Texas Instruments

PORTING GUIDE

DSP/BIOS™ LINK

Version 1.10

LNK 017 PRT

MAR 03, 2004

This page has been intentionally left blank

IMPORTANT NOTICE

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

This page has been intentionally left blank.

TABLE OF CONTENTS

TABLE OF FIGURES

A. INTRODUCTION

1 Purpose

This document describes the changes that a user needs to make in the LINK sources for porting LINK to another OS or H/W 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 Abstraction Layer
HAL	Hardware Abstraction Layer

3 References

1.	DSP/BIOS™ LINK User Guide Version 1.10, dated MAR 03, 2004
2.	DSP/BIOS™ LINK OS Adaptation Layer for Linux Version 1.10, dated MAR 03, 2004
3.	DSP/BIOS™ LINK Link Driver Version 1.10, dated MAR 03, 2004
4.	DSP/BIOS™ LINK Shared Memory IOM Driver for OMAP5910 Version 1.10, dated MAR 03, 2004

4 Overview

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

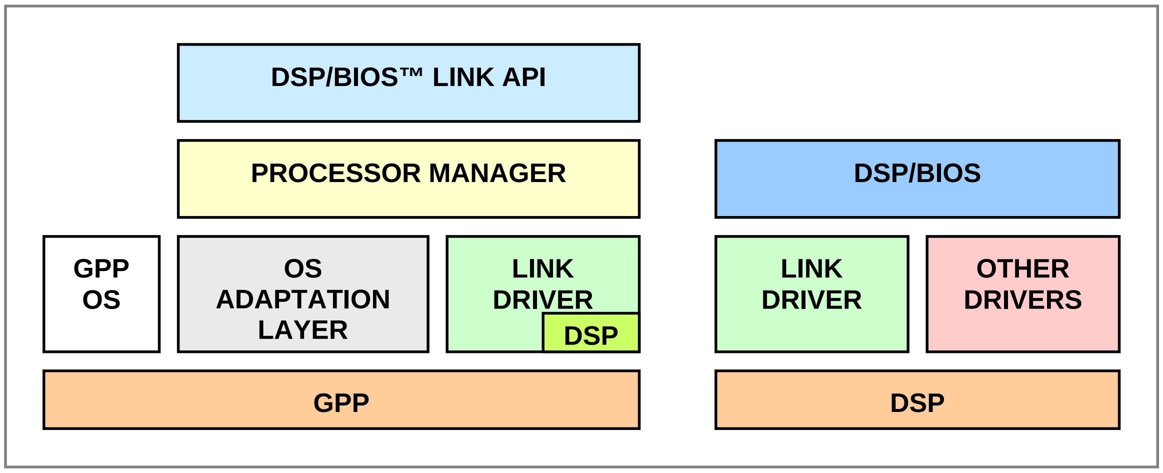

Figure 1. DSP/BIOS™ LINK Architecture

4.1 On the 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 DSP/BIOS™ LINK portable across different operating systems.

- 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 protocol across the GPP-DSP boundary.

  - DSP

    This subcomponent provides a generic interface to abstract 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.

    A hardware abstraction layer provides low-level functions to operate on the physical hardware. This layer brings more portability into implementation of DSP subcomponent.

- 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. 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.

4.2 On the DSP side

Here, the LINK DRIVER is one of the drivers in DSP/BIOS™. This driver specializes in communicating with the GPP over the physical link.

There is no specific DSP/BIOS™ LINK API on the DSP. The communication (data/ message transfer) is done using the DSP/BIOS™ modules- SIO/ GIO/MSGQ.

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 between GPP & DSP

§ Any combination of above

Currently DSP/BIOS™LINK is available on following platforms:

§ Innovator™ Development Kit (based on OMAP5910) running Montavista Linux® Professional Edition

§ DM310 EVM with DM642 connected through HPI.

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 OS

Figure 1 illustrates that the higher layers of DSP/BIOS™ LINK uses 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.

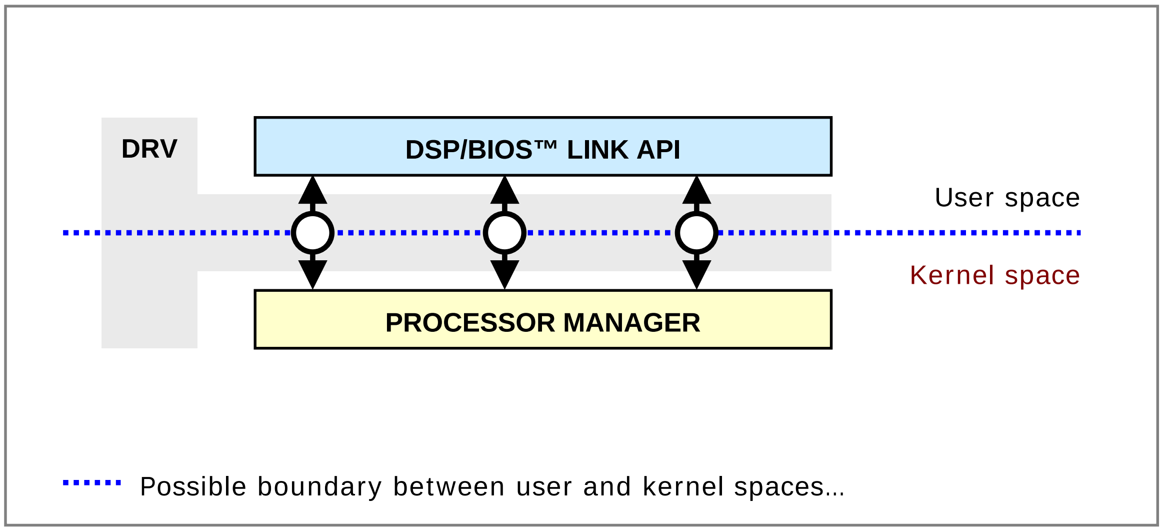

Figure 2. Interactions of DRV

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.

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.

5.1.6 PRCS

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

5.1.7 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.

5.1.8 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

5.2 Make 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. A brief summary of common tasks while porting to a different operating system is here.

5.2.1 Changing the Compiler

The variable `COMPILER` (defined in the file `compile.mk`) represents the fully qualified path to the compiler used.

To change the compiler, simply change the value of this variable to the new compiler.

If the new compiler uses different switch settings, than the previous one, you may be required to update the following variables based on the switches supported by the new compiler:

`STD_INC_PATH`

`STD_CC_FLAGS`

`STD_CC_DEFNS`

`COMPILER_DEB`

`COMPILER_REL`

If specific compiler flags were added in any of the COMPONENT file(s) then following variables should also be updated accordingly.

`USR_CC_FLAGS`

5.2.2 Changing the Archiver

The variable `ARCHIVER` (defined in the file `link.mk`) represents the fully qualified path to the linker used.

To change the archiver, simply change the value of this variable to the new archiver.

If the new archiver uses different switch settings, than the previous one, you may be required to update the following variables based on the switches supported by the new archiver:

`STD_AR_FLAGS`

`ARCHIVE_DEB`

`ARCHIVE_REL`

5.2.3 Changing the Linker

The variable `LINKER` (defined in the file `link.mk`) represents the fully qualified path to the linker used.

To change the linker, simply change the value of this variable to the new linker.

If the new linker uses different switch settings, than the previous one, you may be required to update the following variable(s) based on the switches supported by the new linker:

`STD_LD_FLAGS`

You may also be required to update the commands using the variable `LINKER` for the targets - `$(target_deb)` and `$(target_rel)`.

If specific linker flags were added in any of the COMPONENT file(s) then following variables should also be updated accordingly.

`USR_LD_FLAGS`

5.2.4 Other changes

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.

These tools are defined in the file `systools.mk`. Based on the changes to the development environment, these variables may need to be updated.

Base path for the header files and target specific libraries is defined in the file `osdefs.mk`.

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 can easily 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.

5.3.2 Updating build system

1. Update the MAKE system with rules specific for new OS and corresponding tool chain.
2. Create the file - COMPONENT – for each component including. the test suite.

5.3.3 Preliminary Testing

OSAL test-suite will help validating the functionality.

- This test suite has not been integrated into the remaining test suite so far. On Linux, the test suite executes in the user space whereas the OSAL operates in the kernel space.
- A common mechanism to execute OSAL test suite that can be used on the operating systems with/ without user-kernel boundary restrictions will be implemented in the future releases.

6 Porting to a different Hardware Platform

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

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

The hardware abstraction layer provides a set of functions that configure 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.

The DSP 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.

6.1 DSP

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.

6.2 HAL

This component provides functions to configure these hardware parameters:

- Clock settings
- Interrupt services
- MMU

Make the appropriate changes based on the target platform.

6.3 Other Changes

6.3.1 Make System

Make changes to the make system to specify platform specific options for compilers and linkers. Refer to section 5.2 for a summary of changes to the MAKE system.

6.3.2 Setting up the communication between GPP and DSP

The communication between the GPP and DSP depends upon the hardware. It involves:

§ Implementing the low-level control functions. This is usually done on the GPP side when the DSP component has being ported. However, the similar functionality needs to be implemented on the DSP side as well[1].

§ Defining a protocol that utilizes these functions to transfer data. The protocol must provide safeguards against the read-write hazards when both the GPP and DSP try to access the common control information.

[1] It is recommended that the link driver on DSP side be based on the IOM driver model.

In the current implementations, two different communication mechanisms are used:

§ Shared Memory (in OMAP)

§ HPI (On DM310 + DM642)

These implementations can be taken as a staring point for a port to another platforms. There is a possibility that protocol may easily be re-used.

6.4 Checklist of activities involved

6.4.1 Updating sources

1. Create an appropriate directory for new platform.
2. Port the hardware abstraction layer. OR use the Chip Support Library (if available).
3. Port the DSP component. The interface for the DSP component is already defined as a structure of function pointers. These functions need to be implemented during this activity.

6.4.2 Updating build system

1. Update the MAKE system with rules specific for new OS and corresponding tool chain.

6.4.3 Preliminary Testing

The DSP test-suite will help validating the functionality.

O This test suite has not been integrated into the remaining test suite so far. This limitation will be removed in the future releases.

7 Porting to a different Physical Link

Hardware platforms may provide different kinds of connectivity between GPP and DSP. In such cases, modify the DSP and hardware abstraction[2] components of DSP/BIOS™ LINK to factor in the different characteristics of the physical link.

7.1 DSP

The DSP subcomponent is closely tied to the configuration of the DSP vis-à-vis the GPP. The DSP may either be:

§ Directly accessible from the GPP or accessible through intermediate hardware

§ DSP boots independently or GPP is required to boot it.

These characteristics impact the design of DSP subcomponent. The basic services required from the DSP subcomponent are defined in the Link Driver design document.

7.2 Hardware Abstraction Layer

This layer must factor in the different characteristics of the physical link to achieve its tasks. The characteristics include:

§ Mechanism to read/write data from/to DSP memory space.

§ Method to interrupt the DSP.

§ Protocol to be implemented

§ Control functions for the device(s) affecting the operation of the link

7.3 Checklist of activities involved

7.3.1 Updating sources

1. Define the logical protocol to transfer data across the physical link.
2. On the GPP side, make necessary changes to LDRV_IO, DSP subcomponents.
3. On the DSP side, implement the link driver to use the new protocol[3].
4. Create the textual configuration file.

7.3.2 Updating build system

1. No change is normally required to the build system.

7.3.3 Preliminary Testing

The DSP test-suite will help validating the functionality.

O This test suite has not been integrated into the remaining test suite so far. This limitation will be removed in the future releases.

[2] If the CSL is available for the platform, it is advisable to use the same.
[3] It is recommended that the link driver on DSP side be based on the IOM driver model.

8 Testing the port

8.1 Quick testing

The LOOP sample application can be used as a unit test for validating the data transfer. Unlike the test suite, this application does 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 test suite provided with DSP/BIOS™ LINK 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.