DESIGN DOCUMENT

DSP/BIOS™ LINK

Configurable TSK and SWI approach

LNK 207 DES

Version <1.00>

Template Version 1.2

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

No table of figures entries found.

1 Introduction

1.1 Purpose & Scope

This document describes the design to configure the TSK or SWI mode for the existing SWI functions of ZCPYMQT and ZCPYDATA. If MPCS protection is TSK-base, then DSPLink MSGQ and CHNL drivers will use TSK Mode on DSP-side. If MPCS protection is SWI-base, then DSPLink MSGQ and CHNL drivers will use SWI Mode. So that systems are fully TSK-based or SWI based.

1.2 Terms & Abbreviations

DSPLINK	DSP/BIOS™ LINK
SWI	Software interrupt manager
TSK	Task manager
⚐	This bullet indicates important information. Please read such text carefully.
❑	This bullet indicates additional information.

1.3 References

1.	Spru404n.pdf	TMS320C55x DSP/BIOS 5.32 Application Programming Interface (API) Reference Guide
2.	LNK_041_DES.pdf	ZERO COPY LINK DRIVER

1.4 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 (GPP) 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.

This document gives an overview of the SWI and TSK mode detailed design for DSPLINK.

2 Requirements

DSPLN00001021:- DSPLink should use configurable TSK-Sem or SWI-enable and SWI-disable approach for MPCS.

Right now in both the cases (SWI and TSK) DSPLINKs components MQT and CHNL works in SWI mode only. Ideally for TSK mode the components should work in context of Task.

To support this feature following changes are required:-

- Select the mode while configure the DSPLINK. Mode can be SWI or TSK (DSP_SWI_MODE or DSP_TSK_MODE).
- Handle the components properly for both modes.
- Don't disable the scheduler by calling TSK_disable. Use semaphore SEM_pend and SEM_post.

3 High Level Design

The zero-copy driver provides a fast physical link between the GPP and the DSP, based on the concept of pointer exchange between the GPP and DSP applications. For data transfer, the link driver manages a configurable number of logical channels. The IPS component manages the transfer of data and messages across the two processors. For this, it uses the shared memory control structure and interrupts between the processors to inform about any changes in status of buffer/message availability on the channels.

The IPS component shall maintain lists of messages, which are shared between the GPP and the DSP. There shall be two unidirectional lists of messages, for messages to and from the DSP. Similar lists shall also be used for data transfer. To protect these shared lists, the IPS component shall utilize the services of a generic component that shall provide critical section protection between the two processors.

In a multiprocessor system having shared access to a memory region, a multiprocessor critical section between GPP and DSP can be implemented. This MPCS object can be used by applications to provide mutually exclusive access to a shared region between multiple processors, and multiple processes on each processor.

Following functionality added to support TSK mode:-

1. Create a task to execute (TSK_create).
2. Initializes the semaphore object
3. Wait and signal a semaphore (SEM_pend() and SEM_post()).
4. ZCPYMQT and ZCPYDATA in TSK context.
5. Delete a task (TSK_delete).
6. Configuration and make system changes.

3.1 Create a task to execute (TSK_create)

The TSK object is created during the ZCPYMQT_open phase for TSK mode by calling TSK_Create. When ZCPY MQT is opened and configured, at that time create the TSK object.

```
Static int ZCPYMQT_open (MSGQ_TransportHandle mqtHandle)
{
.
.
#if defined (DSP_TSK_MODE)
      tskAttrs. priority=15 ;
      mqtState->tskHandle = TSK_create(ZCPYMQT_tskFxn, &tskAttrs, 0) ;
      if (mqtState->tskHandle == NULL) {
          status = SYS_EALLOC ;
          SET_FAILURE_REASON (status) ;
}
#endif
```

Create the static TSK object for ZCPYDATA, use the following steps.

```
var ZCPYLINK_TSK_OBJ= bios.TSK.create("ZCPYLINK_TSK_OBJ");
/* To create a TSK object*/
ZCPYLINK_TSK_OBJ.comment = " This TSK handles the data transfer in
DSPLINK";
ZCPYLINK_TSK_OBJ.autoAllocateStack = true;
/* Check this box if you want the task's private stack space to be allocated
automatically */
ZCPYLINK_TSK_OBJ.priority = 15;
/* The priority level for this task. */
ZCPYLINK_TSK_OBJ.fxn = prog.extern("ZCPYDATA_tskFxn");
/* The function to be executed when the task runs. */
ZCPYLINK_TSK_OBJ.arg0 = prog.decl("ZCPYDATA_devObj");
/* Task function argument 0-7 */
```

3.2 Initializes the semaphore object

SEM_new () initializes the semaphore object pointed to by sem with count. The function should be used on a statically created semaphore for initialization purposes only.

Create and initialize the semaphore for MPCS:-

```
Int
_MPCS_open (IN     Uint16         procId,
            IN     Char *         name,
            OUT    MPCS_Handle *  mpcsHandle,
            IN OPT MPCS_ShObj *   mpcsShObj)
.
.
#if defined (DSP_TSK_MODE)
            if((*mpcsHandle)->dspMpcsObj.localLock == NULL) {
                (*mpcsHandle)->dspMpcsObj.localLock      =      (Uint32)
SEM_create(1, NULL) ;
                if ((*mpcsHandle)->dspMpcsObj.localLock == NULL) {
                    status = SYS_EALLOC ;
                }
                else {
                    HAL_cacheWbInv ((Ptr) &((*mpcsHandle)->dspMpcsObj),
                                sizeof (MPCS_ProcObj)) ;
                }
            }
```

```
#endif /* if defined(DSP_TSK_MODE) */
```

Create and initialize the semaphore for ZCPYMQT:-

```
Static Int ZCPYMQT_open (MSGQ_TransportHandle mqtHandle)
{
.
.
#if defined (DSP_TSK_MODE)
    SEM_new (&(mqtState->zcpyMqtSem), 0) ;
#endif
}
```

Create and initialize the semaphore for ZCPYDATA:-

```
Void
ZCPYDATA_mdBindDev (Ptr * devp, Int devid, Ptr devParams)
{
.
.
#if defined (DSP_TSK_MODE)
    SEM_new (&(ZCPYDATA_devObj.zcpyDataSem), 0) ;
#endif
}
```

3.3 Wait and signal a semaphore

In case of TSK context. SEM_pend and SEM_post will control the task processing. Initially it calls SEM_pend to acquire the semaphore if it is available and tries to get the multiprocessor lock. SEM_pend and SEM_post are use with counting semaphores, which keep track of the number of times the semaphore has been posted.

The MPCS component in TSK context:-

MPCS_enter calls SEM_pend to acquire the semaphore.

```
Int MPCS_enter (IN     MPCS_Handle mpcsHandle)
{
    Int status = SYS_OK ;
#if defined (DDSP_PROFILE)
    Bool conflictFlag = FALSE ;
#endif
    DBC_require (mpcsHandle != NULL) ;
    if (mpcsHandle == NULL) {
        status = SYS_EINVAL ;
        SET_FAILURE_REASON (status) ;
    }
    else {
#if defined (DSP_TSK_MODE)
        SEM_pend (&(mpcsHandle->dspMpcsObj.localLock), SYS_FOREVER) ;
        .
#else
        SWI_disable () ;
        .
#endif
```

MPCS_leave call SEM_post to post the semaphore to allow the others that are waiting or blocked in MPCS_enter.

```
Int MPCS_leave (IN     MPCS_Handle mpcsHandle)
{
    Int status = SYS_OK ;
    DBC_require (mpcsHandle != NULL) ;
    if (mpcsHandle == NULL) {
        status = SYS_EINVAL ;
        SET_FAILURE_REASON (status) ;
    }
    else {
        /* Check if DSP side is using the resource i.e.there has been a
         * corresponding MPCS_enter.
         */
        if (mpcsHandle->dspMpcsObj.flag == (Uint16) MPCS_BUSY) {
            /* Release the resource. */
            mpcsHandle->dspMpcsObj.flag = (Uint16) MPCS_FREE ;

            HAL_cacheWbInv ((Ptr) &(mpcsHandle->dspMpcsObj),
                            sizeof (MPCS_ProcObj)) ;
#if defined (DSP_TSK_MODE)
            SEM_post (mpcsHandle->dspMpcsObj.localLock) ;
            .
#else
            SWI_enable () ;
            .
#endif
.
.
}
```

3.4 ZCPYMQT and ZCPYDATA in TSK context

When either the GPP or DSP is ready to send a message to the other processor, it sends the message to the IPS component. On receiving a message from the other processor, the IPS component makes a call back to the ZCPY MQT and DATA component, which places the received message onto the appropriate local message queue.

The Callback functions (ZCPYDATA_callback and ZCPYMQT_callback) are registered with IPS component. These Callback functions will call SEM_post to post the semaphore to allow the others that are waiting or blocked in ZCPYMQT_tskFxn and ZCPYDATA_tskFxn.

In case of ZCPYMQT :-

```
Static Void ZCPYMQT_callback (Uint32 eventNo, Ptr arg, Ptr info)
{
    ZCPYMQT_State * mqtState = (ZCPYMQT_State *) arg ;
    (void) eventNo ;
    (void) info ;
    DBC_assert (mqtState != NULL) ;
#if defined (DSP_TSK_MODE)
    SEM_post (&(mqtState->zcpyMqtSem)) ;
#else
    SWI_post (mqtState->swiHandle) ;
#endif
}
```

In case of ZCPYDATA :-

```
Static Void ZCPYDATA_callback (Uint32 eventNo, Ptr arg, Ptr  info)
{
    (void) eventNo ;
    (void) info ;
#if defined (DSP_TSK_MODE)
    ZCPYDATA_DevObject * dev      = (ZCPYDATA_DevObject *)arg ;
    SEM_post (&(dev->zcpyMqtSem)) ;
#else
    (void) arg ;
    SWI_inc (&ZCPYDATA_SWI_OBJ) ;
#endif
}
```

ZCPYMQT_tskFxn and ZCPYDATA_tskFxn are register for TSK mode and both functions will call the SEM_pend to wait the semaphore.

In case of ZCPYMQT :-

```
static
Void
ZCPYMQT_tskFxn (Arg arg0, Arg arg1)
{
    Int                     status = SYS_OK ;
.
.
    DBC_require (arg0 != NULL) ;

    (Void) arg1 ;
    mqtState = (ZCPYMQT_State *) arg0 ;
.

/*While (1) is to make the task continuously active*/
    While(1) {
        SEM_pend(&(mqtState->zcpyMqtSem), SYS_FOREVER) ;
        .
    }
.
.
}
```

In case of ZCPYDATA :-

```
Void ZCPYDATA_tskFxn (Arg arg0, Arg arg1)
{
    ZCPYDATA_DevObject *     dev     = (ZCPYDATA_DevObject *) arg0 ;
.
.
    (Void) arg1 ;

    DBC_require (dev != NULL) ;
/*While (1) is to make the task continuously active*/
    While(1) {
        SEM_pend(&(ZCPYDATA_devObj.zcpyDataSem), SYS_FOREVER) ;
        .
    }
.
}
```

3.5 Delete a task (TSK_delete)

The TSK and SWI objects are deleted by calling SWI_delete and TSK_delete. When ZCPYMQT_close is called delete the objects. ZCPYMQT_close Closes the ZCPY MQT, and cleans up its state object.

In case of ZCPYMQT:-

```
static
Int
ZCPYMQT_close (MSGQ_TransportHandle mqtHandle)
{
    Int               status  = SYS_OK ;
    QUE_Handle        queHandle ;
    ZCPYMQT_State *   mqtState ;
    MSGQ_Msg          msg ;

    DBC_require (mqtHandle != NULL) ;
.
.

#if defined (DSP_TSK_MODE)
    if (mqtState->tskHandle != NULL) {
        TSK_delete (mqtState->tskHandle) ;
    }
#else
    if (mqtState->swiHandle != NULL) {
        SWI_delete (mqtState->swiHandle) ;
    }
#endif
```

3.6 Configuration and make system changes

To make it configurable need to export the mode e.g DSP_SWI_MODE or DSP_TSK_MODE. Using dsplinkcfg.pl mode can be exported e.g.

```
****************** ADVICE !!! ***************************
To enable TSK mode select: --DspTskMode=1
Provided:
Assuming SWI mode enable and continuing...
==========================================
```

```
perl      dsplinkcfg.pl      --platform=DAVINCIHD      --nodsp=1      --
dspcfg_0=DM6467GEMSHMEM      --dspos_0=DSPBIOS5XX      --gppos=MVL5G   --
comps=ponslrmc --DspTskMode=1
```

or

```
perl dsplinkcfg.pl --platform=DAVINCIHD --nodsp=1 --
dspcfg_0=DM6467GEMSHMEM --dspos_0=DSPBIOS5XX --gppos=MVL5G --
comps=ponslrmc
```

In case of TSK Mode the CURRENTCFG.mk :-

```
#======================================================================
#   DSP SWI/TSK MODE SPECIFIC DEFINES
# =====================================================================
export  TI_DSPLINK_DM6467GEM_MODE := DSP_TSK_MODE

#======================================================================
#   DSP SPECIFIC DEFINES
#======================================================================
export TI_DSPLINK_DSP0_DEFINES := PROCID=0 OMAP2530
OMAP2530_INTERFACE=SHMEM_INTERFACE PHYINTERFACE=SHMEM_INTERFACE
DSP_TSK_MODE
```

In case of SWI Mode the CURRENTCFG.mk :-

```
export  TI_DSPLINK_DSP_MODE := DSP_SWI_MODE

#======================================================================
#   DSP SPECIFIC DEFINES
#======================================================================
export TI_DSPLINK_DSP0_DEFINES := PROCID=0 OMAP2530
OMAP2530_INTERFACE=SHMEM_INTERFACE PHYINTERFACE=SHMEM_INTERFACE
DSP_SWI_MODE
```

3.7 Impact and Backward Compatibility:

Earlier ZCPYDATA and ZCPYMQT worked in SWI context only. Now both can work in either TSK mode or SWI mode.

In case of ZCPYDATA, SWI and TSK are created statically in applications e.g.

SWI :- loop.tcf and scale.tcf.

TSK :- loop_tsk.tcf and scale_tsk.tcf.

For SWI support there is no change in samples.

If user wants to ZCPYDATA worked in task mode following changes are required:-

Create new loop_tsk.tcf for all platforms .It will include the dsplink-zcpydata-tsk.tci.

```
/* ====================================================================
 *  Load generic DSP/BIOS Link configuration
 * ====================================================================
 */
utils.importFile ("dsplink- omap2530-base.tci");
utils.importFile ("dsplink-iom.tci");
utils.importFile ("dsplink-zcpydata-tsk.tci");
.
.
.
```

Create new scale_tsk.tcf for all platforms. It will include the dsplink-zcpydata-tsk.tci.

```
/* ===================================================
 *  Load generic DSP/BIOS Link configuration
 *  ===================================================
 */
utils.importFile ("dsplink- omap2530-base.tci");
utils.importFile ("dsplink-iom.tci");
utils.importFile ("dsplink-zcpydata-tsk.tci");
.
.
.
```

Static TSK objects will be created in dsplink-zcpydata-tsk.tci for ZCPYDATA. Now we have two tcf file one for swi and other for tsk (loop.tcf and loop_tsk.tcf). And these files are included according to compilation check for DSP_SWI_MODE or DSP_TSK_MODE.

Changes in sample’s SOURCES file of loop and scale samples for all platforms :-

```
#====================================================
#TCONF configurations file (from component base path)
#====================================================
ifeq ("$(TI_DSPLINK_DSP_MODE)", "DSP_TSK_MODE")
TCF_FILE                                                      :=
$(TI_DSPLINK_DSPOS)$(DIRSEP)$(TI_DSPLINK_DSPOSVERSION)$(DIRSEP)$(TI_DSP
LINK_DSPDEVICE)$(DIRSEP)loop_tsk.tcf
else
TCF_FILE                                                      :=
$(TI_DSPLINK_DSPOS)$(DIRSEP)$(TI_DSPLINK_DSPOSVERSION)$(DIRSEP)$(TI_DSP
LINK_DSPDEVICE)$(DIRSEP)loop.tcf
endif
```

4 Typedefs & Data Structures

4.1 ZCPYMQT_State

This structure defines the ZCPYMQT state object, which contains all the component-specific information.

Definition

```
typedef struct ZCPYMQT_State_tag {
    Uint16          poolId        ;
    QUE_Obj         ackMsgQueue   ;
    Uint32          ipsId         ;
    Uint32          ipsEventNo    ;
    ZCPYMQT_Ctrl *  ctrlPtr       ;
#if defined (DSP_TSK_MODE)
    TSK_Handle      tskHandle     ;
#else
    SWI_Handle      swiHandle     ;
#endif
    SEM_Obj           zcpyMqtSem  ;
} ZCPYMQT_State ;
```

Fields

`poolId`
Pool ID used for allocating control messages. This pool is also used in case the ID within the message received from the DSP is invalid. This can occur in case of a mismatch between pools configured on the GPP and the DSP.

`ackMsgQueue`
Queue of `locateAck` messages received from the GPP.

`ipsId`
IPS ID associated with MQT.

`ipsEventNo`
IPS Event no associated with MQT.

`swiHandle`
SWI for processing of locate functionality in non-ISR context.

Only defined if callback processing is to be performed within a SWI instead of interrupt context.

`tskHandle`
Only defined if callback processing is to be performed within a TSK context.

`zcpyMqtSem`
Zero copy semaphore object.

Comments

An instance of this object is created and initialized during `ZCPYMQT_open ()`, and its handle is returned to the caller. It contains all information required for maintaining the state of the MQT.

Constraints

None.

See Also

`ZCPYMQT_open ()`

4.2 ZCPYDATA_DevObject

LINK device object structure.

Definition

```
typedef struct ZCPYDATA_DevObject_tag {
    Uns                      inUse            ;
    Uns                      devId            ;
    Uint32                   ipsId            ;
    Uint32                   ipsEventNo       ;
    Uns                      numChannels      ;
    Uns                      outputMask       ;
    Uns                      ongoingOutputMask ;
    Uns                      lastOutputChannel ;
    ZCPYDATA_Ctrl *          ctrlPtr          ;
    ZCPYDATA_ChannelObject * chnlObj          ;
#if defined (DSP_TSK_MODE)
    SEM_Obj                  zcpyDataSem ;
#endif
} ZCPYDATA_DevObject ;
```

Fields

`inUse`	Non zero value means this LINK device is in use..
`devId`	Data driver ID.
`ipsId`	IPS ID associated with the data driver.
`ipsEventNo`	IPS event number associated with the data driver.
`numChannels`	Maximum channels supported by this device.
`outputMask`	Indicates on which channels output buffer available.
`ongoingOutputMask`	Indicates on which channels output data transfer is ongoing
`lastOutputChannel`	Variable indicating on which channel last output was done
`ctrlPtr`	Pointer to shared memory control structure
`chnlObj`	Array of channel objects that belong to this device.
`zcpyDataSem`	ZCPYDATA semaphore object.

Comments

An instance of this object is initialized during `ZCPYDATA_init ()`.

Constraints

None.

See Also

None.

5 API Definition

5.1 ZCPYMQT_tskFxn

Implements the TSK function for the ZCPYMQT.

Syntax

```
Static Void ZCPYMQT_tskFxn (Arg arg0) ;
```

Arguments

```
IN       Arg                          arg0 ;
```

ZCPYMQT state object, which contains all the component-specific information.

Return Value

```
void
```

Comments

SEM_pend is called to wait for semaphore. This function will call the ZCPYMQT_msgCtrl () to transfer the data in DSPLINK.

Constraints

None.

See Also

None.

5.2 ZCPYDATA_tskFxn

Implements the TSK function for the ZCPYDATA.

Syntax

```
Static Void ZCPYDATA_tskFxn (Arg arg0) ;
```

Arguments

```
IN       Arg                          arg0 ;
```

Pointer to LINK device structure.

Return Value

```
void
```

Comments

SEM_pend is called to wait for semaphore. This function will call the ZCPYDATA_dataCtrl () to transfer the data in DSPLINK.

Constraints

None.

See Also

None.

5.3 ZCPYMQT_msgCtrl

Message control function for the SWI and TSK.

Syntax

Static Void ZCPYMQT_msgCtrl (ZCPYMQT_State * mqtState) ;

Arguments

```
IN      Arg                         arg0 ;
```

Pointer to LINK device structure.

Return Value

```
void
```

Comments

Make locate request to remote MQT by sending event to IPS containing control message.

If the locate call is synchronous, wait for receiving locate acknowledgement message from remote MQT as an IPS event containing control message.

If the locate call is asynchronous, return from the function without blocking. When the locate acknowledgement arrives from the remote processor, allocate and send an asynchronous locate message to the reply message queue specified by the caller.

Constraints

None.

See Also

None.

5.4 ZCPYDATA_dataCtrl

Message control function for the SWI and TSK.

Syntax

Static Void ZCPYDATA_dataCtrl (ZCPYMQT_State * mqtState) ;

Arguments

```
IN      Arg                         arg0 ;
```

Pointer to LINK device structure.

Return Value

```
void
```

Comments

The ZCPY DATA transfer component uses the features provide by the IPS component for transferring the data between the DSP and GPP. Both the GPP and DSP issue buffers for data transfer and the ZCPY driver exchanges the buffer pointers to complete the transfer..

Constraints

None.

See Also

None.