating Systems

For such Operating Systems, the APIs shall drop down into the kernel-side.

- The core implementation for all modules shall be available on kernel-side. This shall enable provision of both user-side and kernel-side APIs for all modules.
- The configuration of the modules shall be done from kernel-side. This shall enable application clients on both kernel and user-sides to directly start using SysLink.
- There are two types of configuration: Module system-level configuration and application specific configuration.
 - The module-level configuration shall happen only once.
 - The application-level configuration is limited to specific instances of the module (e.g. MessageQ instances) that are created and used by the application. This configuration shall not happen at system initialization time, but shall happen whenever the instance is created by the module. This may happen either from user-side or kernel-side, depending on whether the specific instance is being created from user-side or kernel-side.
- A set of Util services shall be designed for the kernel-side, which shall provide OS abstraction and generic utility services, to enable easy porting to other Operating Systems.

Flat Memory Operating Systems

For such Operating Systems, only the kernel-side implementation shall be directly used. The user-side does not need to be ported at all.

Modules

System Manager

The Ipc module (System Manager) provides a means of integrating all the individual Processor Managers and IPCs into a single comprehensive product. It also provides management of system resources such as system memory.

The basic functionality provided by the Ipc module on all processors includes:

- Initialization and Finalization of the SysLink product
 - This includes internally calling the initialization and finalization functions for the individual modules included in the build, including IPCs and Processor Managers.
 - Providing memory (for example, shared memory) to the other SysLink modules. This includes interaction with the IPCs and Processor Manager as required.
- System configuration
 - Any system level configuration information is taken by the System Manager.
 - The Ipc module also consolidates configuration information for other modules to present a unified SysLink view.
 - The Ipc module on HLOS also configures itself with the information from the slave-side when the slave executable is loaded. This enables the HLOS-side to configure itself with the identical configuration as the slave-side (for example, shared memory configuration). This ensures that the user is not required to modify HLOS-side configuration whenever the slave-side configuration changes.

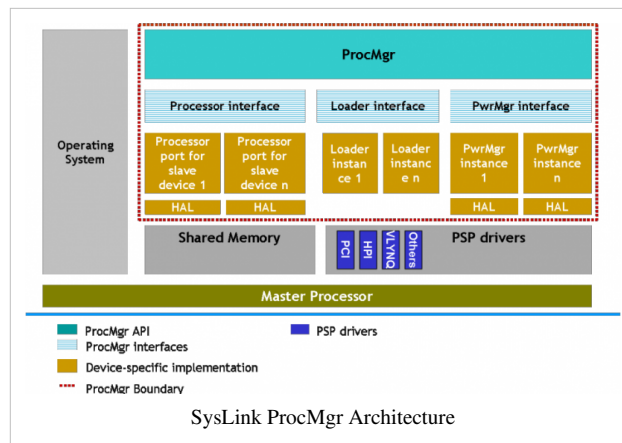
Processor Manager

The ProcMgr module provides the following services for the slave processor:

- Slave processor boot-loading
- Read from or write to slave processor memory
- Slave processor power management
- Slave processor error handling
- Dynamic Memory Mapping

The Processor Manager (Processor module) shall have interfaces for:

- **Loader:** There may be multiple implementations of the Loader interface within a single Processor Manager. For example, COFF, ELF, dynamic loader, custom types of loaders may be written and plugged in.
- **Power Manager:** The Power Manager implementation can be a separate module that is plugged into the Processor Manager. This allows the Processor Manager code to remain generic, and the Power Manager may be written and maintained either by a separate team, or by customer.
- **Processor:** The implementation of this interface provides all other functionality for the slave processor, including setup and initialization of the Processor module, management of slave processor MMU (if available), functions to write to and read from slave memory etc.



All processors in the system shall be identified by unique processor ID.

As per need, a Hardware Abstraction Layer (HAL) shall be internally used to abstract hardware-specific details from generic implementation. This shall enable easier port to different underlying hardware.

Types of slave executable loaders

SysLink ProcMgr module is designed to support multiple types of loaders for the slave executable. Two loader types, supporting COFF and ELF formats are provided as part of the SysLink release.

The support available depends on the supported platform. For details on support for each platform, refer to the InstallGuide for the specific release.

Additional points to note:

- The config.bld file has been updated to build the above mentioned targets for both COFF and ELF
 - For the ELF targets in the config.bld, the following additional linker options need to be specified:
 - `--dynamic`
 - `-retain=_Ipc_ResetVector`
 - For example: `C64P_ELF.lnkOpts.suffix += " --dynamic --retain=_Ipc_ResetVector";`

- Note that the extension of generated ELF executables has an 'e' in it, e.g. notify_omap3530dsp.xe64P as compared to the corresponding COFF executable, which is notify_omap3530dsp.x64P
- On Platforms with multiple salve:
 - The current ELF loader only supports sequential loading of the slaves.
 - One slave must be fully loaded and started (including loadcallback and startcallback) before the load & start of the next slave is commenced. Parallel load/start is not supported
 - APIs such as ProcMgr_getSymbolAddress, which require support for simultaneous multiple loader instances, are not supported once multiple slaves have been started.

Types of ProcMgr boot modes

- ProcMgr module supports 4 different types of boot modes for the slave processors.
 - **ProcMgr_BootMode_Boot:** ProcMgr is responsible for loading the executable on the slave core, powering it up, starting it running, stopping its execution and powering it down.
 - **ProcMgr_BootMode_NoLoad_Pwr:** An external loader is used, and ProcMgr is not responsible for loading the slave. It is responsible for powering it up, starting it running, stopping its execution and powering it down.
 - **ProcMgr_BootMode_NoLoad_NoPwr:** An external loader is used, which also powers up the slave core. ProcMgr is not responsible for loading or powering up the slave. It expects the slave core to be powered up and in reset when the ProcMgr APIs are called. ProcMgr is responsible for starting the slave core running and stopping its execution by placing it in reset.
 - **ProcMgr_BootMode_NoBoot:** An external boot-loader is used, which powers up the slave core and also starts it running. ProcMgr is not responsible for boot-loading the slave core. It expects the slave core to be already executing.

ProcMgr Boot Modes Calling Procedure Comparison

Boot Mode	proc_attach	proc_detach	proc_start	proc_stop	pwr_attach	pwr_detach	loader_attach	loader_detach
ProcMgr_BootMode_Boot	reset	reset	release	reset	power_on	power_off	load	unload
ProcMgr_BootMode_NoLoad_Pwr	reset	reset	release	reset	power_on	power_off	-	-
ProcMgr_BootMode_NoLoad_NoPwr	-	-	release	reset	-	-	-	-
ProcMgr_BootMode_NoBoot	-	-	-	-	-	-	-	-

- Except ProcMgr_BootMode_Boot, the remaining boot modes require the users to provide the address of _Ipc_ResetVector symbol from the slave executable to the Ipc_control (cmd type = Ipc_CONTROLCMD_LOADCALLBACK) API. The address of the **_Ipc_ResetVector** symbol can be found from the map file of the respective core executable.

e.g

ti/syslink/samples/rtos/gateMP/package/cfg/ti_syslink_samples_rtos_platforms_ti816x_dsp/whole_program_debug/gatemp_ti816x_d

Example to call Ipc_control for **ProcMgr_BootMode_Boot** Mode:

```
Ipc_control (procId,
            Ipc_CONTROLCMD_LOADCALLBACK,
            NULL) ;
```

Example to call Ipc_control for **ProcMgr_BootMode_NoLoad_Pwr** Mode:

```
Ipc_control (procId,
            Ipc_CONTROLCMD_LOADCALLBACK,
            (Ptr) &resetVector) ;
```

Example to call Ipc_control for **ProcMgr_BootMode_NoLoad_NoPwr** Mode:

```
Ipc_control (procId,  
            Ipc_CONTROLCMD_LOADCALLBACK,  
            (Ptr) &resetVector);
```

Example to call Ipc_control for **ProcMgr_BootMode_NoBoot** Mode:

```
Ipc_control (procId,  
            Ipc_CONTROLCMD_LOADCALLBACK,  
            (Ptr) &resetVector);
```

Inter-Processor Communication Protocols

Multiple Inter-Processor Communication Protocols are supported by SysLink:

- Notify
- MessageQ
- ListMP
- GateMP
- HeapBufMP
- HeapMemMP
- FrameQ
- RingIO

All IPC modules have the following common features:

- Instances of the IPC are identified by system-wide unique names
- On HLOS-side, each IPC has a <Module>_setup and <Module>_destroy API that initializes/finalizes the IPC module. IPC-specific configuration may be provided to the <IPC>_setup call.
- An instance of the IPC shall be created with a <Module>_create and deleted with <Module>_delete API. A system-wide unique name is used to create and delete the IPC instance
- <Module>_open API is used to get a handle to the instance that can be used for further operations on the IPC instance. The handle can be relinquished through a call to <Module>_close.
- Further calls to the IPC depend on the type of IPC.
- The configuration of IPC and their physical transports is either dynamic or static as decided by the system integrator, and the Operating System on each processor. Where XDC configuration is supported, some configuration is possible through static configuration.
- Each IPC module supports enabling trace information for debugging. Different levels of trace shall be supported.
- Some IPCs support APIs to provide profiling information.

The architecture of each IPC module follows the generic module architecture described in the earlier sections. For some IPC modules, there may be different implementations of the IPCs transports/drivers depending on the physical link supported and chosen on the device.

As per need, a Hardware Abstraction Layer (HAL) shall be internally used to abstract hardware-specific details from generic implementation. This shall enable easier port to different underlying hardware.

Notify

The Notify module abstracts physical hardware interrupts into multiple logical events. It is a simple and quick method of sending up to a 32-bit message.

The Notify IPC provides the following functionality:

- Initialize and configure the Notify component
- Register and un-register for an event.
 - Multiple clients can register for same event
 - Clients may be processes or threads/tasks
 - Callback function can be registered with an associated parameter to receive events from remote processor
 - All clients get notified when an event is received
 - Notification can be unregistered when no longer required
- Send an event with a value to the specified processor.
 - Multiple clients can send notifications to the same event
 - Events are sent across to the remote processor
 - 32-bit payload value can be sent along-with the event
- Receive an event through registered callback function.
 - Event IDs may be prioritized.
 - 0 <-> Highest priority
 - (Max Events - 1) <->Lowest priority
 - If multiple events are set, highest priority event is always signaled first
- Disable/enable events

Notify component may be used for messaging/data transfer if:

1. Only 32-bit information (payload) needs to be sent between the processors.
2. Prioritization of notifications is required. For example, one low priority event (e.g. 30) can be used for sending buffer pointers. A higher priority event (e.g. 5) can be used to send commands.
3. The notification is to be sent infrequently. Notify module allows the user to specify, when sending an event, whether it wishes to spin till the previous event (for the same event number) has been cleared by the other side, or send the event and potentially lose it.
 1. If the waitClear option is chosen, if multiple notifications for the same event are sent very quickly in succession, each attempt to send a specific event spins, waiting till the previous event has been read by the other processor. This may result in inefficiency.
 2. If the waitClear option is not chosen, the application protocol must not use the payload value being sent along with the notification. This is because since there is no wait for the other side to clear the previous event, the payload will get overwritten. Also, if this option is used, the application protocol must be able to withstand 'lost' events. i.e. it may receive only one event notification even if the same event has been sent multiple times.
4. Multiple clients need to be able to register for the same notification event. When the event notification is received, the same notification with payload is broadcast to all clients registered for that event.

Reserved Events

Out of total available events some are reserved for use by different modules for special notification purposes. Following is the list of events that are currently reserved by different modules:

Module	Event Ids
FrameQBufMgr	0
FrameQ	1
MessageQ(TransportShm)	2
RingIO	3
NameServerRemoteNotify	4

Notify_sendEvent calling context

- Applications should not call Notify_sendEvent from ISR context.
 - When Notify_sendEvent is called with waitClear as TRUE, it spins waiting for the receiving processor to clear the previous event for the same event ID.
 - Due to this, if Notify_sendEvent is called from ISR context, it may result in high interrupt latency.
 - The receiving processor may be occupied in higher priority activities due to which it may not be able to process the event in a deterministic amount of time.
 - If Notify_sendEvent is called from ISR context, this may result in the interrupt latency on the sending processor to become non-deterministic, which is not advisable.
- This API must not be called under GateMP lock protection to avoid deadlocks

MessageQ

This component represents queue based messaging. It is an acronym for 'message queue'. The MessageQ IPC has the following features:

- This component is responsible for exchanging messages of variable length between the clients on one or more processors.
- The messages are sent and received through message queues.
- Each Message Queue is identified by a system-wide unique name.
- A reader gets the message from the queue and a writer puts the message on a queue. A message queue can have only one reader and many writers. A task may read from and write to multiple message queues.
- The client is responsible for creating the message queue if it expects to receive messages. Before sending the message, it must open the queue where message is destined.

The MessageQ component may be used for messaging/data transfer if:

1. Application requires single reader and multiple writers.
2. More than 32-bit information needs to be sent between the processors using application-defined message structures.
3. Variable sized messages are required.
4. Reader and writer operate on the same buffer sizes.
5. Messages need to be sent frequently. In this case, the messages are queued and there is no spin-wait for the previous event to be cleared.
6. The ability to wait when the queue is empty is desired. This is inbuilt within the MessageQ protocol, and no extra application code is required. If MessageQ_get () is called on an empty queue, it waits till a message is received. If Notify is used, the application must register the callback, or wait on a semaphore that is posted by the

application's notification callback function.

7. It is desired to have the ability to move the Message Queue between processors. In this case, the MessageQ_open() on other processors internally takes care of locating the queue, and the application code sending messages to this queue does not need to change.

ListMP

The ListMP module provides multi-processor doubly-linked circular linked list (ListMP) between multiple processors in the system. Features of the ListMP module include:

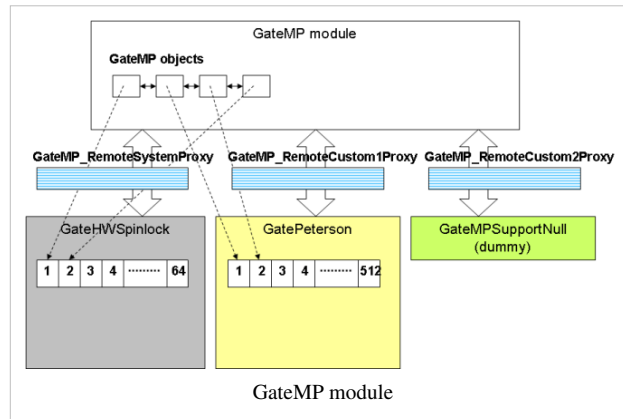
- Simple multi-reader, multi-writer protocol.
- Instances of ListMP objects can be created and deleted by any processor in the system.
- Each instance of the ListMP is identified by a system-wide unique string name.
- To use the ListMP instance, client opens it by name and receives a handle.
- Elements can be put at the tail of the list, removed from head of the list
- APIs are also available for inserting & removing elements at/from an intermediate location
- To peek into list contents, APIs are available for list traversal.
- The APIs are protected internally using Gate.
- ListMP does not have any inbuilt notification mechanism. If required, it can be built on top using Notify module.
- Buffers placed into the ListMP need to be from shared memory. This includes buffers allocated from Heap, as well as those mapped using Dynamic Memory Mapping.

ListMP component may be used for messaging/data transfer if:

1. Application requires multiple writers and multiple readers.
2. The application wishes to perform out-of-order processing on received packets. This is not possible with MessageQ or with Notify. With ListMP, the reader may traverse the shared list and choose any buffer within the list to be removed. However, this is possible only when 'shared' ListMP is used, i.e. ListMPSharedMemory is used. If fast queues are used (hardware assisted), then list traversing may not be possible.
3. Making a specific buffer as high priority is desired. The sender to an ListMP can make a specific buffer/message as high priority by pushing it to the head of the queue instead of placing it at the end of the queue. APIs are provided to traverse the list and insert element before any specified element in the queue. However, this is possible only when 'shared' ListMP is used, i.e. ListMPSharedMemory is used. If fast queues are used (hardware assisted), then this may not be possible.
4. Inbuilt notification is not required. If the application desires flexibility in when notification is to be sent/received, ListMP module can be used. The application may use Notify module to send & receive notifications as per its specific requirements. This may result in better performance and lesser number of interrupts, tuned to application's requirements. However, the disadvantage is that additional application code needs to be written for notification, which is present inherently within MessageQ component.
5. Reader and writer operate on the same buffer sizes.
6. More than 32-bit information needs to be sent between the processors using application-defined message structures.
7. Variable sized messages/data buffers are required.
8. Messages/data buffers need to be sent frequently. In this case, the messages are queued directly. No notification/spin-wait for notification is performed.

GateMP

The GateMP module provides the Multi-processor critical section between multiple processors in the system. Depending on the hardware functionality available, it shall be possible to write different types of Gate implementations. For example, GatePeterson is a software multi-processor Gate where h/w support is not available. GateHwSpinlock is a Gate that uses h/w spin-lock feature.



The Multi-processor Gate module provides the following functionality:

- Achieves mutually exclusive access to shared objects (data structures, buffers, peripherals etc.)
- Instances of Multi-processor Gate objects can be created and deleted by any processor in the system.
- Each instance of the Multi-processor Gate is identified by a system-wide unique string name.
- To use the Multi-processor Gate instance, client opens it by name and receives a handle.
- Client calls an enter API to achieve exclusive access to the protected region. It blocks/spins till access is granted.
- Client calls a leave API to release the critical section.

Multi-processor Heaps

The HeapMemMP, HeapBufMP, and HeapMultiBufMP modules configure and manage shared memory regions across processors. The features provided by these modules include:

- Support for multiple clients on HLOS and SYS/Bios
- Allows configuration of shared memory regions as buffer pools
- Support for allocation and freeing of buffers from the shared memory region.
- These buffers are used by other modules from IPC for providing inter-processor communication functionality.

The Memory APIs, into which the individual modules plug in, provide a uniform view of different memory pool implementations, which may be specific to the hardware architecture or OS on which IPC is ported.

This component is based on the IHeap interface in XDC.

HeapBufMP

This module provides a single fixed size buffer pool manager. It manages multiple buffers of one fixed size, as specified while configuring the Heap.

This module is similar to the SYS/Bios HeapBuf, however it works off shared memory and ensures the appropriate protection and features to allow other processors to use this Heap to allocate and free buffers.

HeapMultiBufMP

This module provides a fixed size buffer pool manager. It manages multiple buffers of some fixed sizes, as specified while configuring the Heap. For example, it would support a pre-configured set of 'n' buffers of size 'x', 'm' buffers of size 'y' etc.

This module is similar to the SYS/Bios HeapMultiBuf, however it works off shared memory and ensures the appropriate protection and features to allow other processors to use this Heap to allocate and free buffers.

HeapMemMP

This module provides a heap based variable sized memory allocation mechanism.

The main difference between this module and the HeapMem available with SYS/Bios, is that the HeapMemSM works off shared memory regions, and also takes into account the fact that other processors would be also using this Heap to allocate and free buffers.

FrameQ

FrameQ is a connecting component designed to carry video frames from one processing component in the system to another. FrameQ is a queue data structure which allows queuing and de-queuing of frames; frame carries all the relevant information for a video frame such as frame buffer pointers (YUV data), frame type (RGB565/YUV420/YUV422 etc.), frame width, frame height, presentation timestamp etc.

FrameQ can be implemented to operate across processors/processes boundaries, hiding the nuances of IPC, address translation and cache management etc.

FrameQ internally uses a special buffer manager: FrameQBufMgr. The FrameQ module provides the following features:

- Single writer, multiple readers
- FrameQ instances identified by system-wide unique names.
- Frames can be allocated and freed
- Duplicate operation allocates and initializes an additional frame which points to the same frame buffer
- The frame can be put into the Frame Queue by the writer
- Notifications can be set by writer for free buffer available to allocate the frames. For reader, notifications provide a capability to minimize interrupts by choosing specific notification type. The notification is configurable based on threshold and type (ONCE/ALWAYS)
- Readers can retrieve frames from the Frame Queue.
- Multi-Queue operations are also supported, in which multiple frames can simultaneously be allocated/freed/put/get into multiple channels in each FrameQ, for better multi-channel performance.

FrameQ module internally uses the FrameQBufMgr, which provides allocation, free and notification mechanisms for frames.

- FrameQBufMgr is configured at create time with all characteristics of the frame, including number of frame buffers, sizes, characteristics etc.
 - The API to allocate a new frame returns a fully constructed frame as per the configuration done at create time.
-

- The module maintains reference count for frame buffers to allow the same buffers to be used by multiple clients (duplication)
- The module free the allocated buffers by decrementing the reference count. When all clients using the buffer have freed it, it actually frees the buffer to the pool. If there are clients registered for notification on free, this results in sending a notification to the client waiting for a free buffer. The client may be on the same or a different processor.
- The frames can be allocated/duplicated and freed on any processor in the system.

FrameQ component may be used for data transfer if:

1. Multiple readers and single writer are required.
2. This module is suitable for video processing.
3. The data is exchanged as a set of frames with each frame pointing to one or more frame buffers.
4. The same data is required to be consumed (read) by multiple clients. FrameQ module allows the 'dup' functionality, which creates a new frame with the same data buffers plugged into it. This new frame can be sent to a different client, which can also read the same data. The frame data buffers are reference counted, and freed only when all clients using the 'duplicated' frames free them. However, care should be taken when using this feature, since any modifications in-place in the frame buffers can impact other clients using the same buffers.
5. The writer and reader(s) work with the same buffer sizes.
6. By creating multiple readers, the FrameQ module internally supports automatic duplication and sending of the frames to multiple clients. If a FrameQ instance has multiple readers, each frame sent to the FrameQ instance is duplicated and placed into all readers.
7. One or more buffers can be attached to a single frame. This is useful when multiple buffers (e.g. YUV) are to be sent as a single unit.
8. In case free buffers are no longer available, it is required for application to be able to wait on a notification when free buffers become available. FrameQ module supports such notification based on buffer threshold and notification types.
9. Applications have different notification needs. The application can minimize interrupts by choosing the most appropriate type of notification based on watermark.

Usage of Extended Header

Frame

A frame contains Frame Header and Frame Buffer. To send frames (frame buffer and associated frame Header) to reader client, writer needs to allocate frames using **FrameQ_alloc** API. **FrameQ_alloc** internally allocates buffers for frame header and frame buffer and populates the base frame Header and returns the frame to the caller. The physical size of the frame and frame buffer are fixed and specified during creation time.

Frame Header

The frame Header contains Base Frame Header. Base Frame Header is the initial portion of the Frame Header (in turn Frame) which is mandatory and should be at the beginning of the frame Header(Frame)

Base Frame Header contains the following fields:

```
{
    reserved0;

    reserved1;

    baseHeaderSize;

    headerSize;
```

```

    frmAllocatorId;

    frameBufType;

    freeFrmQueueNo;

    numFrameBuffers;

    frameBufInfo[1];
        +
        frameBufInfo[n-1]; //based on number of Frame Buffers
}

```

Caller can modify the non reserved fields of Base Frame Header using FrameQ header manipulation API.

Extended Frame Header

Frame Header can be extended to contain application specific header. But such extended frame header must start after base frame header contents in a frame header(frame). Extended Header is user defined and can be used by applications based on their requirement.They can pass additional information such as codec params, additional attributes of the frame buffers etc.

For example

Consider Application header structure is

```

struct appHeader {
    int a;
    int b;
};

```

So application should configure the frame header equal to sizeof appHeader + size of base frame header.

If application wants to write application header or Read application header,

FrameQ_getExtendedHeaderPtr will give a pointer where application can write/read their application header information.

/* Example of write*/

```

struct appHeader *apphdr;
extptr = FrameQ_getExtendedHeaderPtr (frame);/* frame is the
pointer to a frame obtained through FrameQ_alloc */
apphdr =(struct appHeader *)extptr;
apphdr->a = 1080;
apphdr->b = 60;

```

/* Example of Read*/

```

int a;
int b;
struct appHeader *apphdr;
extptr = FrameQ_getExtendedHeaderPtr (frame);/* frame is the
pointer to a frame obtained through FrameQ_get */

```

```
apphdr  =(struct appHeader *)extptr;  
a  = apphdr->a;  
b  = apphdr->b;
```

RingIO

This component provides Ring Buffer based data streaming, optimized for audio/video processing. The RingIO IPC has the following features:

- Single reader, single writer
- RingIO instances identified by system-wide unique names.
- Provides true circular buffer view to the application
 - Internally handles wraparound
- Data and attributes/messages associated with data can be sent
 - In-band attributes are supported
- Reader and writer can work on different data sizes
 - Use case: Writer can be unaware of reader buffer size needs
- Reader and writer can operate completely independent of each other
 - Synchronization only needed when acquire fails due to insufficient available size
- Capability to minimize interrupts by choosing specific notification type
 - Notification configurable based on threshold and type (ONCE/ALWAYS)
- Ability to cancel unused (acquired but unreleased) data
 - Use case: Simultaneously work on two frames: previous and new
- Ability to flush out released data
 - Use case: stop/fast forward
- Optimum usage of memory
 - One large ring buffer used by reader & writer for smaller buffer sizes as required
 - Avoids fragmentation issues
 - Variable sized buffers can be used

RingIO component may be used for messaging & data transfer if:

1. Single reader and single writer are required.
2. This module is suitable for audio processing.
3. Data, as well as attributes/messages associated with data are to be sent & received.
4. Writer and reader need to execute independently of each other. The size of buffer used by writer may be different from the buffer size needed by reader.
5. The buffer sizes for acquire and release for writer & reader do not need to match. Writer/reader may choose to release lesser buffer size than was acquired, or may choose to acquire more data before releasing any data.
6. Applications have different notification needs. The application can minimize interrupts by choosing the most appropriate type of notification based on watermark.
7. It is desired to have the capability to cancel unused data that was acquired but not released.
8. It is desired to flush the contents of the ring buffer to clear the ring buffer of released data. For example, when ongoing media file play is stopped and new media file is to be streamed and decoded.

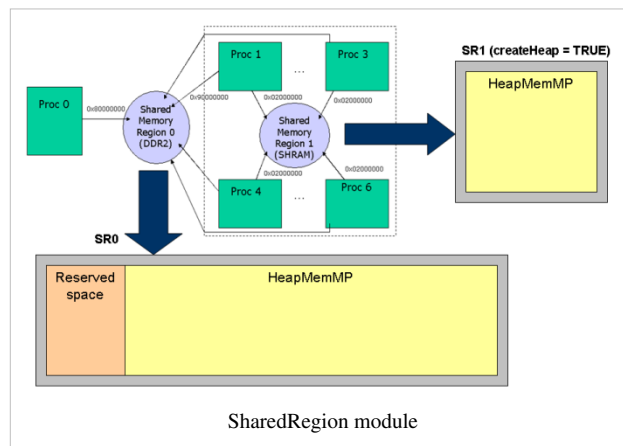
Utility modules

The following utility modules are supported by SysLink:

- SharedRegion
- List
- Trace
- MultiProc
- NameServer

SharedRegion

SharedRegion module provides the management of shared memory regions across processors. It provides a way to configure (statically or dynamically) shared memory regions, which can be used by other IPC modules for standardized addressing, including address translation. On each core, the virtual address on that core must be specified when configuring or adding the SharedRegion entry.



- SharedRegion + system heap + system gate (for protection) gives a similar look and feel for multi-core version of OS heap
- Multiple SharedRegions can be defined in different regions of memory. They may be in cacheable or non-cacheable memory. The cache characteristics are used by other modules to determine whether cache coherence should be done on buffers allocated within that SharedRegion.
- Creation of heaps is simple, just by specifying createHeap as TRUE when configuring the Shared Region. This gives similar behavior to how heap can be configured in SYS/Bios memory segment.
- Multiple heaps can be created, one in each memory region designated as a Shared Region
 - System heaps can be used to allocate/free any sized buffers from any cores in the system.
 - Usage of system heaps enables zero-copy by passing buffer pointers across cores without any copy of content
 - The system heaps are protected by system gate (multi-core protection). This gate is also used as the default gate for other modules/instances.
- The SharedRegion module is designed to be used in a multi-processor environment in which memory regions are shared and accessed across different processors. The module itself does not use any shared memory, because all module state is stored locally. SharedRegion APIs use the system gate for thread protection.
- This module creates and stores a local shared memory region table. The table contains the processor's view for every shared region in the system. The table must not contain any overlapping regions. Each processor's view of a particular shared memory region is determined by the region id. In cases where a processor cannot access a certain shared memory region, that shared memory region should be left invalid for that processor.
- A region is added by setting the SharedRegion entry. The length of a region must be the same across all processors. The owner of the region can be specified. If specified, the owner manages the shared region. It creates

a HeapMemMP instance which spans the full size of the region. The other processors open the same HeapMemMP instance.

- SharedRegion 0 is a special region which needs to be configured by the application if any IPCs are to be used. The IPCs internally use SharedRegion 0 for their shared memory needs. The application can also get a handle to the SharedRegion heap and start using it for their own requirements by using the SharedRegion_getHeap API.

Static configuration

SharedRegion configuration can be done statically by updating the CFG file for a particular slave. Using this provided static configuration, the initial table is created on the local slave processor.

On HLOS-side, information about Shared Regions is read from each slave executable when loaded from the host (Cortex A8)

- System heap (memory manager) is created/opened automatically if required in the Shared Regions
- Thus, the heap becomes usable from Linux side also

Adding dynamic entries

In addition to the static configuration, Shared Region entries can be created dynamically too. This is possible using the SharedRegion_setEntry API on both RTOS and HLOS-sides. Since SharedRegion module always works on virtual addresses, information about the SharedRegion to be added for a particular SharedRegion ID includes the virtual address of the SharedRegion entry.

The SharedRegion_setEntry API only adds the SharedRegion into the local SharedRegion information table. To enable usage from other processors also, the same API must be called on each core that shares this SharedRegion with other cores.

On each core, the virtual address on that core must be specified when adding the SharedRegion entry dynamically. On RTOS-side, this maps to the virtual memory map for that slave executable and SharedRegions. On HLOS-side, the physical address for that SharedRegion must be mapped into HLOS address space to generate a virtual address that can be passed to the dynamic SharedRegion_setEntry API. If this API is being called from kernel-side, the virtual address to be provided must be the kernel virtual address. If being called from user-side, the virtual address provided to SharedRegion_setEntry API must be the user virtual address. Also, applications must ensure that this memory region is mapped into the slave address space to map the physical memory region to the slave virtual address space, which was used to call SharedRegion_setEntry on that slave processor. SysLink provides ProcMgr_map API to enable the following mappings of a physical address range:

- Mapping to HLOS kernel space
- Mapping to HLOS user space
- Mapping to slave virtual space

Once these mappings are done, the SharedRegion_setEntry API can be called with the appropriate virtual address to set the entry dynamically.

Note: The number of entries must be the same on all processors.

MultiProc

MultiProc module provides the management of Processor ID. The Processor ID can be used by other modules that communicate with remote processors. The processors can be identified by name and their corresponding IDs retrieved.

Note: The MultiProc configuration must be the same on all processors across the system.

List

The List module supports the creation and management of local doubly linked circular lists. All standard linked list operations including creation, construction, destruction, deletion, traversal, put, get, etc. are supported by the module.

NameServer

The NameServer module manages local name/value pairs that enables an application and other modules to store and retrieve values based on a name.

- The module supports different lengths of values.
- It provides APIs to get and add variable length values.
- Special optimized APIs are provided for UInt32 sized values.
- The NameServer module maintains thread-safety for the APIs. However, the NameServer APIs cannot be called from an interrupt context.
- Each NameServer instance manages a different name/value table. This allows each table to be customized to meet the requirements of user:
 - Size differences: one table could allow long values, (e.g. > 32 bits) while another table could be used to store integers. This customization enables better memory usage.
 - Performance: improves search time when retrieving a name/value pair.
 - Relax name uniqueness: names in a specific table must be unique, but the same name can be used in different tables.
- When adding a name/value pair, the name and value are copied into internal buffers in NameServer.
- NameServer maintains the name/values table in local memory (e.g. not shared memory). However the NameServer module can be used in a multiprocessor system.
- The NameServer module uses the MultiProc module for identifying the different processors. Which remote processors and the order they are queried is determined by the procId array in the get function.

Trace

The Trace module on HLOS provides the following functionality:

- Enable entry/leave/intermediate trace prints
 - The prints can be configured at build time and run-time
- Enable prints whenever any failure occurs, which gives the exact file name, line number and function information, along with information on the cause of failure
- Enable assertions in debug build, which indicate unexpected behavior

The trace prints can be configured at build time and run-time. Details are given in a later section in this document.

Configuration

RTOS module configuration

The RTOS side modules are configured using the application CFG file. They use RTSC configuration. The default configuration can be overridden by explicitly changing the configuration through the application CFG file. For example, the number of SharedRegion entries can be modified from the default to 16 by using the following command in the CFG file:

```
var SharedRegion =  
xdc.useModule('ti.sdo.ipc.SharedRegion'); SharedRegion.numEntries = 16;
```

The full set of module configuration items are detailed in the CDOC module reference available as part of the IPC and SysLink products.

HLOS module configuration

On the HLOS, the module configuration is done from the kernel-side, by the system integrator at startup. A platform-specific file `syslink/family/knl/hlos/<Platform>/Platform.c` needs to be updated to implement the function `Platform_overrideConfig`. The configuration structure corresponding to the module level configuration items for each module are defined in a structure `<Module>_Config`, defined in file `_<Module>.h`.

The default values for all module configuration values match the defaults in the RTOS-side modules.

This needs to be used only for reference to implement the `Platform_overrideConfig` function, since the application shall never make any calls to configure the modules.

For example,

- `SharedRegion_Config` structure defined in file `_SharedRegion.h` has the configuration fields for the `SharedRegion` module on HLOS-side.

To change the number of `SharedRegion` entries to match the modified RTOS-side configuration of 16, add the following code in the `Platform_overrideConfig` function: `config->sharedRegionConfig.numEntries = 16;` The SysLink kernel module needs to be rebuilt after this change.

- `MessageQ_Config` structure defined in file `_MessageQ.h` has the configuration fields for the `MessageQ` module on HLOS-side.

To change the number of number of heaps that can be registered with `messageQ` to match with the modified RTOS-side configuration, add the following code in the `Platform_overrideConfig` function: `config->messageQConfig.numHeaps = 8; /* new value must match RTOS side */` The SysLink kernel module needs to be rebuilt after this change.

Trace, debug and build configuration

Trace logging

SysLink provides a mechanism to enable or disable different levels of trace prints on both the kernel and user side. To use trace, SysLink must be built with the compile option **SYSLINK_TRACE_ENABLE**. By default, i.e. if not explicitly set to 0 when building SysLink, this compile option is enabled for the build. If this option is not provided during build, the trace prints code is compiled out to save on code size and optimize performance. To disable this build option, explicitly pass `SYSLINK_TRACE_ENABLE=0` when invoking the make commands.

Trace types

If the `SYSLINK_TRACE_ENABLE` compile option is provided during build, three different trace settings can be provided at run-time:

- **TRACE=[0|1]** Disables or enables all trace prints
- **TRACEFAILURE=[0|1]** Disables or enables prints about failures occurring within the SysLink implementation. If this feature is enabled and a failure occurs anywhere within the code, a print is put out, which indicates a descriptive reason for failure, the exact file name and line number where the failure has occurred, along-with the failure code value. This feature can be used for isolating the exact failure cause when an error code is returned from any SysLink API, and is usually the first mechanism used for debugging.
- **TRACEENTER=[0|1]** Disables or enables entry and leave prints from each function within SysLink. When the failure is difficult to isolate or does not result in an error being returned from any SysLink function, these prints can be enabled to give full information about the call flow within SysLink.
- **TRACECLASS=[1|2|3]** Class of additional trace prints. The number of trace prints that are given out increase with increasing trace class value. i.e TRACECLASS 1 gives the least number of prints. These are for block-level or critical information. TRACECLASS 2 includes the prints for class 1 and adds more information. TRACECLASS 3 includes prints for classes 1 and 2 and adds more information for prints that are in iterative loops (e.g. names of all loaded symbols, sections etc).

Configuring trace

TRACE needs to be enabled for any of the prints to be given out. The others can be enabled or disabled independently of each other. i.e. it is possible to enable only failure prints, or failure prints and traceclass prints, etc.

On the kernel-side, the trace values can be provided as insmod parameters while inserting the kernel module.

For example:

```
insmod syslink.ko TRACE=1 TRACEFAILURE=1
```

On the user-side, the trace values can be exported on the shell as environment variables. In the application, these can be read from the environment and used to set the 'curTrace' variable. Refer to any SysLink sample application for an example of the usage.

For example:

```
export TRACE=1
export TRACEFAILURE=1
export TRACECLASS=3
```

Changing trace configuration from application

The user or kernel trace configuration can be controlled from user application through the use of the `GT_setTrace()` API. This API is available in `ti/syslink/Utils/Trace.h`

This feature is useful when the trace is to be enabled for a specific part of the user application only, so that a limited number of trace prints are seen, and the debugging can be more effective.

For example, to enable full trace for a certain section of application processing that involves SysLink, the following can be used: `UInt32 oldMask; UInt32 traceClass = 3;`

```
/* Temporarily enable full trace */ oldMask = GT_setTrace((GT_TraceState_Enable |
```

```
    GT_TraceEnter_Enable |
    GT_TraceSetFailure_Enable |
    (traceClass << (32 - GT_TRACECLASS_SHIFT))),
GT_TraceType_Kernel);
```

```
/* Processing here */
```

```
/* Restore trace to earlier value */ GT_setTrace(oldMask, GT_TraceType_Kernel);
```

To enable user-side, use `GT_TraceType_User` instead of `GT_TraceType_Kernel` in the above. To modify both user and kernel-side trace, `GT_setTrace()` needs to be called twice, once for each user and kernel-side.

Debug Build Configuration

In addition to trace logging, SysLink also has a mechanism to display assertion prints when an unexpected condition is seen. The assertion prints display information about the failed condition and the line and file number where the issue is seen. These checks are present for failure conditions which are not expected in normal run, and usually indicate incorrect behavior. To enable assertion prints, SysLink must be built with the compile option **SYSLINK_BUILD_DEBUG**. By default, i.e. if not explicitly set to 0 when building SysLink, this compile option is enabled for the build. If this option is not provided during build, the assertion condition checks and prints are compiled out to save on code size and optimize performance. To disable this build option, explicitly pass `SYSLINK_BUILD_DEBUG=0` when invoking the make commands.

Configuring an optimized build

SysLink provides a mechanism to substantially optimize the SysLink code through a compile option: **SYSLINK_BUILD_OPTIMIZE**.

By default, this option is not enabled for the SysLink build. To ensure run-time stability and avoid kernel/user crashes, SysLink code has checks for error conditions, invalid parameter checks and intermediate failures. These are useful during application development, to ensure that the user does not need to reboot the hardware device due to incorrect usage of SysLink. However, once the application development is complete and the product is being deployed, these checks can be removed. SysLink provides a mechanism to perform such optimization through the **SYSLINK_BUILD_OPTIMIZE** compile time flag. If this flag is provided during SysLink build, all the parameter checks and checks for error conditions are stripped out during the build. This reduces code size, as well as substantially enhances performance. However, note that once the checks are removed, any run-time errors shall not get handled, and result in kernel/user crashes.

Interpreting SysLink status codes

Most SysLink module APIs return status codes to indicate whether they were successful, or if there was a failure during execution.

At run-time, applications can check for success of SysLink APIs by checking the status codes. The status codes are signed 32-bit integers. Positive status codes indicate success and negative codes indicate failures. To check whether an API call was successful, the status code can be checked for a value greater than or equal to zero.

In general, module success codes are named as `<MODULE>_S_<STATUSCODE>` and failure codes are named as `<MODULE>_E_<STATUSCODE>`. For example:

```
Int status = 0; status = SysLink_api(param1, param2); if (status >= 0) { printf ("The API was successful. Status [0x%x]", status); } else { printf ("The API failed! Status [0x%x]", status); }
```

The SysLink API reference document and header files contain information about critical specific success or failure codes returned by the APIs. Some modules return specific failure codes indicating a particular behavior, which may need to be specifically checked by applications. For example, if a MessageQ instance is not found to be created when `MessageQ_open` is called, it returns `MessageQ_E_NOTFOUND` code. In such cases, applications can specifically check for this code. For example:

```
do { status = MessageQ_open (msgqName, &MessageQApp_queueId [info->procId]); } while (status == MessageQ_E_NOTFOUND);
```

In most other scenarios, however, unless explicitly mentioned, checking for a `(status >= 0)` is advised. In many cases modules return more than one possible success code. Hence, an explicit check against `MessageQ_S_SUCCESS` may fail, however, this does not indicate failure. Checking status code to be a positive value will ensure correct behavior.

Build

HLOS Build system

- The HLOS build system exports the device-specific kernel modules (syslink as well as sample application kernel modules) to `$(SYSLINK_ROOT)/bin/<PLATFORM>`. This enables side-by-side build for multiple devices to generate the syslink kernel modules.
 - syslink kernel module exports to `$(SYSLINK_ROOT)/bin/<PLATFORM>`.
 - sample application kernel modules export to `$(SYSLINK_ROOT)/bin/<PLATFORM>/samples`.
- The HLOS build system exports the device-independent library module to `$(SYSLINK_ROOT)/lib`.
- The HLOS build system exports the device-specific sample application user modules to `$(SYSLINK_ROOT)/bin/<PLATFORM>/samples`. This enables side-by-side build for multiple devices to generate the syslink kernel modules.
- The HLOS build system generates intermediate files for library module to `$(SYSLINK_ROOT)/obj/.objs/usr`.
- The HLOS build system generates intermediate files for sample application user modules to `$(SYSLINK_ROOT)/obj/<PLATFORM>/samples/<SAMPLE_APP>`.
 - Except an intermediate file for ProcMgr module, which is getting generate at `$(SYSLINK_ROOT)/obj/<PLATFORM>/samples`.

Build Options (For User Modules)

- There are three different build-options for library and sample application user modules:
 - Debug & Release - both profile (**Default**)
 - Debug
 - Release

Note: For more information on build commands, please refer to platform specific Install Guide ^[1]

Clean Options (For Kernel Modules)

- There are two clean options in HLOS build system for device-specific kernel modules (syslink as well as sample application kernel modules).
 - **clean:** It removes all intermediate files.
 - **cleanall:** It removes all intermediate files and exported kernel module(s).

Note: To build kernel modules for many devices, we need to clean and then build the system for new the device.

Clean Options (For User Modules)

- There are four clean options in HLOS build system for device-specific user modules (library as well as sample application user modules).
 - **clean**: It removes all intermediate files, which was build with **debug & release** build-option.
 - **cleandebug**: It removes all intermediate files, which was build with **debug** build-option.
 - **cleanrelease**: It removes all intermediate files, which was build with **release** build-option.
 - **cleanall**: It removes all intermediate files for **all** build-options and their exported executables.

Miscellaneous

- The SysLink HLOS build system enables passing a semi-colon separated list of include paths for the build through the `SYSLINK_PKGPATH` variable. For example, the following can be specified to the kernel make command to override the defaults in the base Makefile.inc file.
 - `make ARCH=arm`
`CROSS_COMPILE=/toolchains/git/arm-2009q1-203/bin/arm-none-linux-gnueabi-`
`SYSLINK_PKGPATH="/toolchains/IPC/ipc_<version>/packages;$HOME/syslink"`
 - To do a verbose build where the make commands are printed out, use `V=1` when running make.

Note: Replace `ipc_<version>` above with the specific IPC versioned package name

References

- [1] http://ap-fpdsp-swapps.dal.design.ti.com/index.php/SysLink_02.00.00.67_alpha2_InstallGuide

Article Sources and Contributors

SysLink 02.00.00.68 beta1 UserGuide *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=90727> *Contributors:* DeepaliUppal, Goutamkumar, MugdhaKamoolkar

Image Sources, Licenses and Contributors

Image:SysLinkScope.png *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:SysLinkScope.png> *License:* unknown *Contributors:* MugdhaKamoolkar

Image:SysLinkIpcProducts.png *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:SysLinkIpcProducts.png> *License:* unknown *Contributors:* MugdhaKamoolkar

Image:SysLinkSysBiosModuleArchitecture.png *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:SysLinkSysBiosModuleArchitecture.png> *License:* unknown *Contributors:* MugdhaKamoolkar

Image:SysLinkHlosModuleArchitecture.png *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:SysLinkHlosModuleArchitecture.png> *License:* unknown *Contributors:* MugdhaKamoolkar

Image:SysLinkProcMgrArchitecture.png *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:SysLinkProcMgrArchitecture.png> *License:* unknown *Contributors:* MugdhaKamoolkar

Image:SysLinkGateMP.PNG *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:SysLinkGateMP.PNG> *License:* unknown *Contributors:* MugdhaKamoolkar

Image:SysLinkSharedRegion.PNG *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:SysLinkSharedRegion.PNG> *License:* unknown *Contributors:* MugdhaKamoolkar