DSP/BIOS™ LINK

MULTI-DSP DESIGN

LNK 182 DES

Version 1.20

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

TABLE OF FIGURES

1 Introduction

1.1 Purpose and Scope

This document defines the multi-DSP design of the DSP/BIOS™ LINK.

The architecture is intended to be independent of operating system on the GPP side. It, however, assumes DSP/BIOS™ to be running on the DSP.

DSP/BIOS LINK provides communication and control infrastructure between GPP and DSP and is aimed at traditional embedded applications. Many applications require a specific framework for communication and control between GPP and DSP. Therefore, the document also extends the architecture beyond DSP/BIOS LINK and discusses the possibility of building a reference framework (e.g. DSP/BIOS™ Bridge) over LINK. This is discussed in detail on section 6.4.

The document does not discuss the packaging and installation.

The development teams for DSP/BIOS LINK are the intended audience of this document.

1.2 Terms and Abbreviations

GPP	General Purpose Processor
DSP	Digital Signal processor
OS	Operating System
LINK	A generic term used for DSP/BIOS LINK. It appears in *italics* in all usages.

1.3 References

1.	LNK 137 DES	DYNAMIC CONFIGURATION

2 Overview

DSP/BIOS™ Link is runtime software, analysis tools, and an associated porting kit that simplifies the development of embedded applications in which a general-purpose microprocessor controls and communicates with a TI DSP. DSP/BIOS™ Link provides control and communication paths between GPP OS threads and DSP/BIOS™ tasks, along with analysis instrumentation and tools.

Previous releases of DSP/BIOS™ Link are targeted towards a GPP and a DSP type of platforms only. As the new products are becoming more DSP hungry, so solutions with multiple DSP (each one dedicated for a specific job) are used.

To cater these types of solution, DSP/BIOS™ Link must be upgraded to handle multiple-DSP with a GPP. This document outlines the changes and upgrades required.

3 Design

Below diagram shows the block level architecture of DSPLink.

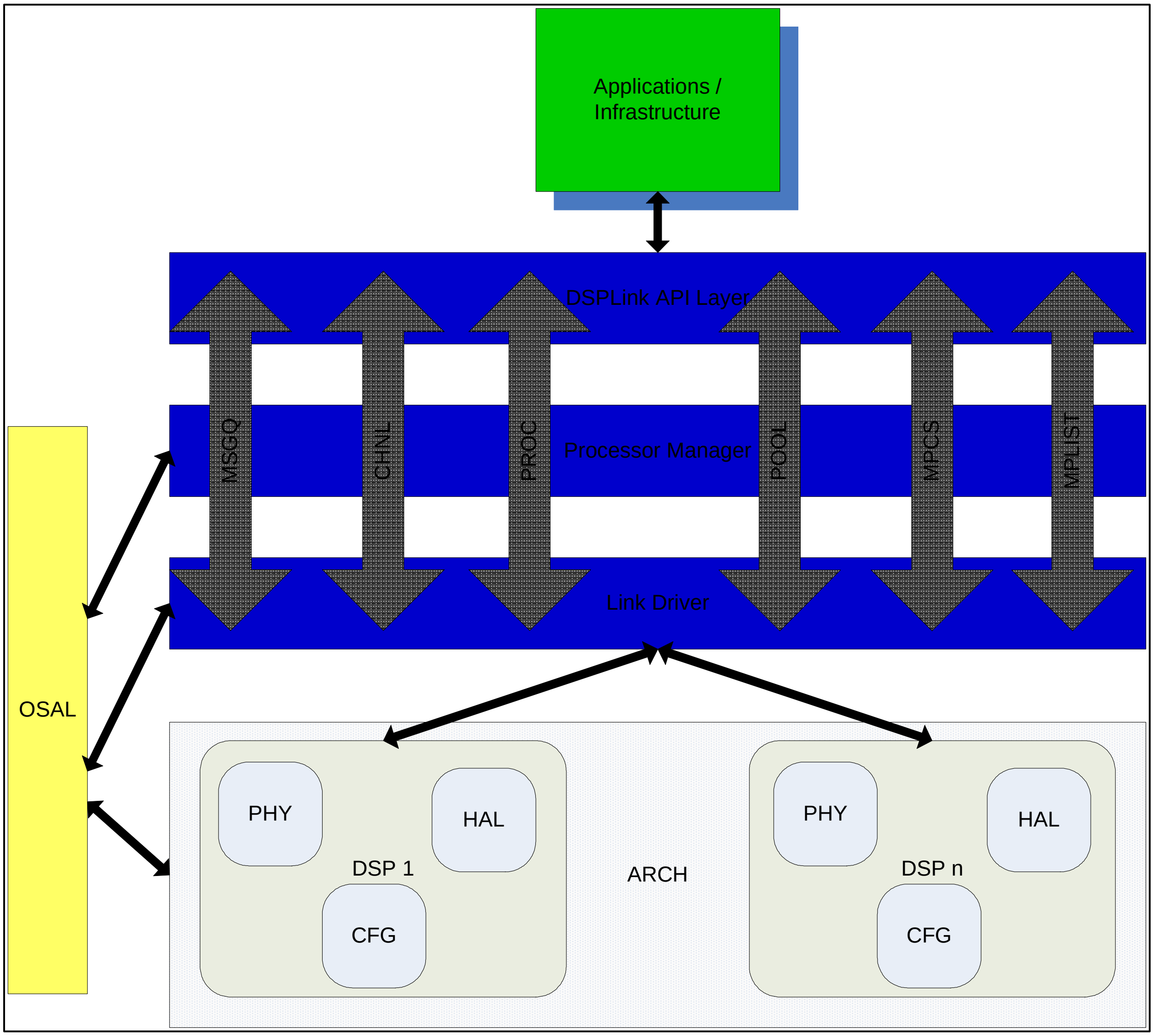

Figure 1. Block Level Architecture of DSPLink

The above diagram shows the block diagram of DSP/BIOS™ Link. Here except RINGIO component all other components fall into the LINK DRIVER. RINGIO component is a logical protocol which uses POOL, MPCS and NOTIFY APIs only, thus it does not uses any features from the LINK Driver. OSAL component abstract the OS on GPP, this makes DSP/BIOS™ Link a cross platform product.

In case of OSes like Linux, where there is a user and kernel level separation, processor manager and link driver layer remains in kernel (as kernel module) and API layer provides the features exposed by the kernel module. OSAL and HAL are also part of kernel module. So RINGIO is a pure user-land protocol.

All these components would provide communication between GPP and a DSP connected via ARCH module.

OSAL component provides OS provided features to all modules (except API layer) in an abstracted form.

3.1 ARCH

3.1.1 Design

This component would abstract the physical connection between GPP and DSPs through a set of hardware features. This hardware features are such as interrupts, DMA, shared memory, PCI/VLYNQ interface.

This component would represent for all the configured DSPs in the DSPLink system. Each DSP would be represented by a DSP object. These objects would be used for serving the request (coming for Link Driver module) and managing/maintaining the DSPs. These DSP objects would be associated to a DSP using a DSP identifier. So there would an array of DSP objects (length equals number of DSPs configured).

For better manageability and portability of ARCH component, this component is divided into four internal parts, their connection is show below:

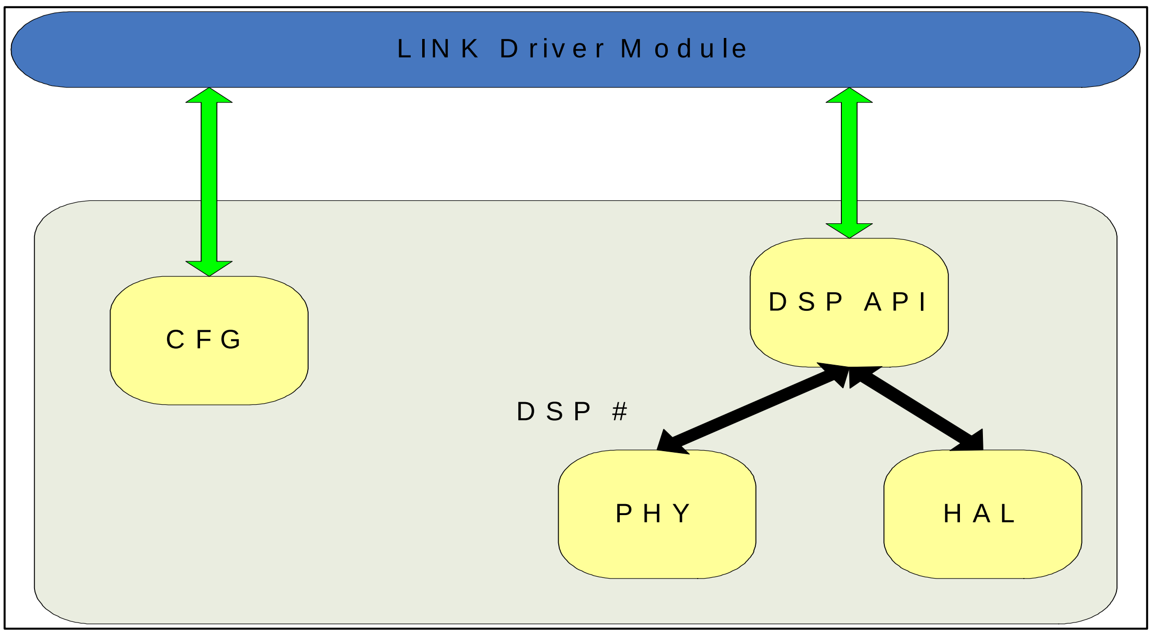

Figure 2. Connectivity Diagram of ARCH component

CFG module represents configuration mapping information, which are supported by a DSP. This information is used at runtime to cross check the user provided configurations, so that DSPLink is configured correctly according to the supported configurations.

DSP API part would connect the ARCH component to the Link Driver module through a set of APIs. This layer would use the PHY and HAL layer to make required hardware operations.

PHY stands for physical connection/physical interface, for e.g. PCI is physical interface. This part would initialize the physical interface so that DSP connected through it is usable by DSPLink. For e.g. for PCI DSP cards, PHY part would initialize all PCI and DSPLink required hardware features (such as mapping of memories). Most often all hardware features required from a DSP are accessible from GPP as a set of registers, which would be mapped to GPP address space by the PHY part.

HAL abstract the required hardware features of a DSP. These hardware features are such as interrupts, DMA, power off/on, mapping of a DSP memory into GPP address space.

HAL and **PHY** are plugged into the DSP API layer as interface so that if a DSP supports multiple of these layers then it easy to chose the required ones.

As above mentioned most often hardware features are just a set of registers, so if a particular DSP can be connected to a GPP in different ways, then developers would create different PHY layer according to physical interface and keep the HAL layer intact as they would take a base address of the register area and add offsets accordingly. The below diagram shows the concept:

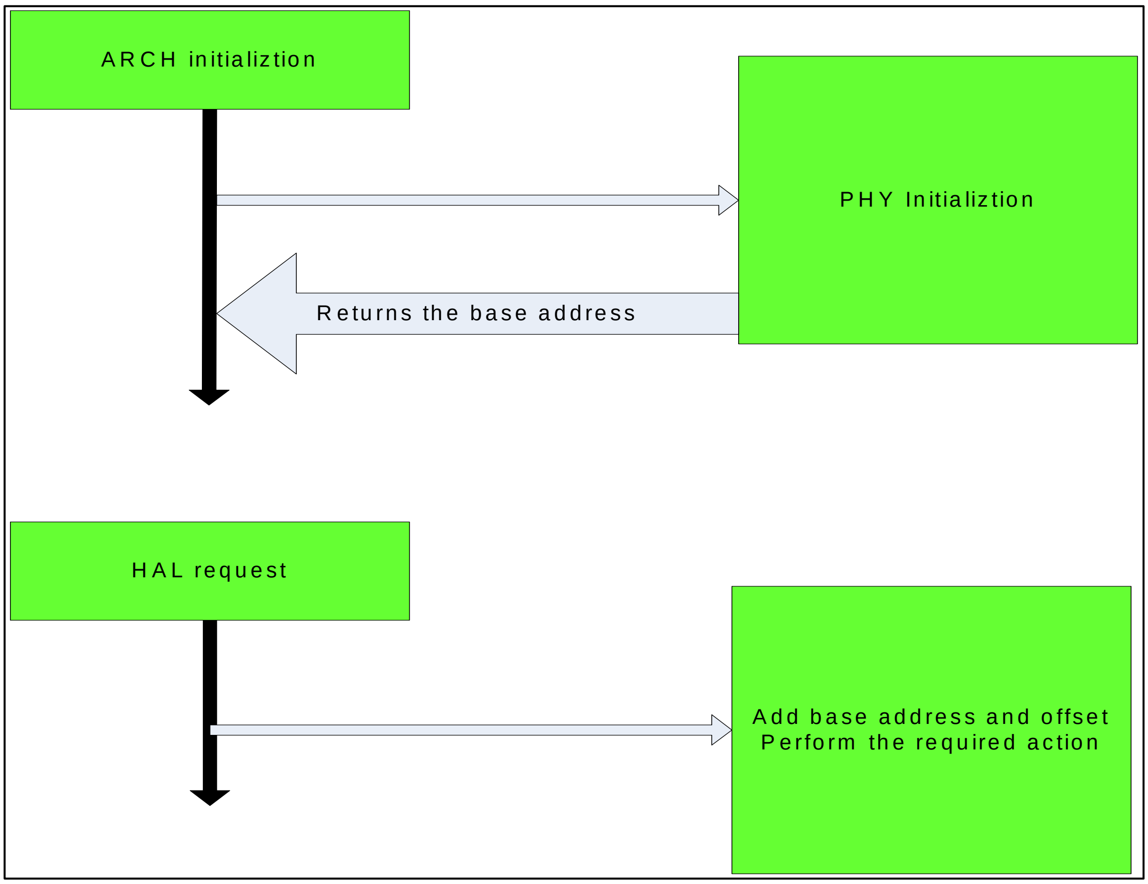

Figure 3. Concept demonstration

Here PHY returns the base address of the exposed window into the DSP address space by the physical interface, this base address is mapped in GPP address space so that GPP can access it. Now, whenever a HAL request comes, HAL logic first maps the required register area to the exposed window and then simply adds the base address to registers offset and performs the required action.

APIs exposed by HAL layer are as follows:

FnBootCtrl	This API would provide the boot loading functionalities.
FnIntCtrl	This API would provide the interrupt management related functionalities.
FnMapCtrl	This API would provide mapping/unmapping of a DSP address into GPP address space. This API would exist only on platforms where

	DSP address space is not directly visible to GPP. For e.g. PCI DSP. VLYNQ DSP.
FnPwrCtrl	This API would provide the power management functionalities such as power off/on, reset/release DSP.
FnRead	This API would read a buffer from DSP memory.
FnWrite	This API would write a buffer to DSP memory.
FnReadDMA	This API would DMA contents from a buffer in DSP address space to a buffer in GPP address space. This API would exist only on platforms where no shared memory is present between GPP and a DSP.
FnWriteDMA	This API would DMA contents from a buffer in GPP address space to a buffer in DSP address space. This API would exist only on platforms where no shared memory is present between GPP and a DSP.

APIs exposed by PHY layer are as follows:

phyInit	This API would initialize the physical interface. On platforms where no shared memory is present between GPP and a DSP, it would return the base address of the exposed window and map the exposed window into GPP address space. On platform like PCI DSP, this is equivalent to PCI driver initialization.
phyExit	This API would finalize the physical interface and relinquish all hardware features that were in used by GPP.

APIs exposed by DSP API layer are as follows:

DSP_init	This API would initialize the DSP identified by the DSP identifier. By using FnPwrCtrl API from HAL layer & phyInit from PHY layer, it would power on all required hardware modules and DSP. Then it would put the DSP in reset mode.
DSP_exit	By using FnPwrCtrl API from HAL layer & phyExit from PHY layer, it would put the DSP in reset mode and power off all hardware modules and the DSP.
DSP_start	By using FnPwrCtrl API from HAL layer, it would program the DSP start address and release the DSP from reset mode.
DSP_stop	By using FnPwrCtrl API from HAL layer, it would put the DSP in reset mode.
DSP_idle	This API would idle the DSP, available on platform where hardware supports idle mode.
DSP_intCtrl	By using FnIntCtrl API from HAL layer, it would perform the specified DSP interrupt control activity for e.g. interrupt generation to DSP, acknowledge, clear interrupt etc.
DSP_read	By using FnRead API from HAL layer, it would read a buffer from DSP memory.
DSP_write	By using FnRead API from HAL layer, it would write a buffer to DSP memory.

DSP_addrConvert	This API would convert addresses between GPP and DSP address space.
DSP_control	This API would provide a hook for performing device dependent control operation. For e.g. it provides DMA functionality using FnReadDMA/FnWriteDMA APIs from HAL layer.

3.2 Configuration

Configuration mapping information is now represented as an array, indexed using the processor identifier. Each element is a configuration mapping structure for a DSP. This array would be populated by CFGMAP_attachObject function called by LDRV_init, which in turn would be called at PROC_attach time.

Main purpose of this function is to tie a DSP to particular procId. For example, in multi-DSP system, DSP#0 can be programmed for procId 1, for a particular iteration and in second iteration it can be moved to procId 0. This creates the true dynamic behavior of DSPLink.

This function would take a DSP name and then it would search the dspName in the array containing name and object association (CFGMAP_objDB). User has to add name and object association into this array (CFGMAP_objDB) for new a platform. For e.g.

```
CONST CFGMAP_ObjDB CFGMAP_objDB [] = {
#if defined (DM6446GEM)
                                    {
                                     "DM6446GEM",
                                     &DM6446GEMMAP_Config,
                                     },
#endif          /* if defined (DM6446GEM) */
```

3.3 Dynamic Configuration

Config is pre-defined 'C' source file that contains with configuration values defined within a fixed structure format. This file is compiled with the DSPLink user library by default.

For multi-DSP DSPLink system, Config would be broken into more than one 'C' file. For a GPP and a DSP system, files would be CFG_gpp.c, CFG_dsp.c, and CFG_common.c. For multiple DSP in the system, there would multiple CFG_dsp#.c files. The idea behind this division is to configure DSPLink with all common and GPP specific configuration at the start of DSPLink and then configure each DSP as required by the application.

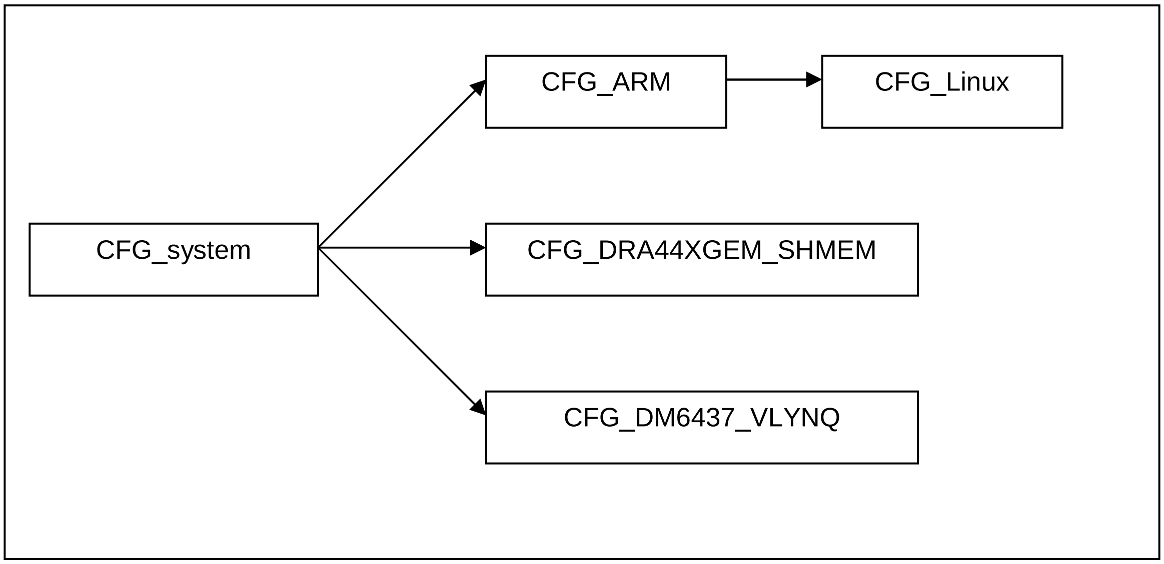

Figure 4. CFG capturing architecture of DSPLink

Configuration values from the files CFG_gpp.c and CFG_common.c are passed to DSPLink through PROC_setup API, whereas values from CFG_DSP#.c file are passed through PROC_attach API.

For backward compatibility, PROC_setup requires all config values (i.e. for DSP and GPP as well) to be passed to it. But values related to DSP are read in PROC_attach step only.

Previous version of DSPLink used to take all DSP related values at the time of PROC_setup time, which limits running a DSP with some modification in configuration values without calling PROC_destroy.

Passing the configuration values in PROC_attach solves the above problem, as new configuration values can be passed after calling PROC_detach and then calling PROC_attach with new values.

Now each CFG_dsp#.c file is logically divided into 2 sections, application specific configuration values and system integrator configuration values. All obvious and frequently changed configuration values by an application writer are exposed as macros (#define in C syntax) so that application writer does not has to go through the whole C file to find out which values he/she update according to his/her system. Similarly all values specific to system integrator are exposed as macros in the beginning of the file.

Currently number of a specific item is hard coded into some structures; we would be replacing them with sizeof operator syntax. This would solve the problem of incorrect configuration values.

For e.g. in LINKCFG_Dsp number of MEMENTRIES would be written as

```
sizeof (LINKCFG_memTable_00) / sizeof (LINKCFG_MemEntry) ;
```

3.4 Configure Script

Static configuration of DSPLink is done through a Perl script called configure.pl, which is invoked through dsplinkcfg script. This static configuration chooses the platform type (from available platforms), GPP OS, DSP type, etc. Also this script generates proper compiler/linker flags and macros to compile correct version of code file if multiple versions exists of the same.

For multi-DSP DSPLink system, this script would be enhanced so that it provides the following features:

- Command line configuration selection.
- Easy addition of new platform.

Command line would enable application writer to automate the configuration step (Previously it was a manual task). Writers can pass the configuration values directly while invoking the configure script. For e.g.:

```
$./configure.pl –-platform=Davinci --gppos=MVL5 --dspos=BIOS5X --
modules=lmrc
```

Following are the full list of option provided by this script:

--platform	Indicates the platform to be used. For e.g. --platform=DAVINCI, Chooses DAVINCI platform.
--nodsp	Indicates number of DSPs present in the system. For e.g. --nodsp=2, two DSPs are presented in the system.
--dsp_#	Indicates which DSP to be used as DSP#. For e.g. --dsp_1=DM6446GEM, Use Davinci Gem DSP.
--phy_#	Indicates which physical interface to be used for DSP#. For e.g. --phy_1=DM6446GEMSHARED, Davinci Gem is connected through shared memory interface.
--dspos_#	Indicates which DSP OS to be used for DSP#. For e.g. --dspos_1=DSPBIOS5XX, Davinci Gem uses DspBios 5.XX.
--gppos	Indicates which GPP OS to be used for chosen platform. For e.g. --gppos=MVL4G, use montavista pro 4.0 with glibc on GPP.
--comps	Indicates which component to be included while building DSPLink. For e.g. --comps=lmrc, chooses MPLIST, MSGQ, RINGIO and CHNL components.
--trace	Indicate whether trace has to be enabled or not. For e.g. --trace=1, trace is enabled.
--loader	Indicate which loader to be used for boot loading DSPs on the GPP. This is required when GPP/Platform supports multiple loaders type otherwise no need to provide it. For e.g. --

	loader=COFF_LOADER.
--fs	Indicates which filesystem to be used on GPP[OS]. This is required when GPP/Platform supports multiple filesystem type otherwise no need to provide it. For e.g. --loader=PRFILE_FS.

This script would display help messages if any required option is not provided. This message would give enough details to allow users to provide correct options.

As DSPLink is being ported to new and newer platforms, configure script must allow writers to add their own definitions inside the script in easier way. This is done in the following way:

1. Loader definitions:

Create entries like below for the new loader:

```
my %CFG_LOADER_YOUR =
(
    'NAME'   => 'YOUR LOADER', # name of the loader
    'ID'     => 'YOUR_LOADER', # Identifier
    'DESC'   => 'Your file format loader', # a small description
);
```

Example:

```
my %CFG_LOADER_COFF =
(
    'NAME'   => 'COFF LOADER',
    'ID'     => 'COFF_LOADER',
    'DESC'   => 'TI Coff file format loader',
);
```

Then add the created entry in the global array of loaders:

```
my %CFG_LOADERS =
(
    '0'  => \%CFG_LOADER_COFF,
    …
    …
    'n'  => \%CFG_LOADER_YOUR
) ;
```

2. Filesystem definitions:

Create entries like below for the new filesystem:

```
my %CFG_FS_YOUR =
(
    'NAME'   => 'Your Filesystem',
    'ID'     => 'YOUR_FS',
    'DESC'   => "Your filesystem",
);
```

Example:

```
my %CFG_FS_PRFILE =
(
    'NAME'   => 'PrFile Filesystem',
    'ID'     => 'PRFILE_FS',
    'DESC'   =>    "Read PrFile guide for further details",
);
```

Then add the created entry in the global array of filesystems:

```
my %CFG_FSS =
(
    '0'  => \%CFG_FS_PSUEDO,
    '1'  => \%CFG_FS_PRFILE,
    …
    …
    'n'  => \%CFG_FS_YOUR
) ;
```

3. GPP OS definitions:

Create entries like below for the new GPP OS:

```
my %CFG_GPPOS_YOUR =
(
    'NAME'   => 'MVL4U', #name of the GPP OS
    'PREFIX' => 'mvl4u', #prefix, used for generating file names
    'ID'     => 'MVL4U', #identifier
    'DESC'   => 'Montavista Pro 4.0 Linux + uCLibc Filesystem',
    'VER'    => '2.6.10', #Version (if any)
    'TYPE'   => 'Linux',  #Type of GPP OS (Linux, PrOs, WinCE)
    'LOADER' => \%CFG_LOADER_YOUR, # loader used in this OS.
);
```

Example:

```
my %CFG_GPPOS_MVL4U =
(
    'NAME'   => 'MVL4U',
    'PREFIX' => 'mvl4u',
    'ID'     => 'MVL4U',
    'DESC'   => 'Montavista Pro 4.0 Linux + uCLibc Filesystem',
    'VER'    => '2.6.10',
    'TYPE'   => 'Linux',
    'LOADER' => \%CFG_LOADER_COFF,
) ;
```

Then add the created entry in the global array of GPP OSes:

```
my %CFG_GPPOS =
(
    '0' => \%CFG_GPPOS_MVL4U,
    '1' => \%CFG_GPPOS_MVL4G,
    '2' => \%CFG_GPPOS_MVL5U,
    '3' => \%CFG_GPPOS_MVL5G,
    '4' => \%CFG_GPPOS_RHEL4,
    '5' => \%CFG_GPPOS_RHL9,
    '6' => \%CFG_GPPOS_PROS,
    'n' => \% CFG_GPPOS_YOUR
) ;
```

4. DSP OS definitions:

Create entries like below for the new DSP OS:

```
my %CFG_DSPOS_YOUR =
(
    'NAME'   => 'YOUR_DSP_OS',
    'PREFIX' => 'yourdspos',
```

```
    'ID'      => 'YOURDSPOS',
    'DESC'    => 'Your DSP OS',
    'VER'     => '5.XX',
    'TYPE'    => 'DspBios',

) ;
```

Example:

```
my %CFG_DSPOS_5XX =
(
    'NAME'    => 'DSPBIOS5XX',
    'PREFIX' => 'dspbios5xx',
    'ID'      => 'DSPBIOS5XX',
    'DESC'    => 'DSP/BIOS (TM) Version 5.XX',
    'VER'     => '5.XX',
    'TYPE'    => 'DspBios',
) ;
```

Then add the created entry in the global array of DSP OSes:

```
my %CFG_GPPOS =
(
    '0' => \%CFG_GPPOS_MVL4U,
    '1' => \%CFG_GPPOS_MVL4G,
    '2' => \%CFG_GPPOS_MVL5U,
    '3' => \%CFG_GPPOS_MVL5G,
    '4' => \%CFG_GPPOS_RHEL4,
    '5' => \%CFG_GPPOS_RHL9,
    '6' => \%CFG_GPPOS_PROS,
    'n' => \% CFG_GPPOS_YOUR
) ;
```

5. Physical Interface definitions:

Create entries like below for the new Physical interface:

```
my %CFG_PHY_YOUR =
(
    'ID'      => 'YOUR_PHY',
    'DESC'    => 'Your Physical Interface',
    'DEV'     => 'DAVINCIGEM', #Target Device connect by this interface
) ;
```

Example:

```
my %CFG_PHY_DAVINCISHARED =
(
    'ID'      => 'DAVINCI_SHAREDPHY',
    'DESC'    => 'Shared Physical Interface',
    'DEV'     => 'DAVINCIGEM',
);
```

Then add the created entry in the global array of DSP OSes:

```
my %CFG_GPPOS =
(
    '0' => \%CFG_GPPOS_MVL4U,
    '1' => \%CFG_GPPOS_MVL4G,
    '2' => \%CFG_GPPOS_MVL5U,
    '3' => \%CFG_GPPOS_MVL5G,
    '4' => \%CFG_GPPOS_RHEL4,
    '5' => \%CFG_GPPOS_RHL9,
```

```
    '6' => \%CFG_GPPOS_PROS,
    'n' => \% CFG_GPPOS_YOUR
) ;
```

6. DSP processor definitions:

Create entries like below for the new DSP processor:

```
my %CFG_DSP_YOUR =
(
    'NAME'   => 'YOURDSP',
    'PREFIX' => 'Yourdsp',
    'ID'     => 'YOURDSP',
    'DESC'   => 'Your DSP',
    'TYPE'   => 'C64XX',
) ;
```

Example:

```
my %CFG_DSP_DAVINCIGEM =
(
    'NAME'   => 'DAVINCIGEM',
    'PREFIX' => 'Davincigem',
    'ID'     => 'DAVINCIGEM',
    'DESC'   => 'On-Chip DSP of DaVinci SoC',
    'TYPE'   => 'C64XX',
);
```

Then add the created entry in the global array of DSP processors:

```
my %CFG_DSPS =
(
    '0'  => \%CFG_DSP_DAVINCIGEM,
    '1'  => \%CFG_DSP_DAVINCIHDGEM,
    '2'  => \%CFG_DSP_JACINTOGEM,
    '3'  => \%CFG_DSP_DM642,
    '4'  => \%CFG_DSP_DM64LC,
    '5'  => \%CFG_DSP_DAVINCIGEM1,
) ;
```

7. Base Platform definitions:

Create entries like below for the new platform:

```
my %CFG_PLATFORM_YOUR =
(
    'NAME'   => 'YOURPLATFORM',
    'ID'     => 'YOURPLATFORM',
    'PREFIX' => 'Yourplatform', # used for generating directories and
filenames, and also used for picking up correct files.
    'DESC'   => "YOUR PLATFORM description",
    'GPPOS'  => [
                  \%CFG_GPPOS_PROS,
                ], # Supported GPP OSes (multiple possible)
    'DSPS'   => [ #Supported DSP with combination of DSP, GPP OS, PHY,
DSP OS
                    [
                      \%CFG_DSP_YOUR, # DSP of the system
                      \%CFG_PHY_YOUR, # Phy type of DSP
                      \%CFG_GPPOS_YOUR, # GPP OS
                      \%CFG_DSPOS_YOUR # DSP OS
```

```
                    ],
                ],
) ;
```

Example:

```
my %CFG_PLATFORM_DAVINCI =
(
    'NAME'   => 'DAVINCI',
    'ID'     => 'DAVINCI',
    'PREFIX' => 'Davinci',
    'DESC'   => "DaVinci SoC - C64P DSP interfaced directly to ARM9",
    'GPPOS'  => [
                  \%CFG_GPPOS_MVL4U,
                  \%CFG_GPPOS_MVL4G,
                  \%CFG_GPPOS_MVL5U,
                  \%CFG_GPPOS_MVL5G,
                  \%CFG_GPPOS_PROS,
                ],
    'DSPS'   => [
                    [
                      \%CFG_DSP_DAVINCIGEM,
                      \%CFG_PHY_DAVINCISHARED,
                      \%CFG_GPPOS_MVL4U, # montavista pro 4.0
                      \%CFG_DSPOS_5XX
                    ],
                    [
                      \%CFG_DSP_DAVINCIGEM,
                      \%CFG_PHY_DAVINCISHARED,
                      \%CFG_GPPOS_MVL4G, # montavista pro 5.0
                      \%CFG_DSPOS_5XX
                    ],
                ],
);
```

Then add the created entry in the global array of DSP processors:

```
my %CFG_PLATFORMS =
(
    '0'   => \%CFG_PLATFORM_DAVINCI,
    '1'   => \%CFG_PLATFORM_DAVINCIHD,
    '2'   => \%CFG_PLATFORM_JACINTO,
    '3'   => \%CFG_PLATFORM_LINUXPC,
    'n'   => \%CFG_PLATFORM_YOUR
);
```

3.5 Modules Changes

As previously, DSPLink used to exist between a GPP and a DSP, so only one form of module implementation used to exist in the DSPLink. For e.g. if user chooses DaVinci platform then only zero copy implementation of MSGQ, CHNL is compiled. But now multiple DSP exists which can have different physical interface, so we would require different implementation of these modules to coexists inside the DSPLink. For example if we have a platform where a PCI DSP is attached to DaVinci board, then zero copy implementation would exist between GPP and DaVinci Gem and sync copy implementation would exist between GPP and PCI DSP.

So, to achieve the above said, all modules would be plugged into Link Driver layer as function interface. This plugging would be done at the runtime using the user provided configuration data.

Below diagram shows the concept:

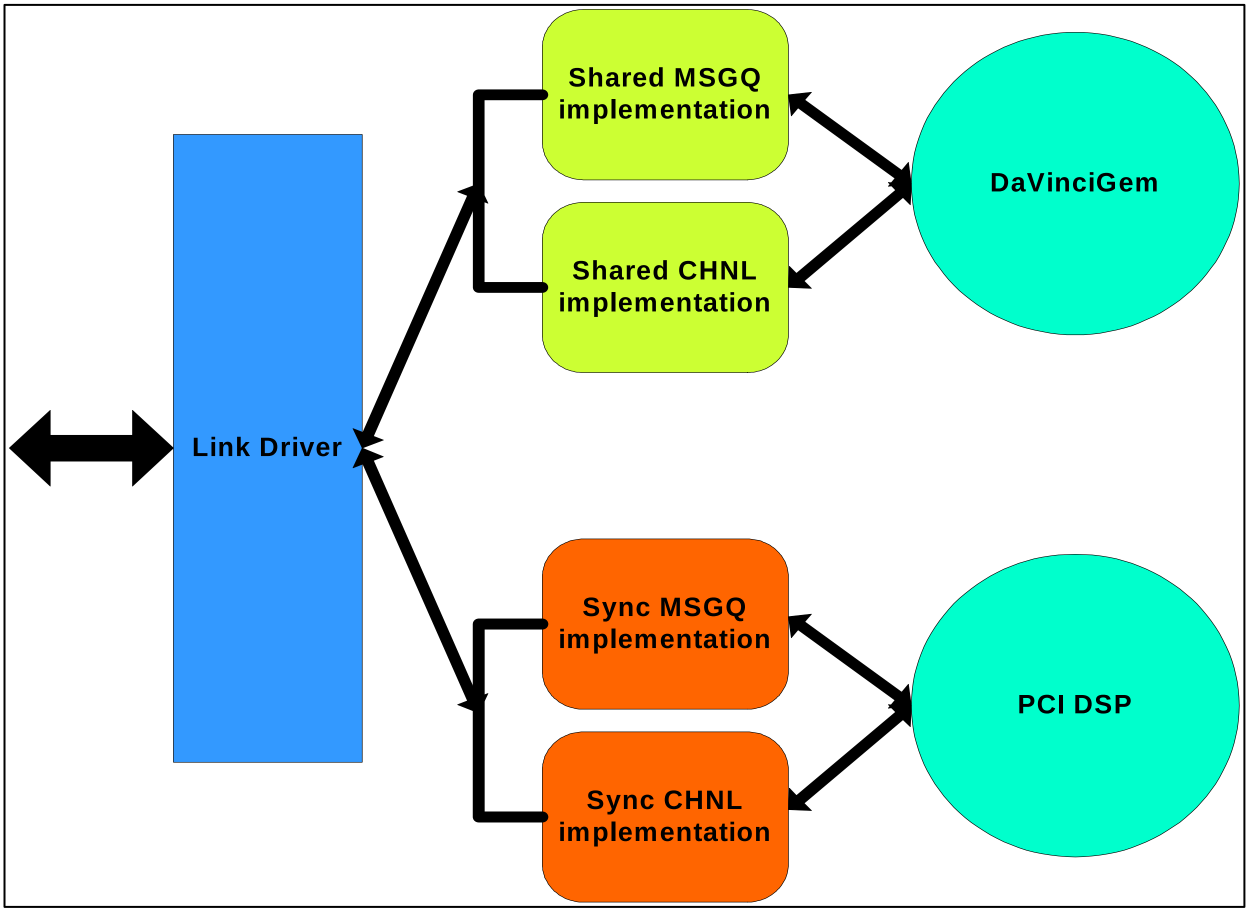

Figure 5. Module changes

4 Details

4.1 DSP Layer

DSP API part would connect the ARCH component to the Link Driver module through a set of APIs.

4.1.1 DSP_moduleInit

This function initializes the DSP module.

Syntax

```
NORMAL_API  Void DSP_moduleInit (Void) ;
```

Arguments

None.

Return Value

None.

Comments

None.

Constraints

None.

See Also

DSP_moduleExit

4.1.2 DSP_moduleExit

This function finalizes the DSP module.

Syntax

```
NORMAL_API  Void DSP_moduleExit (Void) ;
```

Arguments

None.

Return Value

None.

Comments

None.

Constraints

None.

See Also

DSP_moduleInit

4.1.3 DSP_init

This function initializes a DSP and plugs the DSP interface. Also calls the init function from the attached interface .

Syntax

```
NORMAL_API  DSP_STATUS DSP_init (IN ProcessorId     dspId,
                                 IN DSP_Interface * interface) ;
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	DSP_Interface *	interface
	Interface to DSP/DEVICE APIs	

Return Value

DSP_SOK	Operation successfully completed.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	DSP_setup function wasn't called before calling this function.
DSP_EVALUE	Invalid DSP MMU endianism configuration.

Comments

None.

Constraints

None.

See Also

DSP_exit

4.1.4 DSP_ exit

This function finalizes a DSP and also calls the exit function of attached DSP/Device interface.

Syntax

```
NORMAL_API  DSP_STATUS DSP_exit (IN ProcessorId     dspId) ;
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	DSP_setup function wasn't called before calling this function.

Comments

None.

Constraints

None.

See Also

DSP_init

4.1.5 DSP_ start

This function causes DSP to start execution from the given DSP address. DSP is put to STARTED state after successful completion. This is achieved by calling start function from the attached interface.

Syntax

```
NORMAL_API  DSP_STATUS DSP_start (IN ProcessorId dspId,
                                  IN Uint32      dspAddr) ;
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	Uint32	dspAddr
	Address to start execution from.	

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	DSP_setup function wasn't called before calling this function.

Comments

None.

Constraints

None.

See Also

DSP_stop

4.1.6 DSP_ stop

This function stops execution on DSP. DSP is put to STOPPED state after successful completion. This is achieved by calling stop function from the attached interface.

Syntax

```
NORMAL_API  DSP_STATUS DSP_stop (IN ProcessorId dspId) ;
```

Arguments

`IN`	`ProcessorId`	`dspId`
	Processor Identifier	

Return Value

`DSP_SOK`	Operation successfully completed.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EFAIL`	DSP_setup function wasn't called before calling this function.

Comments

None.

Constraints

None.

See Also

DSP_start

4.1.7 DSP_idle

This function idles the DSP. DSP is put to IDLE state after successful completion. This is achieved by calling idle function from the attached interface.

Syntax

```
NORMAL_API  DSP_STATUS DSP_idle (IN ProcessorId dspId) ;
```

Arguments

`IN`	`ProcessorId`	`dspId`
	Processor Identifier	

Return Value

`DSP_SOK`	Operation successfully completed.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EFAIL`	DSP_setup function wasn't called before calling this function.

Comments

None.

Constraints

None.

See Also

DSP_start/DSP_stop

4.1.8 DSP_ intCtrl

This function performs the specified DSP interrupt control activity. This is achieved by calling intCtrl function from the attached interface.

Syntax

```
NORMAL_API  DSP_STATUS DSP_intCtrl
                    (IN         ProcessorId       dspId,
                     IN         Uint32            intId,
                     IN         DSP_IntCtrlCmd    cmd,
                     IN OUT OPT Pvoid             arg);
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	Uint32	intId
	Interrupt ID	
IN	DSP_IntCtrlCmd	cmd
	Interrupt control command to be performed.	
IN/OUT OPT	Pvoid	arg
	Optional input/output argument specific to each control command.	

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	DSP_setup function wasn't called before calling this function.

Comments

None.

Constraints

None.

See Also

DSP_start/DSP_stop

4.1.9 DSP_read

This function reads data from DSP. This is achieved by calling read function from the attached interface.

Syntax

```
NORMAL_API  DSP_STATUS DSP_read
         (IN  ProcessorId  dspId,
          IN  Uint32       dspAddr,
          IN  Endianism    endianInfo,
          IN  Uint32       numBytes,
          OUT Uint8 *      buffer);
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	Uint32	dspAddr
	DSP address to read from.	
IN	Endianism	endianInfo
	endianness of data - indicates whether swap is required or not	
IN	Uint32	numBytes
	Number of bytes to read.	
OUT	Uint8 *	buffer
	Buffer to hold the read data	

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	DSP_setup function wasn't called before calling this function.

Comments

None.

Constraints

None.

See Also

DSP_write

4.1.10 DSP_write

This function writes the data to DSP. This is achieved by calling write function from the attached interface.

Syntax

```
NORMAL_API  DSP_STATUS DSP_write
         (IN  ProcessorId  dspId,
          IN  Uint32       dspAddr,
```

```
IN  Endianism    endianInfo,
IN  Uint32       numBytes,
OUT Uint8 *      buffer);
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	Uint32	dspAddr
	DSP address to write to.	
IN	Endianism	endianInfo
	endianness of data - indicates whether swap is required or not	
IN	Uint32	numBytes
	Number of bytes to write.	
OUT	Uint8 *	buffer
	Buffer to write	

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	DSP_setup function wasn't called before calling this function.

Comments

None.

Constraints

None.

See Also

DSP_read

4.1.11 DSP_ addrConvert

This function converts address between GPP and DSP address space. This is achieved by calling addrConvert function from the attached interface.

Syntax

```
NORMAL_API  DSP_STATUS DSP_addrConvert
                (IN  ProcessorId       dspId,
                 IN  Uint32            addr,
                 IN  DSP_AddrConvType  type);
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	Uint32	addr

Address to be converted. If DSP address, the addr parameter reflects the DSP MADU address.

IN	DSP_AddrConvType	type
	Type of address conversion	

Return Value

ADDRMAP_INVALID	Specified address is not in mapped range
Converted address	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

None.

4.1.12 DSP_ Control

This function is a hook for performing device dependent control operation. This is achieved by calling control function from the attached interface.

Syntax

```
NORMAL_API  DSP_STATUS DSP_control
            (IN  ProcessorId dspId,
             IN  Int32       cmd,
             OPT Pvoid       arg);
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	Int32	cmd
	Command id.	
OPT	Pvoid	arg
	Optional argument for the specified command.	

Return Value

DSP_EINVALIDARG	Invalid arguments specified
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

None.

4.2 HAL Layer

HAL abstract the required hardware features of a DSP.

4.2.1 Device APIs

This APIs are called by DSP Layer as function interface. This APIs calls HAL and PHY APIs to perform the required action.

4.2.1.1 <device>_init

This function Resets the DSP and initializes the components required by DSP. Also puts the DSP in RESET state. Also calls <device>_halInit to initializes the HAL component

Syntax

```
NORMAL_API  DSP_STATUS <device>_init (IN ProcessorId  dspId,
                                      IN DSP_Object * dspState) ;
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	DSP_Object *	dspState
	DSP state Object	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

<device>_exit.

4.2.1.2 <device>_exit

This function resets the DSP and puts it into IDLE Mode. Also calls <device>_halExit to finalizes the HAL component.

Syntax

```
NORMAL_API  DSP_STATUS <device>_exit (IN ProcessorId  dspId,
                                      IN DSP_Object * dspState) ;
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	

IN	DSP_Object *	dspState
	DSP state Object	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

<device>_init.

4.2.1.3 <device>_start

This function causes DSP to start execution from the given DSP address. DSP is put to STARTED state after successful completion. Calls HAL APIs to achieve the required logic.

Syntax

```
NORMAL_API  DSP_STATUS <device>_start (IN ProcessorId  dspId,
                                       IN DSP_Object * dspState,
                                       IN Uint32       dspAddr) ;
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	DSP_Object *	dspState
	DSP state Object	
IN	Uint32	dspAddr
	Address to start execution from	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument

Comments

None.

Constraints

None.

See Also

<device>_stop.

4.2.1.4 <device>_stop

This function stops execution on DSP. DSP is put to STOPPED state after successful completion.

Syntax

```
NORMAL_API  DSP_STATUS <device>_stop (IN ProcessorId  dspId,
                                      IN DSP_Object * dspState) ;
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	DSP_Object *	dspState
	DSP state Object	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument

Comments

None.

Constraints

None.

See Also

<device>_start.

4.2.1.5 <device>_idle

This function idles the DSP. DSP is put to IDLE state after successful completion.

Syntax

```
NORMAL_API  DSP_STATUS <device>_idle (IN ProcessorId  dspId,
                                      IN DSP_Object * dspState) ;
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	DSP_Object *	dspState
	DSP state Object	

Return Value

DSP_EFAIL	All other error conditions

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument

Comments

None.

Constraints

None.

See Also

<device>_start.

4.2.1.6 <device>_intCtrl

This function performs the specified DSP interrupt control activity.

Syntax

```
NORMAL_API  DSP_STATUS <device>_intCtrl
                    (IN         ProcessorId       dspId,
                     IN         DSP_Object     *  dspState,
                     IN         Uint32            intId,
                     IN         DSP_IntCtrlCmd    cmd,
                     IN OUT OPT Pvoid             arg);
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	DSP_Object *	dspState
	DSP state Object	
IN	Uint32	intId
	Interrupt Identifier	
IN	DSP_IntCtrlCmd	Cmd
	Interrupt control command to be performed.	
IN/OUT OPT	Pvoid	Arg
	Optional input/output argument specific to each control command.	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument

Comments

None.

Constraints

None.

See Also

None.

4.2.1.7 <device>_read

This function reads data from DSP.

Syntax

```
NORMAL_API  DSP_STATUS <device>_read
         (IN  ProcessorId  dspId,
          IN  DSP_Object     *  dspState,
          IN  Uint32       dspAddr,
          IN  Endianism    endianInfo,
          IN  Uint32       numBytes,
          OUT Uint8 *      buffer);
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	DSP_Object *	dspState
	DSP state Object	
IN	Uint32	dspAddr
	DSP address to read from.	
IN	Endianism	endianInfo
	endianness of data - indicates whether swap is required or not	
IN	Uint32	numBytes
	Number of bytes to read.	
OUT	Uint8 *	buffer
	Buffer to hold the read data	

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	DSP_setup function wasn't called before calling this function.

Comments

None.

Constraints

None.

See Also

<device>_write

4.2.2 DSP_write

This function writes the data to DSP. This is achieved by calling write function from the attached interface.

Syntax

```
NORMAL_API  DSP_STATUS <device>_write
         (IN  ProcessorId  dspId,
          IN  DSP_Object * dspState,
          IN  Uint32       dspAddr,
          IN  Endianism    endianInfo,
          IN  Uint32       numBytes,
          OUT Uint8 *      buffer);
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	DSP_Object *	dspState
	DSP state Object	
IN	Uint32	dspAddr
	DSP address to write to.	
IN	Endianism	endianInfo
	endianness of data - indicates whether swap is required or not	
IN	Uint32	numBytes
	Number of bytes to write.	
OUT	Uint8 *	buffer
	Buffer to write	

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	DSP_setup function wasn't called before calling this function.

Comments

None.

Constraints

None.

See Also

<device>_read

4.2.3 <device>_ addrConvert

This function converts address between GPP and DSP address space. This is achieved by calling addrConvert function from the attached interface.

Syntax

```
NORMAL_API  <device>_STATUS DSP_addrConvert
                (IN  ProcessorId        dspId,
                 IN  DSP_Object * dspState,
                 IN  Uint32             addr,
                 IN  DSP_AddrConvType   type);
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	
IN	DSP_Object *	dspState
	DSP state Object	
IN	Uint32	addr
	Address to be converted. If DSP address, the addr parameter reflects the DSP MADU address.	
IN	DSP_AddrConvType	type
	Type of address conversion	

Return Value

ADDRMAP_INVALID	Specified address is not in mapped range
Converted address	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

None.

4.2.4 <device>_ Control

This function is a hook for performing device dependent control operation. This is achieved by calling control function from the attached interface.

Syntax

```
NORMAL_API  DSP_STATUS <device>_control
            (IN  ProcessorId dspId,
             IN  DSP_Object * dspState,
             IN  Int32       cmd,
             OPT Pvoid       arg);
```

Arguments

IN	ProcessorId	dspId
	Processor Identifier	

IN	DSP_Object *	dspState
	DSP state Object	
IN	Int32	cmd
	Command id.	
OPT	Pvoid	arg
	Optional argument for the specified command.	

Return Value

DSP_EINVALIDARG	Invalid arguments specified
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

None.

4.2.5 DSP Layer Structures

4.2.5.1 DSP_Interface

This structre defines DSP Function table.

Definition

```
struct DSP_Interface_tag {
    FnDspInit              init            ;
    FnDspExit              exit            ;
    FnDspStart             start           ;
    FnDspStop              stop            ;
    FnDspIdle              idle            ;
    FnDspIntCtrl           intCtrl         ;
    FnDspRead              read            ;
    FnDspWrite             write           ;
    FnDspAddrConvert       addrConvert     ;
    FnDspControl           control         ;
} ;
```

Fields

Init	Function pointer to init function for the DSP.
exit	Function pointer to exit function for the DSP.
start	Function pointer to start function for the DSP.
stop	Function pointer to stop function for the DSP.

idle	Function pointer to idle function for the DSP.
intCtrl	Function pointer to interrupt control function for the DSP.
read	Function pointer to read function for the DSP.
write	Function pointer to write function for the DSP.
addrConvert	Function pointer to address conversion function for the DSP.
Control	Function pointer to device dependent control functionality for the DSP
Instrument	Function pointer to instrument function for the DSP
debug	Function pointer to debug function for the DSP

Comments

Each supported device must export this function table, this function table will be used by DSP layer to program/control the DSP.

Constraints

None.

See Also

None.

4.2.5.2 DSP_Object

This structure defines the state of a DSP.

Definition

```
typedef struct DSP_Object_tag {
    Uint32          dspId        ;
    Pvoid           halObject    ;
    DSP_Interface * interface    ;
#if defined (DDSP_PROFILE)
    DSP_Stats       dspStats     ;
#endif /* if defined (DDSP_PROFILE) */
} DSP_Object  ;
```

Fields

dspId	DSP identifier
halObject	HAL object
interface	Function table for the DSP APIs
dspStats	Profiling information related to the target DSP

Comments

halObject is defined by each DSP separately.

Constraints

None.

See Also

None.

4.2.6 HAL & PHY Layer Structures

4.2.6.1 HAL_Interface

Interface functions exported by the HAL subcomponent.

Definition

```
typedef struct HAL_Interface_tag {
    FnPhyInit  phyInit  ;
    FnPhyExit  phyExit  ;
    FnBootCtrl bootCtrl ;
    FnIntCtrl  intCtrl  ;
    FnMapCtrl  mapCtrl  ;
    FnPwrCtrl  pwrCtrl  ;
    FnRead     read     ;
    FnWrite    write    ;
    FnReadDMA  readDMA  ;
    FnWriteDMA writeDMA ;
} HAL_Interface ;
```

Fields

phyInit	Function pointer to Initializes physical interface function for the DSP.
phyExit	Function pointer to Finalizes physical interface function for the DSP.
bootCtrl	Function pointer to boot control function for the DSP.
intCtrl	Function pointer to interrupt control function for the DSP.
mapCtrl	Function pointer to map control function for the DSP.
intCtrl	Function pointer to interrupt control function for the DSP.
pwrCtrl	Function pointer to power control function for the DSP.
read	Function pointer to read memory function for the DSP.
write	Function pointer to write memory function for the DSP.

readDMA	Function pointer to read DMA function for the DSP.
writeDMA	Function pointer to write DMA function for the DSP.

Comments

Each DSP and HAL must export this structure, if a DSP supports two physical interface, then the DSP must export two of this structure.

Constraints

None.

See Also

None.

4.2.6.2 DSP_ IntCtrlCmd

Defines the types of interrupt control commands; handled by the DSP component.

Definition

```
typedef enum {
    DSP_IntCtrlCmd_Enable    = 0u,
    DSP_IntCtrlCmd_Disable   = 1u,
    DSP_IntCtrlCmd_Send      = 2u,
    DSP_IntCtrlCmd_Clear     = 3u,
    DSP_IntCtrlCmd_WaitClear = 4u,
    DSP_IntCtrlCmd_Check     = 5u
} DSP_IntCtrlCmd ;
```

Fields

DSP_IntCtrlCmd_Enable	Enable interrupt
DSP_IntCtrlCmd_Disable	Disable interrupt
DSP_IntCtrlCmd_Send	Send interrupt
DSP_IntCtrlCmd_Clear	Clear interrupt
DSP_IntCtrlCmd_WaitClear	Wait for interrupt to be cleared
DSP_IntCtrlCmd_Check	Check whether DSP has generated INT or not.

Comments

Each DSP and HAL must export this structure, if a DSP supports two physical interface, then the DSP must export two of this structure.

Constraints

None.

See Also

None.

4.2.6.3 DSP_BootCtrlCmd

Defines the types of boot control commands; handled by the DSP component.

Definition

```
typedef enum {
    DSP_BootCtrlCmd_SetEntryPoint     = 0u,
    DSP_BootCtrlCmd_SetBootComplete   = 1u,
    DSP_BootCtrlCmd_ResetBootComplete = 2u,
} DSP_BootCtrlCmd ;
```

Fields

DSP_BootCtrlCmd_SetEntryPoint	Sets entry point
DSP_BootCtrlCmd_SetBootComplete	Indicate complete of boot sequence
DSP_BootCtrlCmd_ResetBootComplete	Reset the boot complete boot flag.

Comments

None.

Constraints

None.

See Also

None.

4.2.6.4 DSP_ MapCtrlCmd

Defines the types of map control commands; handled by the DSP component.

Definition

```
typedef enum {
    DSP_MapCtrlCmd_Map       = 0u,
    DSP_MapCtrlCmd_Unmap     = 1u,
    DSP_MapCtrlCmd_SetShared = 2u,
} DSP_MapCtrlCmd ;
```

Fields

DSP_MapCtrlCmd_Map	Maps the given dsp address
DSP_MapCtrlCmd_Unmap	Maps the given previous dsp address
DSP_MapCtrlCmd_SetShared	Maps the shared memory to the given

	dsp address

Comments

None.

Constraints

None.

See Also

None.

4.2.6.5 *DSP_PwrCtrlCmd*

Defines the types of power control commands; handled by the DSP component.

Definition

```
typedef enum {
    DSP_PwrCtrlCmd_PowerUp   = 0u,
    DSP_PwrCtrlCmd_PowerDown = 1u,
    DSP_PwrCtrlCmd_Reset     = 2u,
    DSP_PwrCtrlCmd_Release   = 3u,
    DSP_PwrCtrlCmd_PeripheralUp = 4u,
} DSP_PwrCtrlCmd ;
```

Fields

DSP_PwrCtrlCmd_PowerUp	Power the DSP device
DSP_PwrCtrlCmd_PowerDown	Power down the DSP device.
DSP_PwrCtrlCmd_Reset	Reset the DSP device
DSP_PwrCtrlCmd_Rele	Release the DSP device from reset
DSP_PwrCtrlCmd_PeripheralUp	Initialize any peripheral that is used by DSPLink. For example, EDAM/PLL/DDR is needed to be initialized on DM6437 platfrom.

Comments

None.

Constraints

None.

See Also

None.

4.2.6.6 *DSP_DmaCtrlCmd*

Defines the types of DMA control commands; handled by the DSP component.

Definition

```
typedef enum {
    DSP_DmaCtrlCmd_GppToDsp = 0u,
    DSP_DmaCtrlCmd_DspToGpp = 1u
} DSP_DmaCtrlCmd ;
```

Fields

DSP_DmaCtrlCmd_GppToDsp	Start DMA from GPP to DSP
DSP_DmaCtrlCmd_DspToGpp	Start DMA from DSP to GPP

Comments

None.

Constraints

None.

See Also

None.

4.2.7 HAL APIs & PHY APIs

4.2.7.1 <device>_ halInit

This function initializes the HAL object and physical interface. Calls <device>_phyInit to initialize the physical interface.

Syntax

```
NORMAL_API
DSP_STATUS
<device>_halInit (IN Pvoid * halObject, IN Pvoid initParams) ;
```

Arguments

IN	Pvoid *	halObject
	HAL Object	
IN	Pvoid	initParams
	Optional parameters for initialization	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

<device>_halExit.

4.2.7.2 <device>_ halInit

This function finializes the HAL object and physical interface. Calls <device>_phyExit to finalize the physical interface.

Syntax

```
NORMAL_API
DSP_STATUS
<device>_halExit (IN Pvoid * halObject) ;
```

Arguments

IN	Pvoid *	halObject
	HAL Object	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

<device>_halInit.

4.2.7.3 <device>_ phyInit

This function initializes physical interface.

Syntax

```
NORMAL_API
DSP_STATUS
<device>_phyInit (IN Pvoid * halObject) ;
```

Arguments

IN	Pvoid *	halObject
	HAL Object	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

<device>_phyExit.

4.2.7.4 <device>_ phyExit

This function finalizes physical interface.

Syntax

```
NORMAL_API
DSP_STATUS
<device>_phyExit (IN Pvoid * halObject) ;
```

Arguments

IN	Pvoid *	halObject
	HAL Object	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

<device>_phyInit.

4.2.7.5 <device>_ bootControl

This function provides boot control functionality.

Syntax

```
NORMAL_API
DSP_STATUS
<device>_bootControl  (IN          Void *          halObject,
                       IN          DSP_BootCtrlCmd cmd,
                       IN OUT OPT  Pvoid           arg);
```

Arguments

IN	Pvoid *	halObject
	HAL Object	
IN	DSP_BootCtrlCmd	cmd

Boot Command ID.

IN Pvoid arg

Command specific argument (Optional).

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

None.

4.2.7.6 <device>_ intControl

This function provides interrupt control functionality.

Syntax

```
NORMAL_API
DSP_STATUS
<device>_intControl  (IN         Void *          halObject,
                      IN         DSP_ IntCtrlCmd cmd,
                      IN OUT OPT Pvoid           arg);
```

Arguments

IN Pvoid * halObject

HAL Object

IN DSP_ IntCtrlCmd cmd

Interrupt Command ID.

IN Pvoid arg

Command specific argument (Optional).

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

None.

4.2.7.7 <device>_ mapControl

This function provides map control functionality.

Syntax

```
NORMAL_API
DSP_STATUS
<device>_ mapControl (IN          Void *           halObject,
                      IN          DSP_MapCtrlCmd  cmd,
                      IN OUT OPT Pvoid            arg);
```

Arguments

`IN`	`Pvoid *`	`halObject`
	HAL Object	
`IN`	`DSP_MapCtrlCmd`	`cmd`
	Map Command ID.	
`IN`	`Pvoid`	`arg`
	Command specific argument (Optional).	

Return Value

`DSP_EFAIL`	All other error conditions
`DSP_SOK`	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

None.

4.2.7.8 <device>_ pwrControl

This function provides power control functionality.

Syntax

```
NORMAL_API
DSP_STATUS
<device>_pwrControl  (IN          Void *           halObject,
                      IN          DSP_pwrCtrlCmd  cmd,
                      IN OUT OPT Pvoid            arg);
```

Arguments

IN	Pvoid *	halObject
	HAL Object	
IN	DSP_pwrCtrlCmd	cmd
	Power Command ID.	
IN	Pvoid	arg
	Command specific argument (Optional).	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

None.

4.2.7.9 <device>_read

This function to read DSP data.

Syntax

```
NORMAL_API  DSP_STATUS <device>_read
                             (IN  Void *  halObject,
                              IN  Uint32  dspAddr,
                              IN  Uint32  cBytes,
                              OUT Char8 * readBuffer);
```

Arguments

IN	Pvoid *	halObject
	HAL Object	
IN	Uint32	dspAddr
	Address to read from	
IN	Uint32	cBytes
	Number of bytes to be read.	
OUT	Char8 *	readBuffer
	Buffer to hold read data.	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

None.

4.2.7.10 <device>_write

This function to write to DSP memory.

Syntax

```
NORMAL_API  DSP_STATUS <device>_write
                              (IN  Void *  halObject,
                               IN  Uint32  dspAddr,
                               IN  Uint32  cBytes,
                               OUT Char8 * writeBuffer);
```

Arguments

IN	Pvoid *	halObject
	HAL Object	
IN	Uint32	dspAddr
	Address to write to	
IN	Uint32	cBytes
	Number of bytes to be wrtten.	
OUT	Char8 *	readBuffer
	Buffer to containing data.	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

None.

4.2.7.11 <device>_ readDMA

This function DMAs contents from DSP memory to GPP Memory. Here read means DSP write.

Syntax

```
NORMAL_API  DSP_STATUS <device>_readDMA
                              (IN  Void * halObject,
                               IN Uint32  srcAddr,
                               IN Uint32  dstAddr,
                               IN Uint32  size) ;
```

Arguments

`IN`	`Pvoid *`	`halObject`
	HAL Object	
`IN`	`Uint32`	`srcADdr`
	Source address	
`IN`	`Uint32`	`dstAddr`
	Target address	
`OUT`	`Uint32`	`size`
	Number of bytes	

Return Value

`DSP_EFAIL`	All other error conditions
`DSP_SOK`	Operation successfully completed.

Comments

`None.`

Constraints

None.

See Also

None.

4.2.7.12 <device>_ writeDMA

This function DMAs contents from GPP memory to DSP Memory. Here write means DSP read.

Syntax

```
NORMAL_API  DSP_STATUS <device>_writeDMA
                              (IN  Void * halObject,
                               IN Uint32  srcAddr,
                               IN Uint32  dstAddr,
                               IN Uint32  size) ;
```

Arguments

`IN`	`Pvoid *`	`halObject`
	HAL Object	
`IN`	`Uint32`	`srcADdr`
	Source address	
`IN`	`Uint32`	`dstAddr`

	Target address	
OUT	Uint32	size
	Number of bytes	

Return Value

DSP_EFAIL	All other error conditions
DSP_SOK	Operation successfully completed.

Comments

None.

Constraints

None.

See Also

None.

4.3 Dynamic configuration

4.3.1 Data structures

4.3.1.1 CFGMAP_ObjDB

Defines object containing configuration mapping information for all DSPs configured in the DSP/BIOS LINK.

Definition

```
typedef struct CFGMAP_ObjDB_tag {
      Char8 *         dspName ;
      CFGMAP_Object * obj     ;
} CFGMAP_ObjDB ;
```

Fields

dspName	Name of the DSP
obj	CFGMAP object associated with the DSP.

Comments

None.

Constraints

None.

See Also

`CFGMAP_Object`

4.3.2 Functions

4.3.2.1 CFGMAP_ attachObject

This function plugs the CFGMAP object at correct place in CFGMAP_Config array.

Syntax

```
EXPORT_API
DSP_STATUS
CFGMAP_attachObject (IN ProcessorId procId, IN Char8 * dspName) ;
```

Arguments

IN	ProcessorId	procId
	Processor Identifier	
IN	Char8 *	dspName
	Name of the DSP.	

Return Value

DSP_SOK	Operation successfully completed.
DSP_ECONFIG	Incorrect configuration.

Comments

None.

Constraints

None.

See Also

None

4.4 Config

4.4.1 Data structures

4.4.1.1 LINKCFG_Object

This structure defines the configuration structure for the system.

Definition

```
typedef struct LINKCFG_Object_tag {
      LINKCFG_Gpp *       gppObject   ;
      LINKCFG_DspConfig * dspConfigs [MAX_DSPS] ;
} LINKCFG_Object ;
```

Fields

gppObject	Pointer to the GPP specific configuration object.
dspConfigs	DSP/BIOS LINK configuration structures.

Comments

None.

Constraints

None.

See Also

None.

5 Decision Analysis & Resolution

5.1 Platform Configuration

There are two options for platform configuration design.

5.1.1 DAR Criteria

1. Meets customer needs
2. Meets expected requirements for multi-DSP support
3. Ease of use
4. Scalability and flexibility for future usage
5. Ease of porting
6. Consistency with existing DSPLink design and implementation

5.1.2 Available Alternatives

1. Combined configuration file
2. Individual C configuration files

5.1.2.1 Combined configuration file

Summary:

- Here full system architecture would be captured in a single file.
- For this, there would be templates available, containing default values for all supported platforms.
- Static build configuration script would use these templates and generate the CFG_arch.c.

Advantages:

1. Backward compatibility to some level with existing DSPLink, since currently customers are used to only having one configuration file for each platform.
2. Will reduce confusion for customers having to move from earlier non-multi-DSP DSPLink versions to the new version.

Disadvantages:

1. Not intuitive, since now configurations are really different for GPP and each DSP.
2. Since each PROC_attach must take a different configuration for each DSP, having all the objects in a single file is not intuitive.
3. Not scalable and flexible to support multiple types of combinations
4. Static configuration script is required to do parsing of the templates and generate file. This parsing will require more effort.
5. Porting to a different platform combination would take more effort

5.1.2.2 Individual C configuration files

Summary:

1. Every supported DSP device's configuration values would be provided in a separate file.
2. GPP configuration values are also provided in a separate file
3. Static configuration script would generate a file CFG_system.c which will tie the full system architecture.

Advantages:

1. Intuitive usage of configuration files with separate configuration for each logical entity.
2. Scalable and flexible to support multiple types of configurations
3. Simpler logic in static configuration script and easy maintenance.
4. Porting to a different platform combination would be simpler.

Disadvantages:

1. Multiple files represent the configuration, so it may confuse users which file to modify. This can be mitigated by having all files used in a specific configuration copied to a BUILD separate location so that it is clear which files are involved in the build.

5.1.3 Decision

Alternative 2 has been chosen based on the advantages and disadvantages listed for each approach.