DSP/BIOS™ LINK

Version 1.40

LNK 017 PRT

This page has been intentionally left blank

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

This page has been intentionally left blank.

TABLE OF CONTENTS

A. INTRODUCTION 7

1 Purpose 7

2 Terms & Abbreviations 7

3 References 7

4 Overview 8

4.1 DSPLink architecture 8
4.2 System Organization Types 10
4.3 Current ports of DSP/BIOS™ LINK 12

5 Porting to a different GPP Operating System 13

5.1 Requirements of OSAL sub-components 13
5.2 Important considerations 17
5.3 Checklist of activities involved 18

6 Porting to a different Hardware Platform 20

6.1 Port to a platform with similar GPP-DSP combination with an existing physical link implementation 20
6.2 Porting to a different physical link for existing platform(s) 26
6.3 Porting to a different physical link for a different platform 31

7 Build system 32

8 Testing the port 35

8.1 Quick testing 35
8.2 Full validation 35

9 Packaging 35

TABLE OF FIGURES

Figure 1. DSP/BIOS™ LINK Architecture 8
Figure 2. Shared Memory System Organization..... 11
Figure 3. Memory Mapped IO System Organization 11
Figure 4. Fully Discrete System Organization..... 12
Figure 5. Interactions of DRV 13

A. INTRODUCTION

1 Purpose

This document describes the changes that a user needs to make in the DSP/BIOS™ LINK sources for porting DSPLink to another GPP-side Operating System or Hardware platform.

OEMs and software developers involved in the porting activity of DSP/BIOS™ LINK are the target audience of this document.

2 Terms & Abbreviations

OSAL	Operating System Adaptation Layer
HAL	Hardware Abstraction Layer
DSPLink	DSP/BIOS™ LINK

3 References

1.	DSP/BIOS™ LINK User Guide
2.	DSP/BIOS™ LINK LNK_024_DES: OS Adaptation Layer for Linux
3.	DSP/BIOS™ LINK LNK_012_DES: Link Driver
4.	DSP/BIOS™ LINK LNK_041_DES: Zero Copy Link Driver

4 Overview

4.1 DSPLink architecture

DSP/BIOS™ LINK is comprised of the following major components:

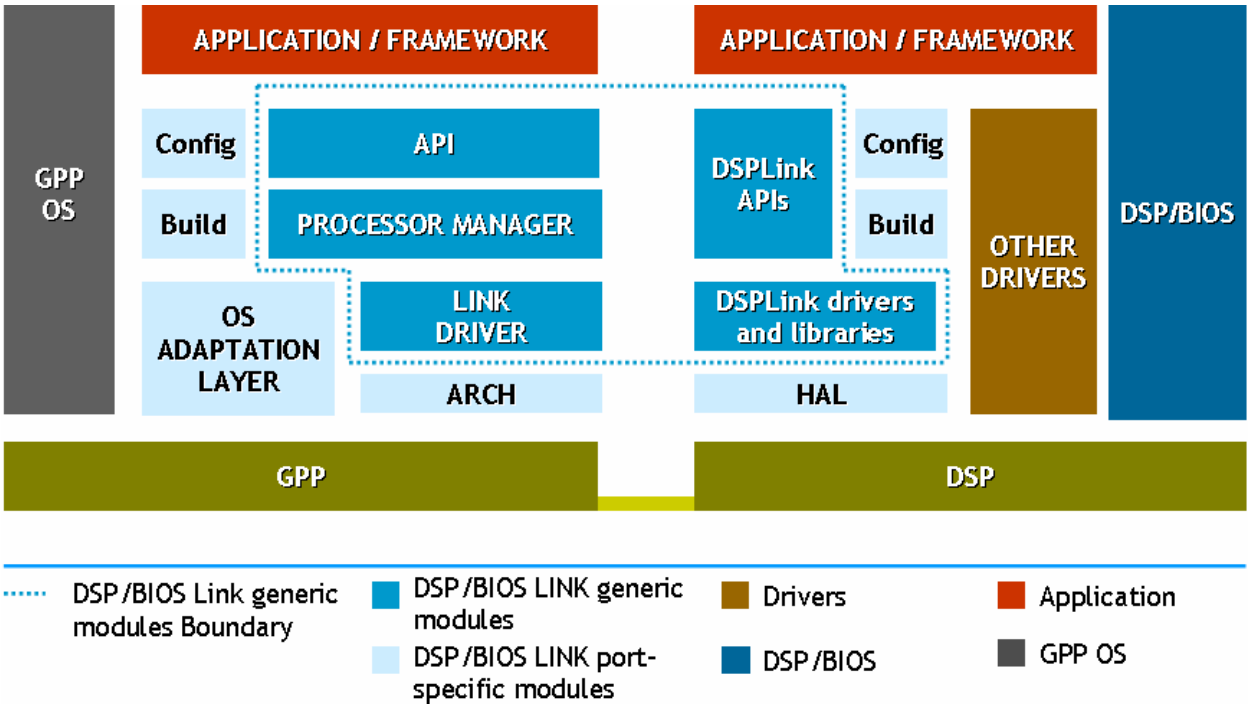

Figure 1. DSP/BIOS™ LINK Architecture

4.1.1 Common modules

This section provides a brief summary of the components shown on the GPP side in the figure above. The architecture is described in more details later in the document.

- **CONFIG**

  This component provides configuration for DSPLink. When any new device or Operating System support is to be added to DSPLink, the configuration needs to be accordingly modified.

- **BUILD**

  DSPLink provides a generic make-based build system. The build system is portable to different Operating Systems or devices. Accordingly, it must be ported as required to ensure correct build of DSPLink for the target environment.

4.1.2 GPP side

This section provides a brief summary of the components shown on the GPP side in the figure above. The architecture is described in more details later in the document.

- **GPP OS**

  On the GPP side, a specific OS is assumed to be running.

- **OS ADAPTATION LAYER**

This component encapsulates the generic OS services that are required by the other components of DSP/BIOS™ LINK. This component exports a generic API that insulates the other components from the specifics of an OS. All other components use this API instead of direct OS calls. This makes DSPLink portable across different operating systems.

- **ARCH**

  The ARCH component contains all device-specific functionality. Whenever DSPLink is to be ported to a different device, this is the main component that needs to be ported.

  Each device must provide an implementation of the standard DSP interface. The generic DSP interface abstracts the hardware specific functions for accessing (communicating with) the target. The interface exports low level function to control the DSP e.g. start, stop, read, write, interrupt, etc.

- **LINK DRIVER**

  This component encapsulates the low-level control operations on the physical link between the GPP and DSP. This module is responsible for controlling the execution of the DSP and data transfer using defined protocols across the GPP-DSP boundary. This module provides physical transports for common configurations of the system.

- **PROCESSOR MANAGER**

  This component manages the key objects visible to the user. It also verifies the necessary access to these objects by the clients.

  It utilizes the services of LINK DRIVER to perform the low level operations for controlling the DSP and data transfer.

- **DSP/BIOS™ LINK API**

  This component is an interface for all clients on the GPP side. For kernel-side modules (e.g. CHNL, MSGQ, POOL), this is a very thin component and usually doesn't do any more processing than parameter validation. After parameter validation, corresponding functions in the PROCESSOR MANAGER layer are invoked.

  The thin API layer allows easy partition of DSP/BIOS™LINK across the user kernel boundary on specific operating systems e.g. Linux. Such partition may not be necessary on other operating systems.

  In addition to the kernel-side modules, the user-side modules such as RingIO, MPCS, MPLIST and NOTIFY are implemented in user space.

4.1.3 DSP side

- **DSP OS**

  On the DSP side, DSP/BIOS is assumed to be running. Since only DSP/BIOS OS is supported, no OSAL is required.

- **HAL**

  On the DSP-side, HAL layer contains all device specific implementation. This needs to be implemented when a new device support is to be added.

- **DSPLink drivers and libraries**

This component encapsulates the low-level control operations on the physical link between the GPP and DSP. This module is responsible for data transfer using defined protocols across the GPP-DSP boundary. This module provides physical transports for common configurations of the system.

- **DSP/BIOS™ LINK API**

  This component is an interface for all clients on the DSP side.

  Message transfer is done using the DSP/BIOS™ MSGQ module. The APIs in this case are part of DSP/BIOS MSGQ module, and DSPLink simply provides the underlying Message Queue transport implementation that plugs into the DSP/BIOS MSGQ module.

  DSPLink also implements POOL instances that plug into the POOL interface provided by DSP/BIOS. DSP/BIOS provides POOL configuration. However, APIs for usage of the POOL are provided by DSPLink.

  Similarly, the driver for data transfer using CHNL module is an implementation of the IOM driver interface in DSP/BIOS. This driver allows users to use SIO/GIO APIs in DSP/BIOS on the DSP-side to communicate with CHNL module on GPP-side.

  APIs for other modules such as MPCS, MPLIST, NOTIFY and RingIO are provided as DSPLink APIs on the DSP-side.

4.2 System Organization Types

There are three basic types of System Organization with which DSPLink is concerned:

- Shared Memory
- Memory mapped IO
- Fully discrete

There may be multiple combinations of any of the above types within a final system. Each type of system organization requires a different handling within DSPLink. All types may exist within a single system with some processors sharing memory and some processors not sharing memory.

4.2.1 Shared Memory

The Shared Memory System Organization is shown below:

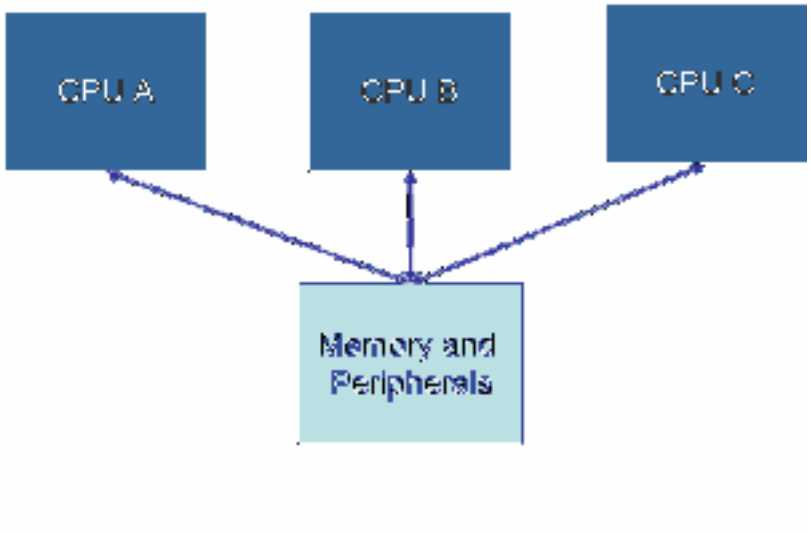

Figure 2. Shared Memory System Organization

Processors that share memory can pass pointers to the shared memories to enable efficient zero-copy transfer protocols. Usage of shared memory usually simplifies implementations of physical transports between the processors and also enables additional features that may not be possible on any of the other system types.

4.2.2 Memory Mapped IO

The Memory Mapped IO System Organization is shown below:

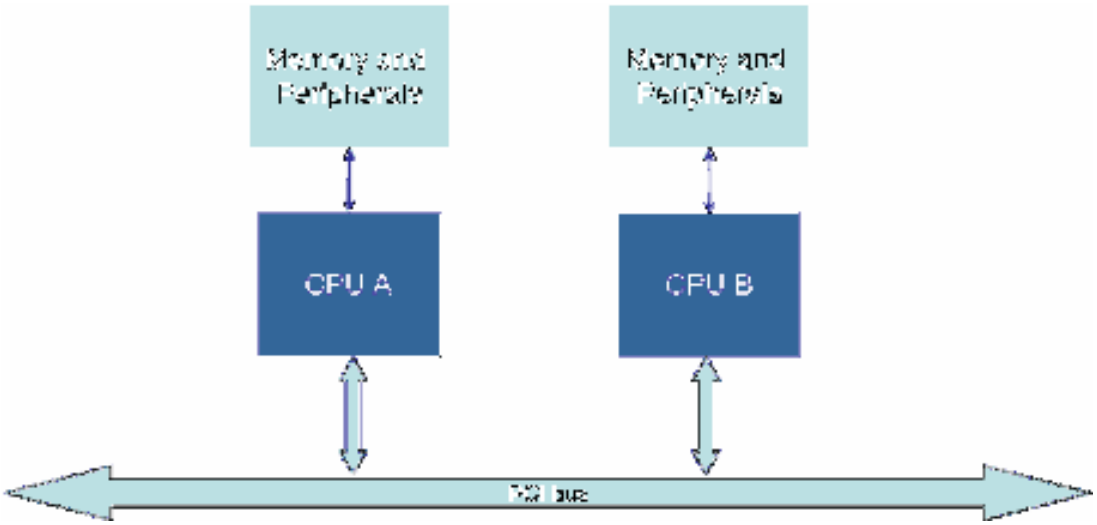

Figure 3. Memory Mapped IO System Organization

Devices that are connected using PCI, VLYNQ etc. provide memory mapped access to the host processor into the memory on the other processor. While this is similar to shared memory, it is not efficient to transfer large data buffers using byte-by-byte write into the remote memory locations. For this, it is more efficient to maintain buffer pools on both processor, and use DMA to keep them synchronized. In software IPC, this is known as a synchronized pool of memory.

In such systems, the memory mapped addresses are used for control structure operations and DMA is used for buffer transfers.

4.2.3 Fully Discrete

The Fully Discrete System Organization is shown below:

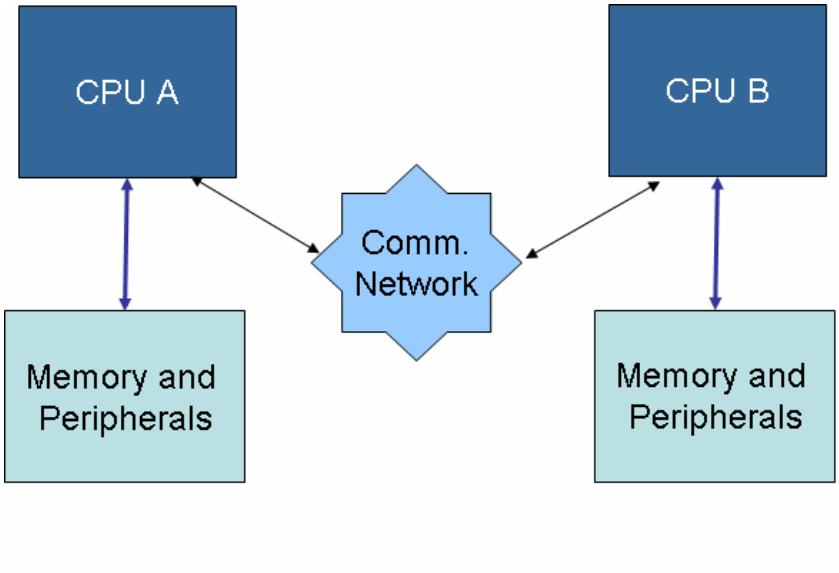

Figure 4. Fully Discrete System Organization

In fully discrete systems, no shared memory is available, and all IPC operations must be packet based. Such systems warrant a different architecture where the physical transport for all IPCs must be able to work off packet-based IO. For example: ethernet connection between two devices.

4.3 Current ports of DSP/BIOS™ LINK

DSP/BIOS™LINK is architected for easy portability to other hardware platforms and operating systems. It can be ported to either of the following:

- A different operating system
- A different hardware platform
- A different physical link
- Any combination of above

Currently DSP/BIOS™LINK is available on devices as mentioned in the Release Notes document.

DSPLink also provides support for Shared Memory and Memory Mapped IO system organizations. Physical transports for these two types of System Organizations are provided along with DSPLink releases, enabling easy port to systems using any of these configurations. Currently, DSPLink does not provide physical transports fro fully discrete systems.

This document discusses steps required to port DSP/BIOS™ LINK. The details on each subcomponent can be obtained from the corresponding design document.

5 Porting to a different GPP Operating System

The DSPLink architecture is designed for portability, and ensures that the higher layers of DSP/BIOS™ LINK use OSAL for operating system specific services. This allows these higher layers to be insulated from the subtleties of different operating systems as long as an appropriate port of OSAL is available on the target operating system.

Also, some operating systems are aware of the DSP on the target hardware platform and they configure the "working environment" of DSP. In such cases, adapt the DSP component to the configurations of the operating system.

Each operating system has its own way of identifying and providing the services expected from these sub-components. One of the core responsibilities of these sub-components is to abstract such subtleties and provide a uniform interface to the upper layers.

5.1 Requirements of OSAL sub-components

5.1.1 DRV

This sub-component provides encapsulation for the driver model that the operating systems require. Some operating systems differentiate between the user and the kernel space and certain others don't.

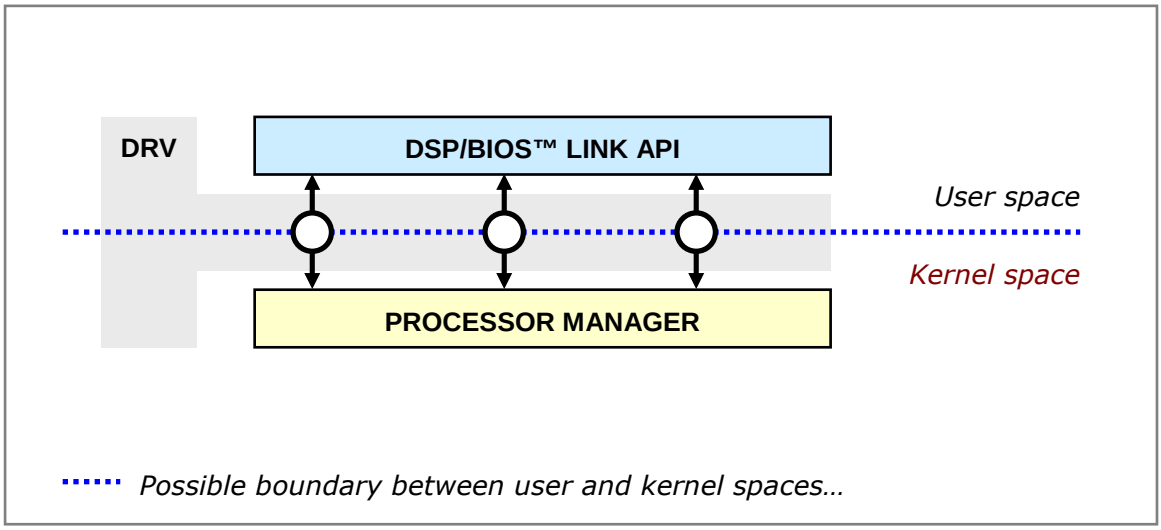

Figure 5. Interactions of DRV

The resources required by DSP/BIOS™ LINK from the driver model may vary across different platforms/ operating systems.

This sub-component is expected to act as the 'glue' between different layers of DSP/BIOS™ LINK allowing greater portability across such operating systems.

5.1.2 DPC

This sub-component provides services to defer the execution of non-critical code from an interrupt context. Typically, the OS achieves this by scheduling a high priority thread of execution from the interrupt context.

Many operating systems have a restriction on when such a thread can be scheduled and do not recognize another call to schedule the thread if it is already in a running

state. In such cases, one of the core requirements of this sub-component is to ensure that when the OS schedules the thread to run, all the pending requests to the DPC get serviced.

- *The DPC must always execute at a higher priority than all other tasks. If implemented as a task, the task must be of the highest priority in the system.*

5.1.2.1 Key functionality

DPC creation/deletion

The DPC module supports creation and deletion of multiple DPC objects. Each DPC object corresponds to one Deferred Procedure Call. Within DPCs, there is no prioritization. All DPCs can run at the same priority; however, the DPC priority must be higher than all other tasks in the system.

It must be possible for users of the DPC module to register an optional argument with each DPC creation. This argument shall be returned back to the user within each callback function as a parameter. If the Operating System inherently does not support such functionality, it must be implemented within the DPC OSAL module.

DPC schedule

The DPC module provides functionality to schedule a specific DPC as identified by the DPC object handle. The schedule implementation must ensure that the DPC executes for the number of times the schedule API was called. If the Operating System does not directly support this feature, this concept of pending count must be added within the DPC OSAL module implementation.

In some cases, there is a possibility of thrashing if the DPC is scheduled with the OS when DPCs are disabled. If this is the case, the DPC schedule call must ensure that it checks for the disabled state of DPC and only schedules with the OS if the DPCs are enabled. Also, in such cases, the enable API must ensure that any pending DPCs get scheduled.

DPC callback

In case the Operating System provides all functionality needed for the DPC OSAL module, no abstraction for the DPC callback may be needed over the user-defined function. The user's callback function can be directly registered with the OS. However, in some cases, where the OS does not provide the full functionality as needed by the OSAL DPC module, the DPC callback function may need to be defined within the DPC module. This function would be responsible for making all user-defined callbacks.

DPC enable/disable

The DPC enable/disable functionality is provided for users requiring protection from all DPCs.

When a DPC is created, it must start off in the enabled state.

DPC cancel

The DPC cancel function cancels all pending schedules. On some Operating Systems, this is required in order for the DPC deletion to succeed. The call may also block till pending schedules are completed, and the behavior is Operating System dependant.

5.1.3 ISR

This sub-component provides services to hookup and service hardware interrupts.

5.1.4 KFILE

Certain versions of operating systems do not have their own file system and certain others do not allow access to the file system from a device driver. This sub-component takes care of such issues and provides a uniform interface to open, close and read files.

5.1.5 MEM

Certain operating systems distinguish between the different kinds of memory that are available in the system. This sub-component takes care of such differences and provides a uniform interface to allocate and free memory.

This module also provides APIs to map and unmap memory as specified within the memory table in the dynamic configuration.

5.1.6 NOTIFY_KNL

The NOTIFY module provides capability to send and receive notifications between the GPP and the DSP. On operating systems with user-kernel boundary, the NOTIFY_KNL component provides services to forward notifications received from the DSP to the user-space process/thread.

5.1.7 PRCS

This sub-component provides services to identify and compare different threads of execution.

5.1.8 PRINT

The facility to display debug messages on the user console is provided by this sub-component. Typically, it is a wrapper over the OS/hardware specific call to 'print' a string to the console.

5.1.9 SYNC

This sub-component provides the following services:

- Facility to synchronize multiple threads of execution
- Facility to provide mutually exclusive access to key data structures/code from multiple threads of execution

5.1.9.1 Key functionality

Semaphores

The SYNC module supports creation, deletion and usage of semaphores. It supports two types of semaphores:

- Counting semaphores
- Binary semaphores

The SYNC Semaphore implementation must make use of the most suitable features provided by the Operating System to implement these. Counting semaphores maintain a count of the number of signalSEM calls and accordingly unblock the number of waitSEM calls correspondingly. This is equivalent to making available/consuming a number of resources. On the other hand, binary semaphores can only have a count of 1 or 0. Even if multiple signalSEM calls are made, the

binary semaphore stays at count 0. The waitSEM resets the count to 0 after consuming the resource.

The semaphore implementation requires support of a timeout mechanism. The waitSEM call must return with error if the semaphore does not become available even though the specified timeout has elapsed. The timeout specified to the API is in a standard of milliseconds. The SYNC semaphore implementation may need to convert this to the OS-specific measures. If the OS semaphore call does not support timeout, this feature needs to be implemented separately in SYNC module, possibly through usage of a timer. It is also possible to specify no-wait (SYNC_NOWAIT) or infinite-wait (SYNC_WAITFOREVER) as special values for timeout, and the SYNC implementation must support these.

Events

The SYNC module supports creation, deletion and usage of events. The basic concept of an event is that it stays set until it is explicitly reset through an API call. Waiting on an event and getting unblocked does not automatically reset the event. Any further calls to wait on the same event would also get unblocked till there was an explicit call to reset the event.

The basic functionality provided with events is:

- Set event: This function sets the event to indicate that the event is available.
- Reset event: This function resets the event to indicate that the event is not available.
- Wait on event: This function waits on the event till a specified timeout till the event becomes available. It does not automatically reset the event even though its wait may be successful. The considerations for timeout are similar to those mentioned in the semaphore section above.
- Wait on multiple events: This function is for waiting on any one of multiple events to be available. This is not currently used within DSPLink and may remain unimplemented.

Critical Sections

The SYNC module also supports creation, deletion and usage of critical sections. The critical sections are similar to binary semaphores. However, the main difference is that the critical section must be initialized to 'available' when it is created, whereas the binary semaphore count must be initialized to 'not available' when created. Also, the critical section APIs are much simpler for entering and leaving the critical sections, and do not need to support timeout.

Spinlocks

The SYNC module supports creation and usage of spinlocks. These provide the highest level of protection in the system. They are expected to disable interrupts and spin till the spinlock is available. They are expected to be callable from any context: task, DPC or ISR. The default implementation of spinlock APIs provides one global spinlock, which can be used in a simple way by just making calls to start and end the spinlock. In addition, where there is a possibility of nesting spinlocks, additional extended APIs are provided, where different spinlock objects can be created and used for the same functionality.

SYNC protection

The SYNC module also provides APIs for SYNC protection. These disable tasks and DPCs and provide a different level of protection where only protection from tasks and DPCs is needed, and interrupts do not need to be disabled.

5.2 Important considerations

The port to any new GPP operating system must take a few important aspects into account. These may or may not be applicable based on the GPP OS to which the port is being made.

5.2.1 Non-cached GPP-side shared memory

On the GPP-side, a certain amount of memory is reserved to ensure that the GPP OS does not use it. The following two types of memory fall into this category:

- **Shared Memory:** This is the memory that is used as the shared region between GPP and DSP. All the control structures for the *DSPLINK* modules, including POOL, and any other memory regions reserved for writing into or reading from DSP memory fall into this category.
- **DSP executable memory:** This is the memory that corresponds to the DSP executable's memory map. The code and data sections in the DSP executable are placed into either DSP external memory or internal memory. The DSP internal memory is accessible separately from GPP-side. However, any external memory used for loading the DSP code & data needs to be reserved on the GPP-side.

Both of the above types of memory regions must be reserved from the GPP OS. The way that this is done depends on the GPP OS. On the DSP-side, the similar kind of reservation is done through TCF file to ensure that linking does not place any DSP-side code/data into any other memory regions.

For example:

- **Linux:** On Davinci Linux, in the default configuration, MEM=254M may be used as part of `bootargs` in u-boot while booting up the ARM. This indicates that 2M (out of the 256M) is reserved, and only 254M is available to the ARM Linux. Out of this 2M, 1M is reserved as the shared memory (DSPLINKMEM) and 1M as the DSP executable memory (DDR). Internally, OSAL MEM module for Linux provides APIs to get virtual addresses corresponding to the physical addresses reserved for these memory regions. ***IMPORTANT: The API used must ensure that physically contiguous non-cached memory regions are returned.*** On Linux, the APIs used for this are: `ioremap_nocache` and `iounmap`.
- **PrOS:** When TI tools are used, the linker command file is used to reserve regions of memory and ensure that PrOS sections do not get placed into the reserved regions. A specific assembly file within the LSP may need to be updated to ***ensure that the reserved regions are non-cached memory***.

5.2.2 Physical to virtual address mapping and translation

Certain GPP Operating Systems such as Linux work across a user-kernel boundary. In such cases, the physical memory is not directly accessible in user space. Shared memory is directly reserved as physical memory in *DSPLINK*.

The following memory mappings are required:

- Physical to kernel virtual

- Physical to user virtual

Once the mappings have been made, the *DSPLINK* DRV module needs to maintain all such mappings to ensure that it can convert addresses as required between any of physical, kernel, user and DSP address spaces.

This section details the mappings to be maintained, and uses the Linux port as a reference.

Creating mappings

The ***physical -> kernel virtual address mapping*** is done as mentioned in the previous section.

Physical -> user mappings are made on Linux using the `mmap` and `munmap` calls. These are made within the DRV module, and map the memory of provided physical address and size to a user virtual address, which can be directly used to access the shared memory in the context of the user process.

A point to note is that `mmap` requires the specified physical addresses to be aligned to `PAGE_SIZE`. The mappings must accordingly be modified to take this into account.

For the POOL, this is done within `POOL_open` API within the user-space DRV module.

Maintaining mappings

The mappings between different address spaces for the POOL are maintained within the DRV_POOL module in user space. The mapping is created in `POOL_open` and maintained separately for all POOLs in each process space.

Using the mappings

The mapping between the different address spaces is used for translation of POOL addresses as required between user virtual, kernel virtual, physical and DSP address spaces.

5.3 Checklist of activities involved

5.3.1 Updating sources

1. Port the OSAL component to the new GPP OS.
2. If DSP/BIOS™ LINK is divided across the user/kernel boundary then DRV module needs to be adapted.
3. If DSP/BIOS™ LINK is not divided across the user/kernel boundary then DRV module can be implemented to glue the API component directly to the Processor Manager. This would mean minimal change to the implementation in both these components.
4. If a different version/variant of a supported operating system is to be added, then the directory for the version can be made within the OS-specific directory and files specific to the version can be added in it.

5.3.2 Updating configuration

Update the DSPLink configuration file dsplinkcfg.pl to ensure that the configuration displays and handles the correct options for the new Operating System also. The dsplinkcfg.pl perl script generates a file CURRENTCFG.MK, which is used by the DSPLink build system for understanding the system configuration being used.

5.3.3 Updating build system

1. Update the MAKE system with rules specific for new OS and corresponding tool chain. Create a new distribution file if required.
2. Create the file - COMPONENT – for each component including the sample applications.

- *For relevant details of the MAKE system, refer to the section 7 on MAKE system in this document.*

5.3.4 Other steps

1. Look for folders within DSPLink that are identified by the Operating System name for currently supported Operating Systems. A corresponding folder would be required to be added for the new Operating System also.
2. Look for defines within the sources that indicate any OS-specific code within the implementation (e.g. OS_PROS)

5.3.5 Preliminary Testing

The sample applications provided with the DSP/BIOS™ LINK release can be used for initial validation of the port to the different GPP operating system. Each sample application demonstrates a different protocol. Successful execution of all samples will ensure validation of most of the OSAL functionality.

5.3.6 System Testing

Once all the sample applications are working as expected, the DSP/BIOS™ LINK test suite product can be used to validate the complete functionality of the port. Executing the test-suite and resolving any issues found shall increase the confidence in the integrity and stability of the port.

6 Porting to a different Hardware Platform

Porting to a different hardware platform requires varying degrees of effort depending on the differences between the new platform and existing supported platforms.

The table below gives some examples of the types of ports and their relative complexity.

GPP	DSP	Physical link	Complexity	Example
Same	Same	Same	Low	ARM9 + C64x+ (DM6446, DM6467 etc.) Zero copy physical link
Similar	Same	Same	Low	ARM9 + C64x+ (DM6446), ARM11 + C64x+ (Raven) Zero copy physical link
Same	Different	Same	Medium	ARM + C64x+ (DM6446), ARM + C55x (OMAP5912) Zero copy physical link
Same	Same	Different	Medium	Zero-copy / Processor copy physical link on OMAP5912
Same	Different	Different	Medium/High	VLYNQ physical link for ARM + DM6437. Zero-copy protocol
Same	Different	Different	High	UART physical link. Packet-based processor-copy protocol

As given in the above table, the complexity of a platform port depends on the degree of variance between an existing port and the platform to be ported to.

The following sections detail the activities involved for different types of ports.

6.1 Port to a platform with similar GPP-DSP combination with an existing physical link implementation

6.1.1 Example

- Existing platform support: Davinci platform (ARM + C64x+)
- New platform support: DRA44x (ARM + C64x+).

The ARM and DSP on both platforms are the same. The physical link is also shared memory based. This is a low-complexity effort, only requiring changes mostly localized to configuration, HAL and DSP modules.

6.1.2 Overview

When porting DSP/BIOS™ LINK to a different hardware platform for this scenario, consider the:

1. Configuration of the target DSP and its working environment for DSP/BIOS™ LINK

2. Communication between GPP and DSP

6.1.3 Changes to source code

6.1.3.1 GPP-side:

All device specific files are contained within the ARCH component.

The HAL sub-components within the ARCH component provide a set of functionality that configures the hardware to make the DSP reachable from the GPP. It also consists of functions to read from and write into the target DSP's memory space and send/receive interrupts between the processors.

The implementation of the DSP interface for each device within the ARCH component uses these services to provide a uniform interface to the higher layers of DSP/BIOS™ LINK. Consider the following changes for the components when porting DSP/BIOS™ LINK to a different hardware platform.

ARCH component

Location:

- `/$(DSPLINK)/gpp/src/arch/<Device name>`

Example:

- `/$(DSPLINK)/gpp/src/arch/DM6446GEM`

The ARCH component contains all device-specific implementation.

<Device>.c

Location:

- `/$(DSPLINK)/gpp/src/arch/<Device name>.c`

Example:

- `/$(DSPLINK)/gpp/src/arch/DM6446GEM/dm6446gem.c`

This file provides the implementation of the DSP interface. It gives the top-level device-specific interface to the rest of DSPLink. This component is responsible for configuring the hardware to make DSP reachable from GPP. It provides services to start, stop, idle and interrupt the DSP. It also manages the DSP resources to provide services to read from and write into DSPs memory space.

An existing implementation can be used as reference.

Device-specific MAP file

Location:

- `/$(DSPLINK)/gpp/src/arch/<Device name>_map.c`

Example:

- `/$(DSPLINK)/gpp/src/arch/DM6446GEM/dm6446gem_map.c`

This file provides the mappings of function table interfaces/function pointers and other such information to the modules as provided in the DSPLink dynamic configuration (`$(DSPLINK)/config/all/CFG_<Device>_<Physical Link>.c`) provided in user-space.

An existing implementation can be used as reference.

HAL implementations

Location:

- /$(DSPLINK)/gpp/src/arch/<Device name>_hal<type>.c

Examples:

- /$(DSPLINK)/gpp/src/arch/DM6446GEM/dm6446gem_hal.c
- /$(DSPLINK)/gpp/src/arch/DM6446GEM/dm6446gem_hal_boot.c

This component provides functions to configure these hardware parameters:

- Clock settings and DDR/EMIF initialization (if required). This may not be required if the setting has already been done by a boot-loader.
- Interrupt services
- Power up and down the DSP
- DSP boot configuration and reset control
- MMU (in case the DSP has an MMU)
- Any other platform specific hardware configuration

Make the appropriate changes based on the target platform.

6.1.3.2 DSP-side

On the DSP-side, the following changes are required to add support for the new platform:

HAL:

Location:

- /$(DSPLINK)/dsp/src/base/hal/DspBios/<Device name>

Example:

- /$(DSPLINK)/dsp/src/base/hal/DspBios/DM6446GEM/hal_interrupt.c

This component provides functions to configure these hardware parameters:

1. DSP data cache
2. Interrupt services

An existing implementation can be used as reference.

TCI configuration file:

Location:

- /$(DSPLINK)/dsp/inc/DspBios/<DSP OS version>/<Device name>/dsplink-<Device name>-base.tci

Example:

- /$(DSPLINK)/dsp/inc/DspBios/5.XX/DM6446GEM/dsplink-dm6446gem-base.tci

A base TCI configuration file is provided for each platform as part of the DSPLink port. This file contains the configuration specific to the platform as needed for the execution of any *DSPLINK* applications. It includes the following information:

1. Platform/device information
2. Memory configuration

3. Cache configuration
4. If CHNL module is also used, TCI files also need to be provided with DIO and IOM driver instances. Any existing platform TCI file for this can be used as a reference.

An existing TCI file can be used as reference.

platform.h:

Location:

- `/$(DSPLINK)/dsp/inc/DspBios/<DSP OS version>/<Device name>/platform.h`

Example:

- `/$(DSPLINK)/dsp/inc/DspBios/5.XX/DM6446GEM/platform.h`

The platform.h file contains device specific information that is required by DSPLink. This includes:

1. Definition of memory segment ID used by DSPLink for dynamic memory allocations.

An existing header can be used as reference.

Changes in sample applications:

Location:

- `/$(DSPLINK)/dsp/src/samples/<Sample app>/DspBios/<DSP OS version>/<Device name>`

Example:

- `/$(DSPLINK)/dsp/src/samples/message/DspBios/5.XX/DM6446GEM`

Platform-specific TCF files need to be added for each sample application. Usually the TCF files for the existing platform can be copied and used for this. Changes may also be required in linker command file (if present).

6.1.4 Other Changes

6.1.4.1 Configuration

Location:

- `/$(DSPLINK)/config/all`

A new configuration file needs to be provided for the platform to which DSPLink is being ported. The name of the configuration file depends on the device and physical link between the connected devices. For example, if the device is DM6446GEM and the physical link between the GPP (ARM) and the physical link is SHMEM (shared memory), the configuration file name is `CFG_DM6446GEM_SHMEM.c`.

A separate configuration file is provided for the GPP also. Example: `CFG_ARM.c`

Based on the number of differences between an existing platform and the target platform for the port, the configuration file for an existing platform can be used as a base for the new configuration.

The structures/fields that most often need to be updated are:

- Memory table (`LINKCFG_memTable_00`)

- GPPINTID and DSPINTID fields within the IPS objects (LINKCFG_ipsTable_00)

Other objects and fields may also undergo changes based on the platform to be supported.

6.1.4.2 Static build configuration

Location:

- /$(DSPLINK)/config/bin

A PERL script is used as the tool for static build configuration of DSPLink. The static configuration tool file needs to be updated to generate the required environment variables for the new platform. The static configuration for an existing platform can be used as reference to make the required changes in the static configuration tool.

6.1.4.3 Make System

Makefile, COMPONENT, SOURCES and DIRS files may need to be added/ modified as per the changes made in the source base. For an overview and description of these files, refer to the section 7 on Build system in this document.

DSP-side:

Location:

- /$(DSPLINK)/make/DspBios/<DSP family name>

For this port, the DSP family for the new platform may be the same (e.g. C64xx). In this case, no changes are required in the DSP-side make system. If a new DSP family is to be supported, a new directory with required files needs to be added for it.

Correspondingly, a new distribution file may not be needed if the DSP family is the same. However, a new distribution file needs to be added to support a new device.

GPP-side:

Location:

- /dsplink/make/$(GPPOS)/$(PLATFORM)
- /dsplink/make/$(GPPOS)/<distribution file>.mk

New files need to be added for compilation and linking for the new device. The files for an existing device with the same Operating System can be used as a reference.

A new distribution file needs to be added for the platform. This distribution file contains all platform and OS specific paths to tools, compile/link options etc.

The section 7 gives a summary of relevant fields in the distribution file.

6.1.4.4 Setting up the communication between GPP and DSP

Since the physical link is the same for this porting scenario, no changes are required for this.

6.1.5 Checklist of activities involved

6.1.5.1 Updating sources

1. Update ARCH implementation on GPP-side

2. Port the hardware abstraction layer. OR use the Chip Support Library (if available).
3. Make the required changes as mentioned in the above sections on GPP and DSP-side of DSPLink.

6.1.5.2 Updating configuration and build system

1. Add the platform specific configuration files
2. Update the MAKE system as described in the earlier sections

6.1.5.3 Preliminary Testing

The sample applications provided with the DSP/BIOS™ LINK release can be used for initial validation of the port to the different platform. Each sample application demonstrates a different protocol. However, all sample applications demonstrate boot-loading the DSP, which should be the first functionality to be validated.

6.1.5.4 System Testing

Once all the sample applications are working as expected, the DSP/BIOS™ LINK test suite product can be adapted for the new platform. Executing the test-suite and resolving any issues found shall increase the confidence in the integrity and stability of the port.

6.2 Porting to a different physical link for existing platform(s)

6.2.1 Example

- Existing platform support:
    - Davinci DM6446 platform (ARM + C64x+) over shared memory
    - Red Hat Linux PC + DM6437 over PCI
- New platform support:
    - ARM of the Davinci DM6446 platform connected to DM6437 using VLYNQ.

In this port, the GPP and DSP are supported in different configurations. In addition, the GPP operating system also is the same. But the physical link used for communication between the two devices is different.

This type of port may be a medium to high complexity port, depending on the features supported by the new physical link.

6.2.2 Overview

The existing ports are zero-copy based protocols. If any physical link does not support the zero-copy protocol, logical layers within *DSPLINK* would also require changes.

When porting DSP/BIOS™ LINK to a different hardware platform for this scenario, consider:

1. Booting the DSP
2. Hardware abstraction layer for communicating using the new physical link
3. Changes in logical layers (if any)

6.2.3 Changes to source code

6.2.3.1 GPP-side:

ARCH component

Location:

- `/$(DSPLINK)/gpp/src/arch/<Device name>`

Example:

- `/$(DSPLINK)/gpp/src/arch/DM6446GEM`

The ARCH component contains all device-specific implementation.

<Device>.c

Location:

- `/$(DSPLINK)/gpp/src/arch/<Device name>.c`

Example:

- `/$(DSPLINK)/gpp/src/arch/DM6446GEM/dm6446gem.c`

This file provides the implementation of the DSP interface. It gives the top-level device-specific interface to the rest of DSPLink. This component is responsible for

configuring the hardware to make DSP reachable from GPP. It provides services to start, stop, idle and interrupt the DSP. It also manages the DSP resources to provide services to read from and write into DSPs memory space.

An existing implementation can be used as reference.

The DSP implementation usually shall not require changes. However, significant differences in DSP boot procedure or read/write into DSP memory space using the new hardware abstraction layer may result in changes to support multiple types of physical links.

Device-specific MAP file

Location:

- `/$(DSPLINK)/gpp/src/arch/<Device name>_map.c`

Example:

- `/$(DSPLINK)/gpp/src/arch/DM6446GEM/dm6446gem_map.c`

This file provides the mappings of function table interfaces/function pointers and other such information to the modules as provided in the DSPLink dynamic configuration (`$(DSPLINK)/config/all/CFG_<Device>_<Physical Link>.c`) provided in user-space.

An existing implementation can be used as reference.

HAL implementations

Location:

- `/$(DSPLINK)/gpp/src/arch/<Device name>_hal<type>.c`

Examples:

- `/$(DSPLINK)/gpp/src/arch/DM6446GEM/dm6446gem_hal.c`
- `/$(DSPLINK)/gpp/src/arch/DM6446GEM/dm6446gem_hal_boot.c`

This component provides functions to configure these hardware parameters:

- Clock settings and DDR/EMIF initialization (if required). This may not be required if the setting has already been done by a boot-loader.
- Interrupt services
- Power up and down the DSP
- DSP boot configuration and reset control
- MMU (in case the DSP has an MMU)
- Any other platform specific hardware configuration

Additional changes may be required based on the physical link to be supported.

6.2.3.2 DSP-side

On the DSP-side, the following changes are required to add support for the new platform:

Location:

- `/$(DSPLINK)/dsp/src/base/hal/DspBios/<Device name>`

Example:

- /$(DSPLINK)/dsp/src/base/hal/DspBios/DM6446GEM/hal_interrupt.c

On the DSP-side HAL module would require changes to use the interrupt services provided by the new physical link. However, cache module would remain the same.

TCI configuration file:

The TCI configuration file will usually not require changes since the platform is the same.

platform.h:

The platform.h file will usually not require changes since the platform is the same.

Changes in sample applications:

Sample applications will usually not require any changes.

6.2.4 Other Changes

6.2.4.1 Configuration

Location:

- /$(DSPLINK)/config/all

A new configuration file needs to be provided for the physical link to which DSPLink is being ported. The name of the configuration file depends on the device and physical link between the connected devices. For example, if the device is DM6446GEM and the physical link between the GPP (ARM) and the device is VLYNQ, the configuration file name would be CFG_DM6446GEM_VLYNQ.c.

The structures/fields that most often need to be updated are:

- Memory table (LINKCFG_memTable_00)
- GPPINTID and DSPINTID fields within the IPS objects (LINKCFG_ipsTable_00)

In addition, if a different System Organization type is supported by the physical link (e.g. discrete) or a different type of physical link is used (e.g. Processor-copy or DMA-copy), the LDRV implementation may need changes, and accordingly, configuration of POOL, DATA driver, MQT etc. may also require changes.

- LINKCFG_Pool
- LINKCFG_DataDrv
- LINKCFG_Mqt
- LINKCFG_RingIo
- LINKCFG_MpList
- LINKCFG_Mpcs

For example, ARGUMENT1 or ARGUMENT2 field in the configuration of these objects may be used to pass physical-link specific information to these modules.

Another example is that a different implementation of POOL can be used with a different name to differentiate it from the existing SMA POOL.

- If a new pool is added with a different function table, a different name can be used (e.g. VLYNQPOOL).
- A corresponding entry must be made in the configuration mapping file within the LDRV layer to provide the mapping information between the POOL name

and the function table. This file is: `/$(DSPLINK)/gpp/src/arch/<Device name>_map.c`

6.2.4.2 Static build configuration

Location:

- `/dsplink/config/bin`

A PERL script is used as the tool for static build configuration of *DSPLINK*. The static configuration tool files need to be updated to generate the required environment variables for the new platform. The static configuration for an existing platform can be used as reference to make the required changes in the static configuration tool.

6.2.4.3 Make System

`Makefile`, `COMPONENT`, `SOURCES` and `DIRS` files may need to be added/ modified as per the changes made in the source base. For an overview and description of these files, refer to the section 7 on Build system in this document.

DSP-side:

For this port, the DSP-side device for the new platform is the same (e.g. C64xx). Hence no changes are required in the DSP-side make system.

GPP-side:

For this port, the GPP side operating system as well as platform is already supported. Hence no changes are required in the GPP-side make files.

6.2.4.4 Setting up the communication between GPP and DSP

If a different System Organization type is supported by the physical link (e.g. discrete), or a different type of physical link is used (e.g. Processor-copy or DMA-copy), LDRV component on GPP-side, and corresponding modules on DSP-side need to be updated to add the relevant physical links.

GPP-side:

Location:

- `/$(DSPLINK)/gpp/src/ldrv/DRV`
- `/$(DSPLINK)/gpp/src/ldrv/DATA`
- `/$(DSPLINK)/gpp/src/ldrv/IPS`
- `/$(DSPLINK)/gpp/src/ldrv/MQT`
- `/$(DSPLINK)/gpp/src/ldrv/POOLS`
- `/$(DSPLINK)/gpp/src/ldrv/MPCS`
- `/$(DSPLINK)/gpp/src/ldrv/MPLIST`
- `/$(DSPLINK)/gpp/src/ldrv/RINGIO`
- `/$(DSPLINK)/gpp/src/ldrv/SMM`

Based on the design for the new physical link, new logical modules may require to be added to support the physical link features. For example, a different kind of shared pool implementation VLYNQPOOL may be added to support the new VLYNQ physical link.

DSP-side:

Location:

- `/$(DSPLINK)/dsp/src/base/drv`
- `/$(DSPLINK)/dsp/src/base/ips`
- `/$(DSPLINK)/dsp/src/data`
- `/$(DSPLINK)/dsp/src/msg`
- `/$(DSPLINK)/dsp/src/pools`
- `/$(DSPLINK)/dsp/src/mpcs`
- `/$(DSPLINK)/dsp/src/mplist`
- `/$(DSPLINK)/dsp/src/ringio`
- `/$(DSPLINK)/dsp/src/notify`

Based on the design for the new physical link, new logical modules may require to be added to support the physical link features. For example, a different kind of shared pool implementation VLYNQPOOL may be added to support the new VLYNQ physical link.

6.2.5 Checklist of activities involved

6.2.5.1 Updating sources

1. Update ARCH implementation on GPP-side as detailed above.
2. Port the hardware abstraction layer to the new physical link. OR use the Chip Support Library (if available).
3. Make the required changes as mentioned in the above sections on GPP and DSP-side of *DSPLINK*.

6.2.5.2 Updating configuration and build system

1. Add the platform specific configuration files.

6.2.5.3 Preliminary Testing

The sample applications provided with the DSP/BIOS™ LINK release can be used for initial validation of the port to the new physical link. Each sample application demonstrates a different protocol. However, all sample applications demonstrate boot-loading the DSP, which should be the first functionality to be validated.

6.2.5.4 System Testing

Once all the sample applications are working as expected, the DSP/BIOS™ LINK test suite product can be adapted for the new physical link. Executing the test-suite and resolving any issues found shall increase the confidence in the integrity and stability of the port.

6.3 Porting to a different physical link for a different platform

6.3.1 Example

- Existing platform support:
    - Red Hat Linux PC + DM6437 over PCI
- New platform support:
    - PowerPC connected to DM648 over a new physical link

The existing port to the DM6437 can be used as a reference. However, the GPP as well as DSP are different, and a different physical link is used for communication. This port may be a high complexity port.

6.3.2 Overview

For this port, a combination of porting activities mentioned for other platform ports needs to be implemented.

7 Build system

The make system used to build DSP/BIOS™ LINK is portable across multiple operating systems and platforms. When porting to a different OS, change to a different tool set may be required.

The User Guide describes the MAKE system in detail. This section gives an overview of common changes that will require to be made to the MAKE system. Specific changes for each type of port are given in the corresponding sections.

Some changes to the make system are required whether porting to a different operating system or to a different platform. This section gives an over

7.1.1 OS distribution file

DSPLINK supports build for the Linux and Windows development hosts. The different platforms may support different variants or versions of the target operating system. In addition, the tool-chain used for building the sources may differ based on the selected platform.

To support this, the make system provides a separate distribution file for the Operating System distribution being used. Typically, there is one distribution file for each platform-OS combination.

The distribution file for a specific platform-OS combination can be found within the *DSPLINK* installation at:

```
$(DSPLINK)/make/<$(GPPOS) | $(DSPOS)>
```

For example, the distribution file for the Davinci platform for Linux is:

```
$(DSPLINK)/make/Linux/davinci_mvlpro5.0.mk
```

The distribution file for the DSP-side for the Davinci platform for Linux is:

```
$(DSPLINK)/make/DspBios/c64xxp_5.xx_linux.mk
```

The configuration values within this distribution file can be modified to customize the make system for the user build environment. Some of the common values that may need to be modified are:

For GPP-side distribution file:

`BASE_BUILDOS`	Base directory for the GPP operating system
`BASE_TOOLCHAIN`	Base directory for the tool-chain
`STD_CC_FLAGS/ OS-specific flags`	Standard build flags for the compiler
`STD_AR_FLAGS/ OS-specific flags`	Standard build flags for the archiver
`STD_LD_FLAGS/ OS-specific flags`	Standard build flags for the linker

For DSP-side distribution file:

`BASE_INSTALL`	Base directory for the installed tools and operating system.

BASE_SABIOS	Base directory for the DSP operating system
XDCTOOLS_DIR	Base directory for the XDC tools installation
BASE_CGTOOLS	Base directory for the code generation tool chain
BASE_CSL	Base directory for the Chip Support Library (if required for the platform)
STD_CC_FLAGS	Standard build flags for the compiler
STD_AR_FLAGS	Standard build flags for the archiver
STD_LD_FLAGS	Standard build flags for the linker

In addition, there may be some configuration values within the distribution file that are specific to the selected platform-OS combination such as:

COMPILER	Name of the compiler
ARCHIVER	Name of the archiver
LINKER	Name of the linker

Other defines may be present/added as required for the specific Operating System.

7.1.2 System Tools configuration file

During the build process, different tools are used from the development environment to accomplish simple tasks e.g. move/ copy/ delete files, echo commands, etc.

For each target operating system, based on whether the build environment is Linux or Windows-based, the *DSPLINK* make system configures the system tools and system calls through a specific file `systools.mk` present in:

```
$(DSPLINK)/make/<$(GPPOS) | $(DSPOS)>
```

This file may also need to be customized for the user build environment.

Typically, the following configuration values may need to be modified:

BASE_PERL	Base directory for the PERL installation

7.1.3 Makefile, DIRS, COMPONENT and SOURCES files

To build a component successfully the developer needs to be aware of the following four files:

- `Makefile`
- `DIRS`
- `COMPONENT`
- `SOURCES`

7.1.3.1 Makefile

Each component requires a make file. This `Makefile` is standard for all the modules. User is not required to change this file. A warning to this effect is shown in these files.

A `Makefile` needs to be added at the root location of any new directory to be included in the build.

7.1.3.2 DIRS

This file provides a list of sub-directories that make up the component in a build configuration.

This file must be present at the base of any directory having sub-directories that need to be included in the build. It lists the sub-directories to be included in the build.

If any new directories are added to a base directory, the `DIRS` file in the base directory must be updated to add the new directory name.

If a directory does not already have a `DIRS` file and new sub-directories have been added in it, a new `DIRS` file must be added to ensure that the new sub-directories are included in the build.

Additional details of the `DIRS` file structure are given in the User Guide document.

7.1.3.3 COMPONENT

There is one `COMPONENT` file for every component in the OS specific folder for the component. This file affects the compilation and linking of the component by specifying OS specific attributes during the build process.

If any new header files are to be added, they need to be mentioned in the `COMPONENT` file to ensure that they are exported for the build process.

Additional details of the `COMPONENT` file structure are given in the User Guide document.

7.1.3.4 SOURCES

This file provides a list of files that make up the component in a build configuration. If any source files in a directory need to be included in the build, they need to be mentioned in the `SOURCES` file in that directory.

If a `SOURCES` file is not present in a newly added directory in which source files are present, a new `SOURCES` file with the list of files to be compiled needs to be added.

Additional details of the `SOURCES` file structure are given in the User Guide document.

8 Testing the port

8.1 Quick testing

The sample applications can be used as a unit test for validating the DSP boot-loading and different data transfer mechanisms. Unlike the test suite, these applications do not require users to understand the complete framework.

8.2 Full validation

Once the data transfers are stable, it is time to use the test suite for full system validation.

After completion of the porting activity, use the DSP/BIOS™ LINK test bench product (separately available) to validate the port.

9 Packaging

The ported version of DSP/BIOS™ LINK must be shipped in a similar manner as the original product. Also, the source code must comply with the coding standards used for the original product.