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

Mailing Address:
Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

This page has been intentionally left blank.

TABLE OF CONTENTS

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.

Presently, DSPLINK MSGQ and CHNL components work in SWI mode only. For TSK mode the components should work in context of a Task.

To support this feature following changes are required within DSPLink:

- 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 Assumptions

The MPCS design makes the following assumptions:

1. The hardware allows provision of a buffer pool, to which both the GPP and the DSP have access.

4 Constraints

The MPCS object must be allocated and freed through POOL APIs provided by DSPLINK. Elements allocated through the POOL API can be accessed by multiple processors. Any other means for memory allocation (for example: standard OS calls) will fail as the elements cannot be accessed across processors.

The user has to use unique identifier to identify individual MPCS objects across the system.

5 Low 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.

5.1 Following functionality added to support TSK mode:-

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

5.1.1 Create a task to execute (Using TSK_create call)

The TSK objects are created during the ZCPYMQT_open phase for TSK mode by calling TSK_Create. Create the task in the initial ZCPYMQT_open function.

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

Create the static TSK objects 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_TSK");
/* The function to be executed when the task runs. */
ZCPYLINK_TSK_OBJ.arg0 = 0;
/* Task function argument 0-7 */
```

5.2 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_create (IN     Uint16         procId,
             IN     Char *         name,
             IN OPT MPCS_ShObj *   mpcsObj,
             IN     MPCS_Attrs *   attrs)
{
.
.
#if defined (USE_TSK)
    SEM_Obj           mpcsSem ;
    #if defined (USE_TSK)
        SEM_new (&mpcsSem, 0) ;
    #endif
#endif
}
```

Create and initialize the semaphore for ZCPYMQT:-

```
Static Int ZCPYMQT_open (MSGQ_TransportHandle mqtHandle)
{
.
.
#if defined (USE_TSK)
    SEM_Obj           zcpyMqtSem ;
    #if defined (USE_TSK)
        SEM_new (&zcpyMqtSem, 0) ;
    #endif
#endif
}
```

Create and initialize the semaphore for ZCPYDATA:-

```
Void
ZCPYDATA_init ()
{
.
.
#if defined (USE_TSK)
    SEM_Obj           zcpyDataSem ;
    #if defined (USE_TSK)
        SEM_new (&zcpyDataSem, 0) ;
    #endif
#endif
}
```

5.3 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 (USE_TSK)
    While(1) {
        SEM_pend(&mpcsSem, SYS_FOREVER) ;
        .
        .
        .
    }
#endif
```

MPCS_leave will 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 (USE_TSK)
            SEM_post(mpcsSem) ;
#endif
.
.
.
}
```

5.4 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 notification 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 (USE_TSK)
    SEM_post(zcpyMqtSem) ;
#endif
#if defined (USE_SWI)
    SWI_post (mqtState->swiHandle) ;
#endif
}
```

In case of ZCPYDATA :-

```
Static Void ZCPYDATA_callback (Uint32 eventNo, Ptr arg, Ptr  info)
{
    (void) eventNo ;
    (void) arg ;
    (void) info ;
#if defined (USE_TSK)
    SEM_post((&ZCPYDATA_TSK_OBJ) ;
#endif
#if defined (USE_SWI)
```

```
    SWI_inc (&ZCPYDATA_SWI_OBJ) ;
#endif
}
```

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

In case of ZCPYMQT :-

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

    (Void) arg1 ;
    mqtState = (ZCPYMQT_State *) arg0 ;
.
    HAL_cacheInv ((Ptr) &(ctrlPtr->toDspList), sizeof (ctrlPtr-
>toDspList)) ;

#if defined (USE_TSK)
    While(1) {
        SEM_pend(&zcpyMqtSem, SYS_FOREVER) ;
        .
    }
.
#endif
```

In case of ZCPYDATA :-

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

    DBC_require (dev != NULL) ;
#if defined (USE_TSK)
    While(1) {
        SEM_pend(&zcpyDataSem, SYS_FOREVER) ;
        .
    }
.
#endif
}
```

5.5 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 (USE_SWI)
    if (mqtState->swiHandle != NULL) {
        SWI_delete (mqtState->swiHandle) ;
    }
#endif /* if defined (USE_SWI) */

#if defined (USE_TSK)
if (mqtState->tskHandle != NULL) {
    TSK_delete (mqtState->tskHandle) ;
}
#endif
```

5.6 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 For 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
```

6 Typedefs & Data Structures

6.1 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 (USE_SWI)
    SWI_Handle      swiHandle     ;
#endif /* if defined (USE_SWI) */
#if defined (USE_TSK)
    TSK_Handle      tskHandle     ;
#endif /* if defined (USE_TSK) */

} 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.

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 ()`

7 API Definition

7.1 ZCPYMQT_tskFxn

Implements the TSK function for the ZCPYMQT.

Syntax

Static Void ZCPYMQT_tskFxn (Arg arg0, Arg arg1) ;

Arguments

```
IN      Arg                        arg0 ;
```

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

Return Value

```
void
```

Comments

Make locate request to remote MQT by sending event to SHMIPS containing control message. If the locate call is synchronous, wait for receiving locate acknowledgement message from remote MQT as an SHMIPS 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.

7.2 ZCPYDATA_TSK

TSK function for data transfer in DSPLINK..

Syntax

Static Void ZCPYDATA_TSK (Arg arg0, Arg arg1) ;

Arguments

```
IN      Arg                        arg0 ;
```

Pointer to LINK device structure.

Return Value

```
void
```

Comments

Functionality will remain same as ZCPYDATA_SWI. Only the change is waiting for semaphore.

Constraints

None.

See Also

None.

8 Impact and Backward Compatibility:

By default, the DSP-side of DSPLink uses SWIs for MSGQ and CHNL physical transports for communication with the peer transports on the GPP-side. Accordingly, the MPCS (Multi-processor critical section) protection uses SWI_disable/SWI_enable to protect from other local threads, since the DSPLink APIs need to be callable from SWI context.

In pure TSK-only systems, this release now supports applications that wish to reduce the scheduler disable latency. For this, the MPCS implementation supports usage of semaphore for protection from other local threads. Accordingly, the MSGQ and CHNL modules use TSK based servers instead of SWIs for communication with the peer transport on GPP. With this change, the scheduler disable latency gets reduced; however applications must not make any DSPLink calls from SWI context.

In both modes, DSPLink calls from HWI context continue to not be supported. The choice of whether TSK mode or SWI mode is to be used, is selectable for each DSP in the system from the DSPLink static build configuration script dsplinkcfg.pl.”

If application writer is using the DSPLink build system and the DSPLink shipped tci files, the application writer can look at the <sample_tsk> files for reference

The changes that application needs to make to move from the default SWI mode to task mode are as follows:

- Remove static creation of SWI related configuration from application TCF file i.e. the code shown below.

```
 var dsplink          = prog.module("UDEV").create("dsplink");
 dsplink.params       = prog.decl("DSPLINK_DEV_PARAMS");
 dsplink.initFxn      = prog.decl("DSPLINK_init");
 dsplink.fxnTable     = prog.decl("DSPLINK_FXNS");
 dsplink.fxnTableType = "IOM_Fxns";
 dsplink.comment      = "DSP/BIOS LINK  - IOM Driver";

  var ZCPYLINK_SWI_OBJ      =
prog.module("SWI").create("ZCPYDATA_SWI_OBJ");
  ZCPYLINK_SWI_OBJ.comment  = "This swi handles the data transfer in
DSPLINK";
  ZCPYLINK_SWI_OBJ.fxn      =  prog.decl("ZCPYDATA_SWI");
  ZCPYLINK_SWI_OBJ.priority = 14;
  ZCPYLINK_SWI_OBJ.arg0     =  prog.decl("ZCPYDATA_devObj");
```

- Add the following TSK mode related code in the application TCF file for CHNL component

```
   var dsplink = prog.module("UDEV").create("dsplink");
  dsplink.initFxn      = prog.decl("ZCPYDATA_init");
  dsplink.fxnTable     = prog.decl("ZCPYDATA_FXNS");
  dsplink.fxnTableType = "IOM_Fxns";
  dsplink.comment      = "DSP/BIOS LINK  - IOM Driver";

  var ZCPYLINK_TSK_OBJ      = bios.TSK.create("ZCPYLINK_TSK_OBJ");
  ZCPYLINK_TSK_OBJ.comment  = "This tsk handles the data transfer in
DSPLINK";
  ZCPYLINK_TSK_OBJ.autoAllocateStack = true;
  ZCPYLINK_TSK_OBJ.priority = 15;
  ZCPYLINK_TSK_OBJ.fxn = prog.extern("ZCPYDATA_tskFxn");
  ZCPYLINK_TSK_OBJ.arg0     =  prog.decl("ZCPYDATA_devObj");
```

- Ensure that the creation of the DIO driver remains unchanged in the application TCF file

```
var dio_dsplink = prog.module("DIO").create("dio_dsplink");
dio_dsplink.comment = "DSP/BIOS LINK  - DIO Driver";
dio_dsplink.deviceName = prog.get("dsplink");
```

- Update priorities of the ZCPYMQT and ZCPYDATA task as per application integrator and system requirements.

- The application writer may need to set the STACKSEG for dynamic TSK_create calls in the application TCF file.

```
bios.TSK.STACKSEG = <some memory seg>;    // for dynamic TSK stacks
```

- Ensure that the applications do not call DSPLink API’s from ISR or SWI context.

In TSK mode, it is not permitted to call DSPLink APIs from ISR or SWI context. In default SWI mode, it is not permitted to call DSPLink APIs from ISR context.

Calling DSPLink API’s from ISR or SWI context could lead to system deadlock

System deadlock could occur when:

- DSP-side executing in a task, takes an MPCS lock (through DSPLink API call), gets preempted by a SWI, which also tries to take an MPCS lock by calling a DSPLink API (not allowed)

- DSP-side executing in a task, takes an MPCS lock (through DSPLink API call), gets preempted by an ISR, which also tries to take an MPCS lock by calling a DSPLink API (not allowed)

- DSP-side executing in a SWI, takes an MPCS lock (through DSPLink API call), gets preempted by an ISR, which also tries to take an MPCS lock by calling a DSPLink API (not allowed)