DSP/BIOS™ LINK

IPS & Notify

LNK 128 DES

Version 0.30

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

1 Introduction 7
1.1 Purpose & Scope 7
1.2 Terms & Abbreviations 7
1.3 References 7
1.4 Overview 7
2 Requirements 8
3 Assumptions 9
4 Constraints 10
5 High Level Design 11
5.1 IPS 11
5.2 Notify 13
6 Sequence Diagrams 15
6.1 IPS_init 16
6.2 IPS_exit 17
6.3 IPS_register 18
6.4 IPS_unregister 19
6.5 IPS_notify 20
6.6 IPS_ISR 21
6.7 _NOTIFY_init 22
6.8 _NOTIFY_exit 23
6.9 NOTIFY_eventWorker 24
6.10 NOTIFY_register 25
6.11 NOTIFY_unregister 26
7 Low Level Design 27
7.1 IPS 27
7.2 Notify 38
8 Internal Discussions 47
8.1 Prioritizing IPS (High level design) 47
8.2 Prioritizing IPS (Low level discussion) 51
8.3 Notify 54
8.4 IPS component (Older Implementation) 56

TABLE OF FIGURES

Figure 1. Block diagram for IPS component 12
Figure 2. On the GPP: IPS_init () control flow 16
Figure 3. On the GPP: IPS_exit () control flow 17
Figure 4. On the GPP: IPS_register () control flow 18
Figure 5. On the GPP: IPS_unregister () control flow 19
Figure 6. On the GPP: IPS_notify () control flow 20
Figure 7. On the GPP: IPS_ISR () control flow 21
Figure 8. On the GPP: _NOTIFY_init () control flow 22
Figure 9. On the GPP: _NOTIFY_exit () control flow 23
Figure 10. On the GPP: NOTIFY_eventWorker () control flow 24
Figure 11. On the GPP: NOTIFY_register () control flow 25
Figure 12. On the GPP: NOTIFY_unregister () control flow 26
Figure 13. Register an Event 47
Figure 14. Event chart 48
Figure 15. Sending an Event 49
Figure 16. Receiving an Event 49
Figure 17. Sending an Event 50
Figure 18. Receiving an Event 50
Figure 19. Location of Notify module 54
Figure 20. Event passed to the user processes 55
Figure 21. IPS Alternative 1: Shared memory layout 57
Figure 22. IPS Alternative 2: Shared memory layout 64

1 Introduction

1.1 Purpose & Scope

This document describes the design of zero copy link driver for DSP/BIOS™ LINK.

The document is targeted at the development team of DSP/BIOS™ LINK.

1.2 Terms & Abbreviations

DSPLINK	DSP/BIOS™ LINK
RT	RealTime
OS	Operating System
CHNL	Data Channel
MSGQ	Message Queue
DPC	Deferred Procedure Call
IPS	Inter processor Signaling
⚑	This bullet indicates important information. Please read such text carefully.
❑	This bullet indicates additional information.

1.3 References

1.	LNK 012 DES	DSP/BIOS™ LINK Link Driver Version 1.11, dated JUL 25, 2003

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.

The IPS provides signaling facility between the GPP and the DSP.

This document provides a detailed description of IPS and Dispatch scheme.

2 Requirements

R1	IPS shall provide uniform interface to the kernel and Notify module.
R2	IPS must provide event prioritization between events.
R3	IPS shall take less time for its own logic execution, so that callbacks can be called within very short time.
R4	IPS shall be configurable for providing a particular execution context to the callbacks. (for example, ISR, DPC, thread etc)
R5	Notify shall APIs same IPS have.
R6	Notify shall maintain the event prioritization policy.

3 Assumptions

A1 On Linux, POSIX threads are available.

4 Constraints

C1	Only 32 events will be available for each process.
C2	Events will have priority in order to their event no. that is event 0 will have highest priority.
C3	Event preemption is not allowed.

5 High Level Design

5.1 IPS

5.1.1 Overview

The Inter-processor signaling (IPS) component is responsible for notifying an event to its peer on the remote processor. This component shall use the services provided on the hardware platform. It shall provide APIs, which shall be used by upper layers to establish communication amongst peers at that level.

5.1.2 Services provided

The IPS component shall provide the following basic services:

1. Register an event.
2. Unregister an event.
3. Send a value to the remote processor.
4. Notify the remote processor about an event.
5. Event prioritization must be done.

5.1.3 Design

The IPS component consists of two major components:

1. IPS send
2. IPS Receive

IPS on either side will provide mechanism to send and receive events. IPS send component will provide the processes/tasks on either side a notification sending mechanism. IPS receive component will provide notification receiving mechanism.

Processes have to register their callbacks with the IPS receiving component inorder to receive an event. While for sending events no need of registration is needed. On receiving an event register request, IPS receive component will store the processes /tasks related information to be used at the time of callbacks. Multiple processes can wait for a particular event to occur i.e. can register for an event.

On receiving an event, IPS receive component calls the callback/callbacks associated for that event. In the event of calling or processing the callback if a higher priority event and a low priority event come, then IPS first completes the processing of the current event and after that it calls the callback associated with the highest priority event then the low priority event's callback. If the event is associated with many interested user processes, then their callbacks are called in the order of registrations. In this case which process will get the event notification first is unspecified (if they are all at the same priority of execution).

Inside a process, there is no possibility of having multiple callbacks for same event.

Below diagram shows the interaction between IPS and user processes.

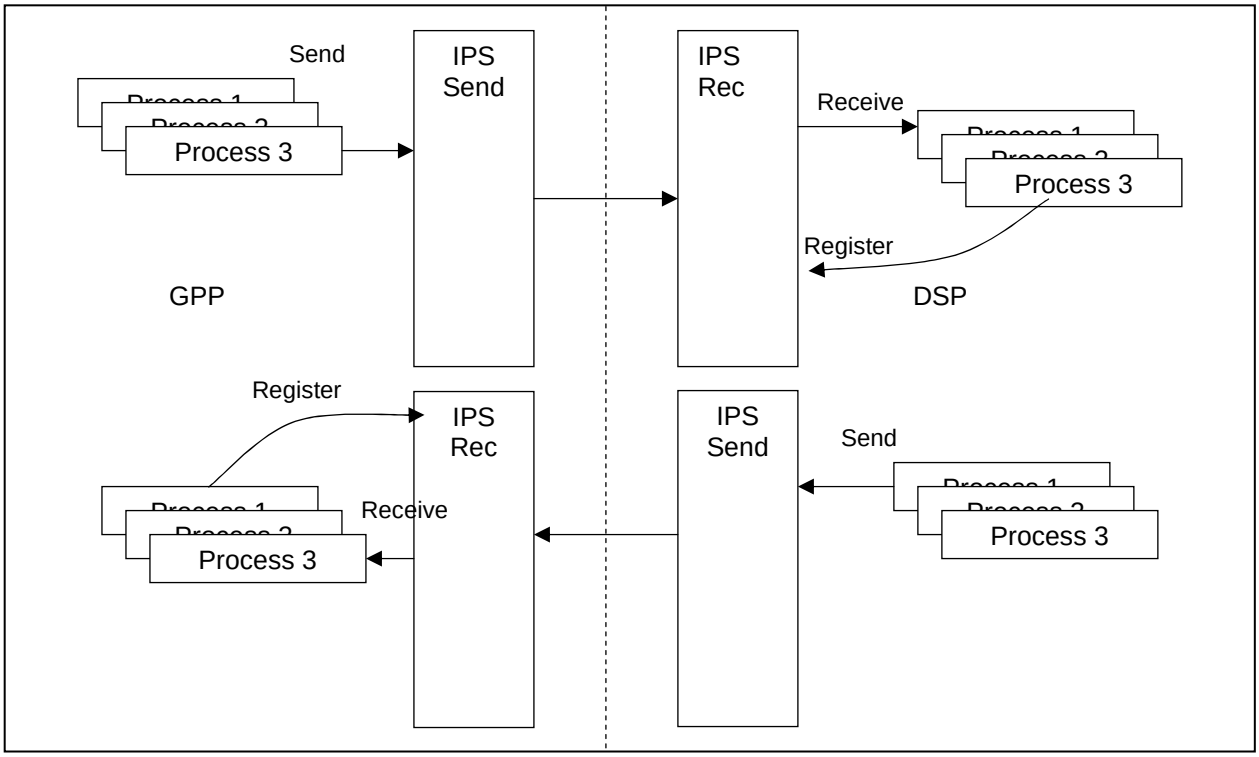

Figure 1. Block diagram for IPS component

IPS Send

User process calls this component to signal other side about an event. IPS Send component registers the request and try to send if it finds the hardware notification transport free. If the transport is not free it buffers the request and tries to send it later meanwhile user processes are in waiting state for event send to complete.

IPS Receive

This component exposes two basic methods:

1. Event register

   User processes calls this method for registering an event notification. IPS stores the user process related information.

2. Event Receive

   On the event of receiving an event, IPS retrieves the information associated with the event and calls its callback.

5.2 Notify

5.2.1 Overview

The Notify component is responsible for exposing the features of IPS to the user applications. In Linux, it also acts as a event dispatcher (Since user mode callbacks can not be called from kernel) to the user processes through thread and List. It can be seen as wrapper around IPS component.

5.2.2 Services provided

The Notify component shall provide the following basic services:

1. Register an event.
2. Unregister an event.
3. Send a value to the remote processor.
4. Notify the remote processor about an event.
5. Event prioritization.

5.2.3 Design

The Notify component consists of two major components:

3. Notify send
4. Notify Receive

Notify on either side will expose the IPS provided mechanism for sending and receiving events. Notify send component will expose the processes/tasks on either side the notification sending mechanism. Notify receive component will expose the notification receiving mechanism.

Processes/tasks will register their callbacks with the IPS through Notify register method. In Linux, user level callbacks are not allowed to be called from kernel.

Notify Component is divided into two parts (both are part of OSAL):

1) Kernel Side Component.

   When an user process wants to get notified for an event, the kernel side component registers a callback with IPS component. This callback is registered with information pertaining to the user process (such as, process group id, user level callback function address, user level param). When ever this callback is called by IPS component, it inserts the information passed by the IPS component into a linked-list.

2) User Side Component.

   In user side component a thread is created, whose job is to look for any buffers in the linked-list (inserts are done by kernel component). Since the linked-list is present in kernel only, thus the read system call of the DSPLINK driver module is modified for reading the linked-list. Since looking for buffers in the linked-list through system call is bulky operations in terms of time, thus the thread looks for a buffer for 50 times (each time if it does not finds any buffer it yields the control), after which it goes to sleep for 2ms.

   After it finds a buffer (which contains information about the listener) it calls the user level callback registered with the notify component.

Figure 2 shows the block diagram for Notify component.

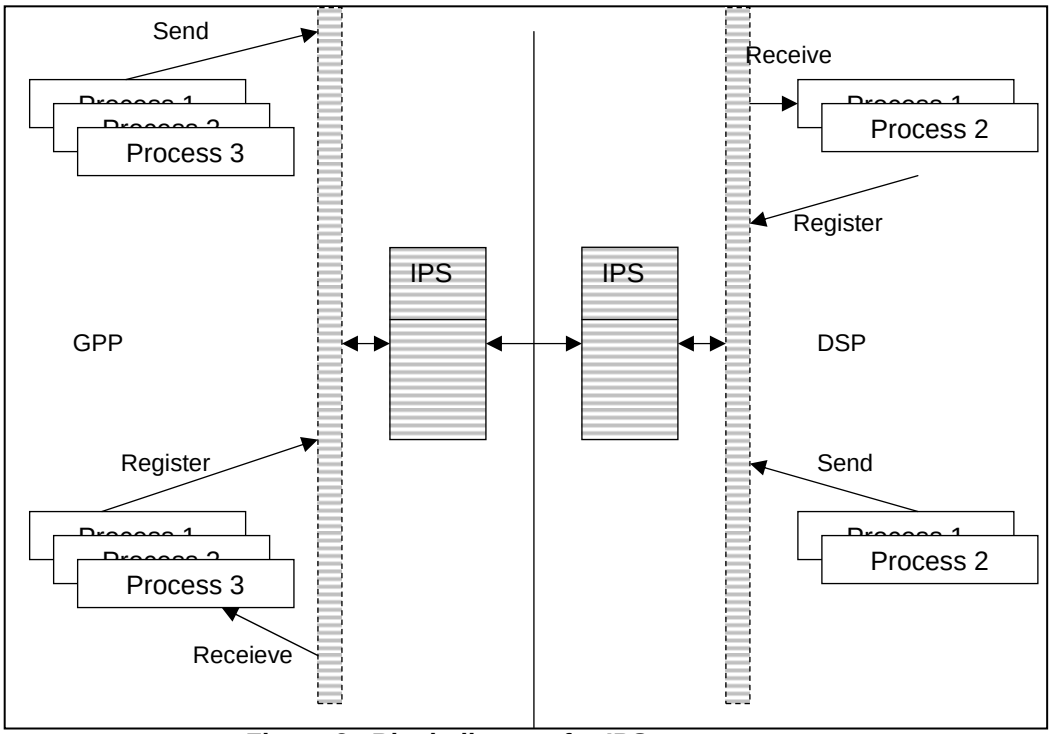

Figure 2. Block diagram for IPS component

6 Sequence Diagrams

The following sequence diagrams show the control flow for a few of the important functions to be implemented within the *DSPLINK* IPS component.

While the following sequence diagrams show the control flow for the GPP-side of *DSPLINK*, the control flow on the GPP-side is similar, and is not detailed in this document.

- *The dashed arrow in all sequence diagrams indicates an indirect control transfer, which does not happen through a direct function call.*

6.1 IPS_init

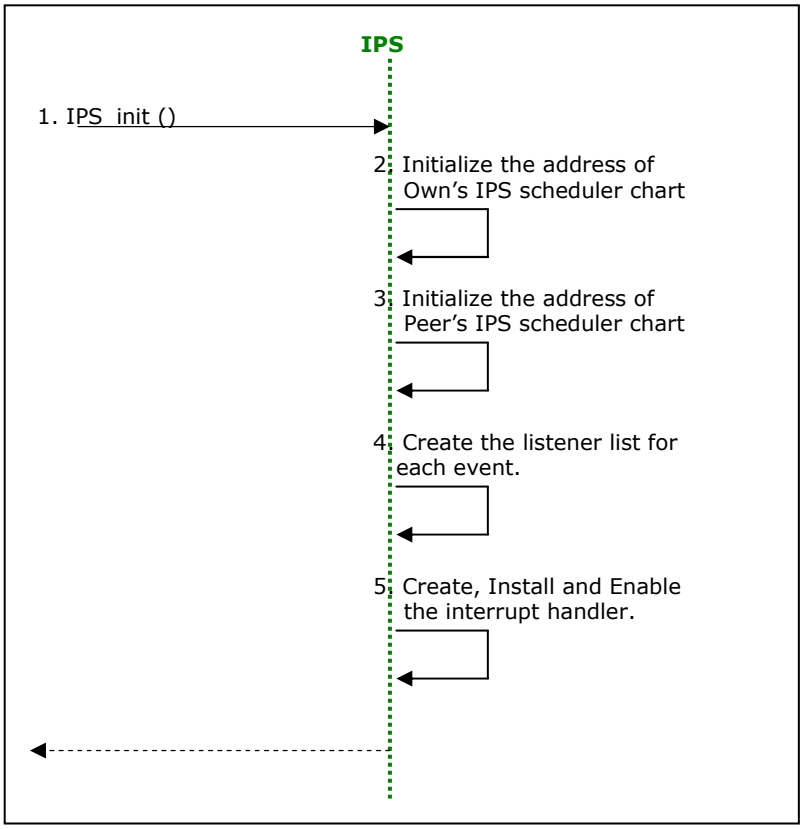

Figure 2. On the GPP: IPS_init () control flow

6.2 IPS_exit

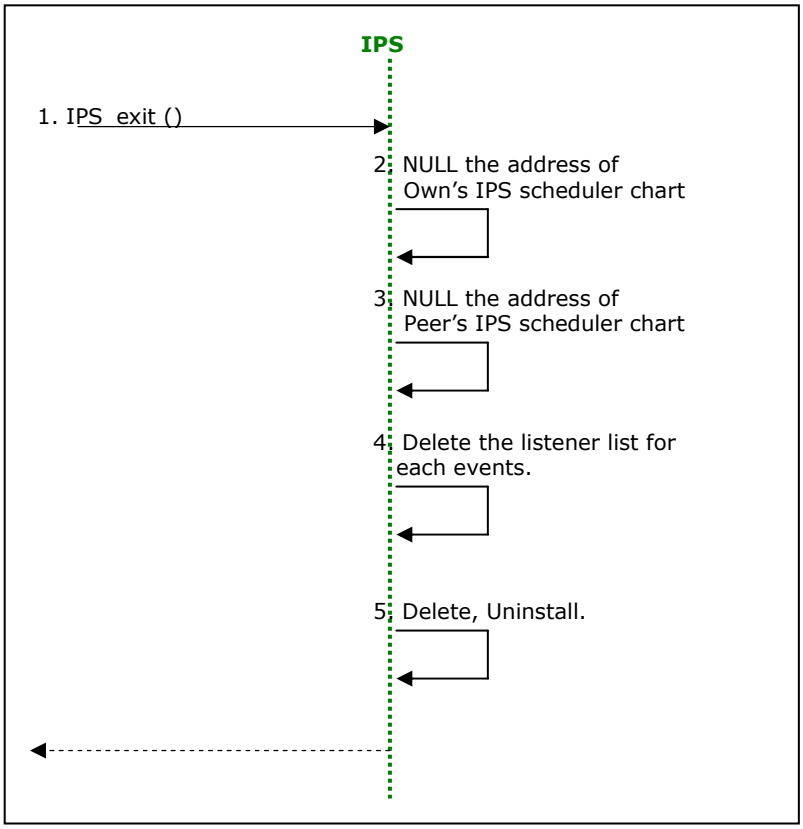

Figure 3. On the GPP: IPS_exit () control flow

6.3 IPS_register

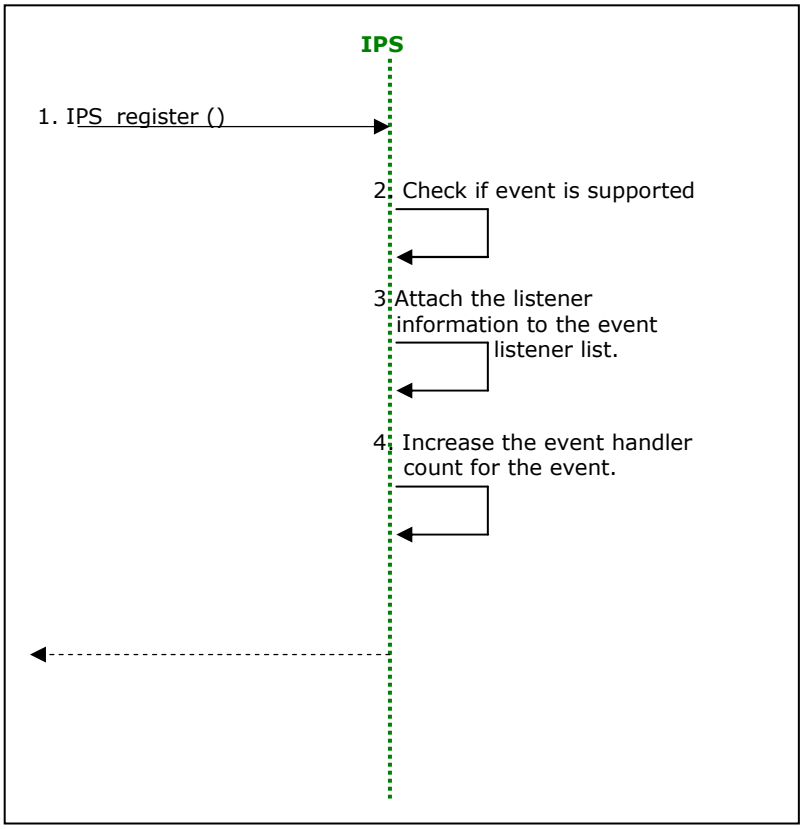

Figure 4. On the GPP: IPS_register () control flow

6.4 IPS_unregister

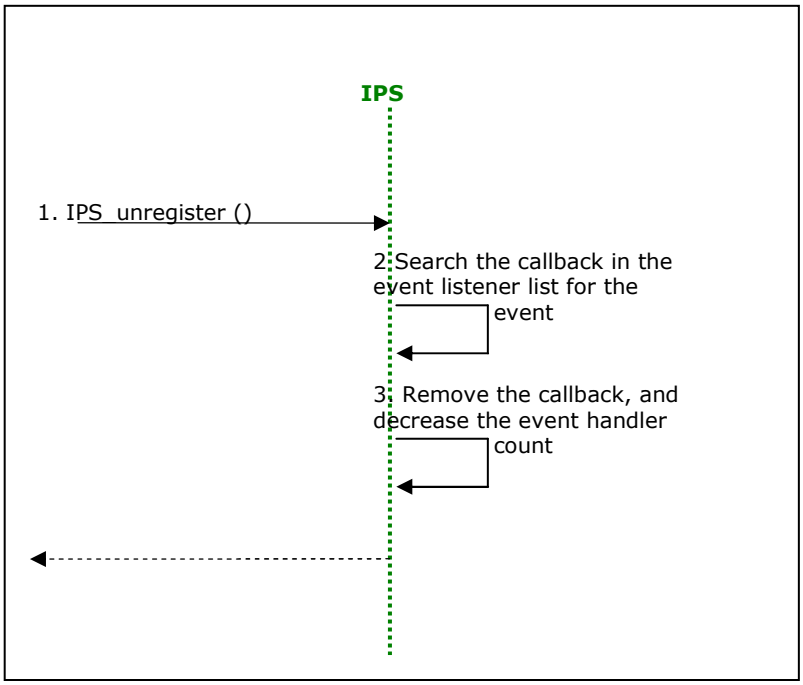

Figure 5. On the GPP: IPS_unregister () control flow

6.5 IPS_notify

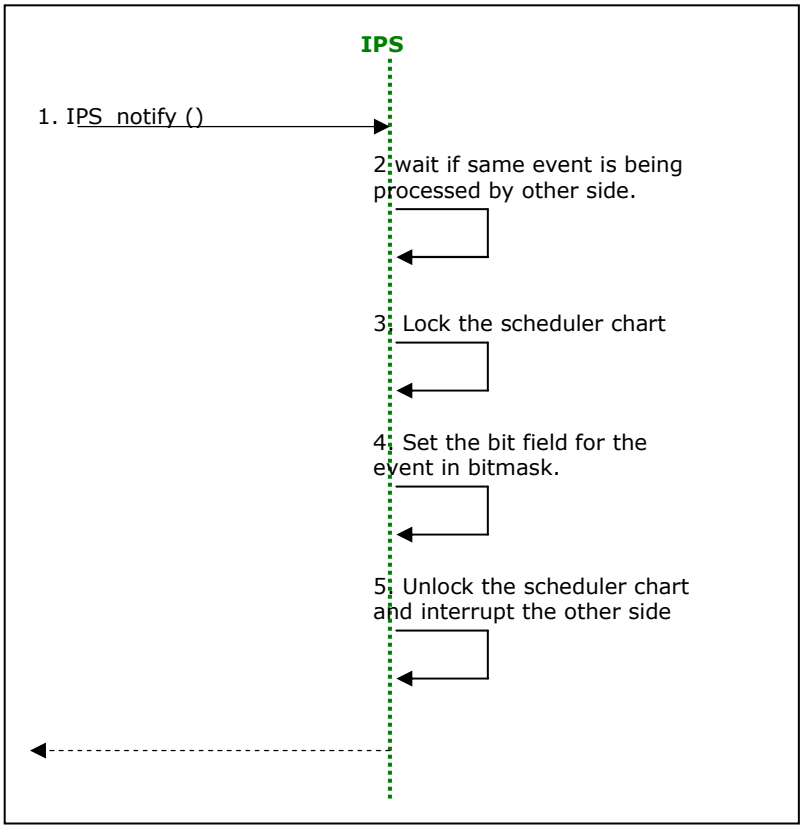

Figure 6. On the GPP: IPS_notify () control flow

6.6 IPS_ISR

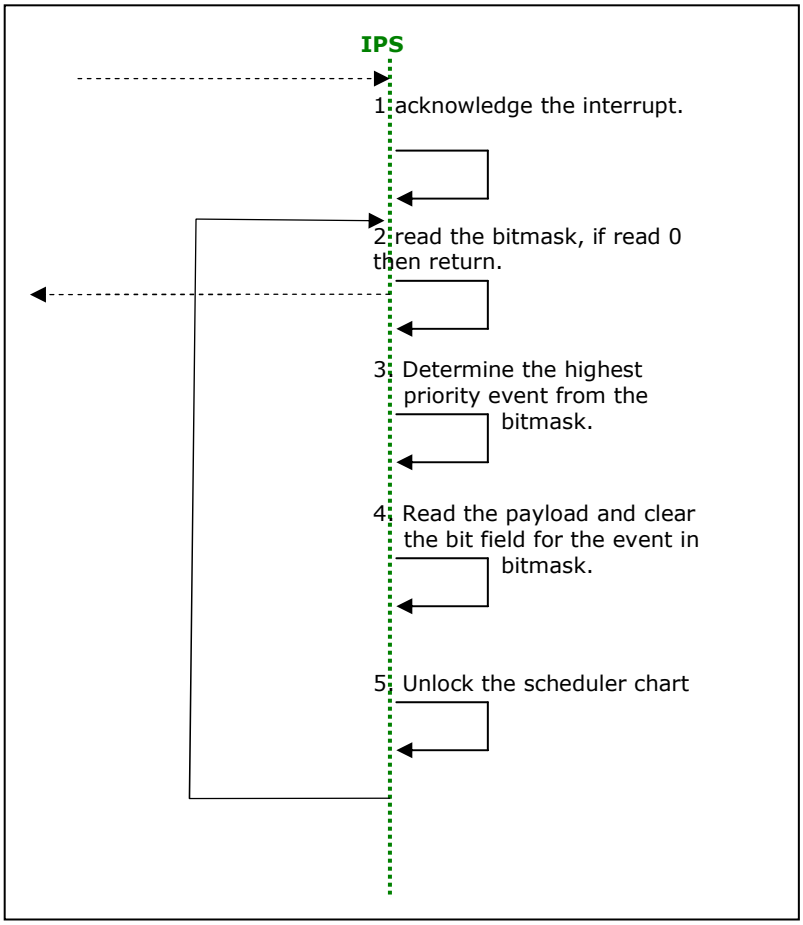

Figure 7. On the GPP: IPS_ISR () control flow

6.7 _NOTIFY_init

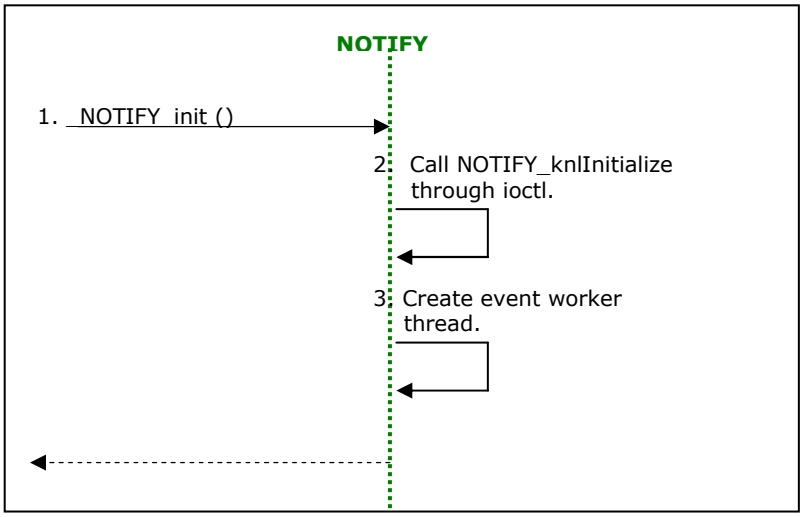

Figure 8. On the GPP: _NOTIFY_init () control flow

6.8 _NOTIFY_exit

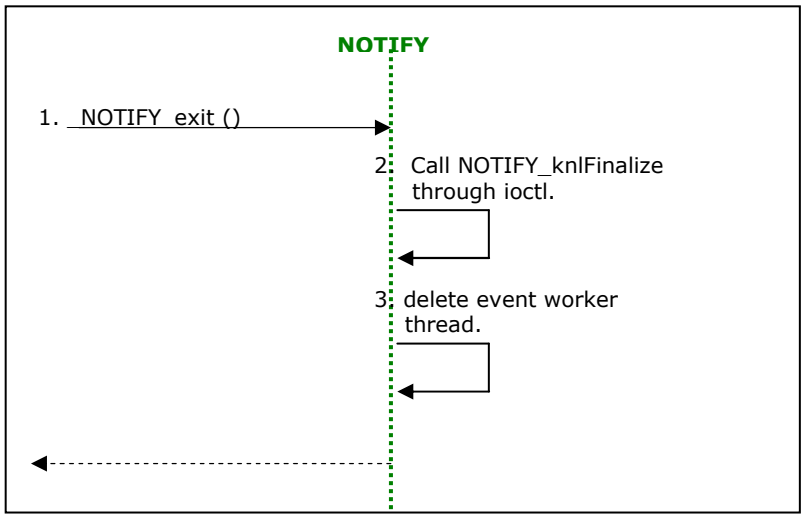

Figure 9. On the GPP: _NOTIFY_exit () control flow

6.9 NOTIFY_eventWorker

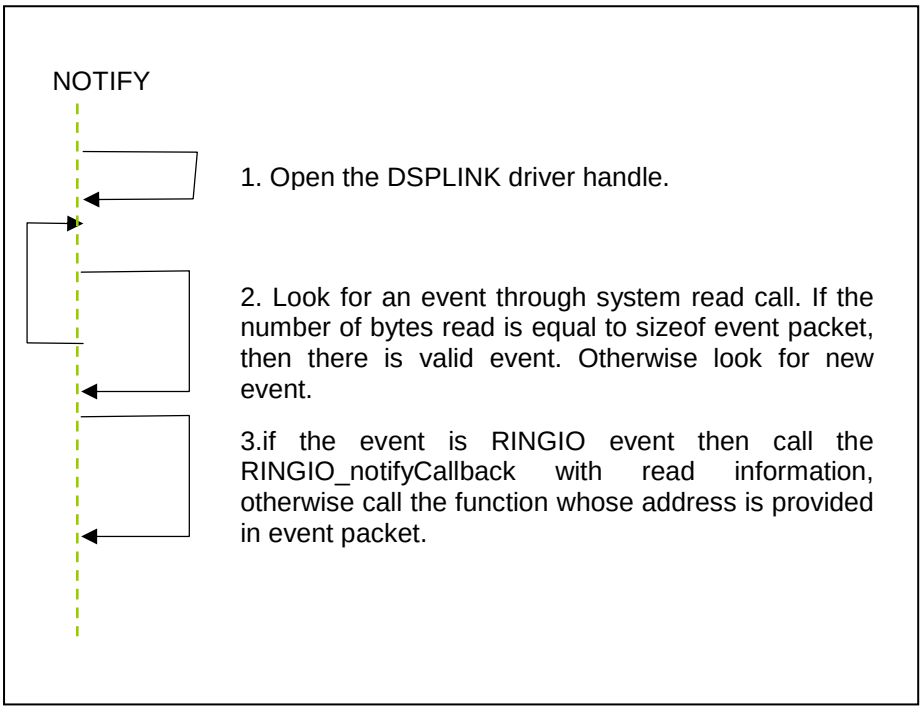

Figure 10. On the GPP: NOTIFY_eventWorker () control flow

6.10 NOTIFY_register

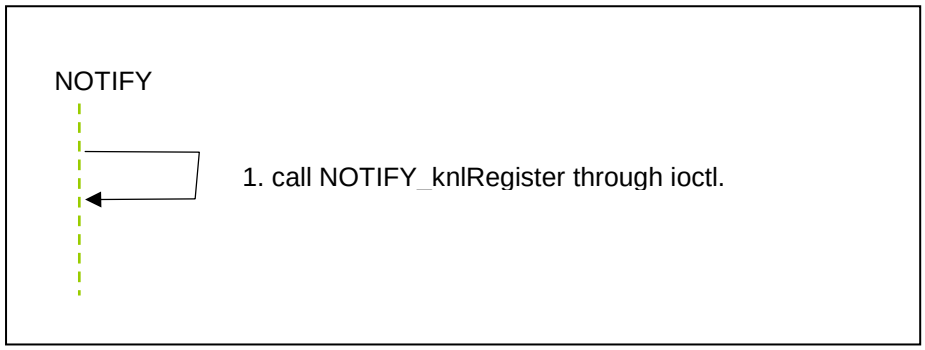

Figure 11. On the GPP: NOTIFY_register () control flow

6.11 NOTIFY_unregister

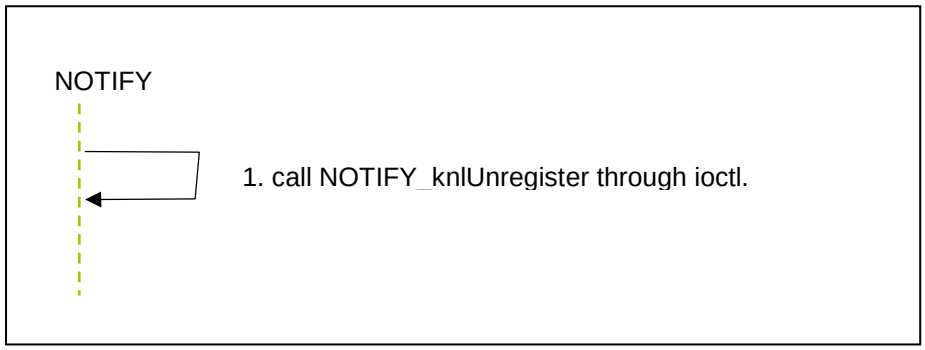

Figure 12. On the GPP: NOTIFY_unregister () control flow

7 Low Level Design

7.1 IPS

7.1.1 Constants & Enumerations

7.1.1.1 MAX_IPS_EVENT

This constant defines the maximum number of IPS events supported by the IPS module

Definition

```
#define MAX_IPS_EVENT 32
```

Comments

This constant can have maximum value of 32 for all platforms.

Constraints

None.

See Also

None.

7.1.2 Typedefs & Data Structures

7.1.2.1 IPS_EventListener

This structure defines the Event Listener object, which contains the listener-specific information.

Definition

```
typedef struct IPS_EventListener_tag {
    ListElement  * element   ;
    FnIpsCbck      fnIpsCbck ;
    Pvoid          cbckArg   ;
} IPS_EventListener ;
```

Fields

`element`	Structure that allows it to be used by LIST.
`fnIpsCbck`	Callback functions for the event.
`cbckArg`	Parameters passed to the callback.

Comments

An instance of this object is created and initialized during `IPS_register()`. It contains all information required for notify the correct listener about the event.

Constraints

None.

See Also

`IPS_register ()`

7.1.2.2 IPS_Event

This structure defines the Event Listener object, which contains information about all listener registered currently.

Definition

```
typedef struct IPS_EventList_tag {
    Uint32      eventHandlerCount ;
    List      * listeners   ;
} IPS_EventList ;
```

Fields

`eventHandlerCount`	Number of listener attached to the event.
`listeners`	Pointer to the first event listener.
`cbckArg`	Parameters passed to the callback.

Comments

An array of this object is created (using MAX_IPS_EVENTS). This holds information about all event listeners registered in the system.

Constraints

None.

See Also

`IPS_register ()`

7.1.2.3 IPS_SchedChart

This structure defines the scheduler Chart, which contains the occurred event-specific information. This is shared between GPP and DSP.

Definition

```
typedef struct IPS_SchedChart_tag {
    Uint32  bitMask ;
    Uint16  payloadArr [MAX_IPS_EVENT] ;
    Uint16  padding [IPS_EVENTCHART_PADDING] ;
} IPS_SchedChart ;
```

Fields

`bitMask`	Bit mask representing current occurred events.
`payloadArr`	Array containing data associated with each occurred event.
`padding`	Padding.

Comments

IPS_EVENTCHART_PADDING is depended upon which CACHE segment is used. Since bitmask is 32 bit wide thus maximum 32 events can be handled by IPS.

Constraints

None.

See Also

`IPS_notify (), IPS_ISR ()`

7.1.2.4 IPS_EventChart

This structure defines the Event object, which contains the occurred event-specific information and lock for the chart. This is shared between GPP and DSP.

Definition

```
typedef struct IPS_EventChart_tag {
    MPCSObj        ipsSchedChartLock ;
    IPS_SchedChart ipsSchedChart ;
} IPS_EventChart ;
```

Fields

`ipsSchedChartLock`	MPCS object used for locking the event chart.
`ipsSchedChart`	Scheduler chart.

Comments

None.

Constraints

None.

See Also

`IPS_notify (), IPS_ISR ()`

7.1.2.5 IPS_Object

This structure defines the IPS objects.

Definition

```
typedef IPS_Object_tag {
    IPS_EventList    ipsEventList [MAX_IPS_EVENTS] ;
    IPS_EventChart * ipsEventChart ;
    IsrObject *      isrObject ;
    ProcessorId      ipsDspId  ;
} IPS_Object ;
```

Fields

`ipsEventList`	Array containing information about registered listeners and events
`ipsEventChart`	Shared IPS event chart (Between GPP and DSP). this contains shared lock and occurred event chart.
`isrObject`	IPS ISR object.
`ipsDspId`	DSP Processor ID.

Comments

This is local to each processor. Here it maintains information about every IPS it has opened.

Constraints

None.

See Also

`IPS_init (), IPS_exit (),IPS_register (), IPS_unregister ()`

7.1.2.6 FnIpsCbck

This typedef defines Signature of the callback function to be registered with the IPS component.

Definition

```
typedef Void (*FnIpsCbck) (IN OPT Pvoid arg, IN OPT Pvoid info) ;
```

Fields

`arg`	Fixed argument registered with the IPS component along with the callback function.
`info`	Run-time information provided to the upper layer by the IPS component. This information is specific to the IPS being implemented

Comments

None.

Constraints

None.

See Also

None.

7.1.3 API Definition

7.1.3.1 IPS_init

This function initializes the IPS component.

Syntax

```
DSP_STATUS
IPS_init (IN ProcessorId dspId) ;
```

Arguments

IN	ProcessorId	dspId

DSP Identifier.

Return Value

DSP_SOK	The IPS for dspId has been successfully opened.
DSP_EFAIL	General failure

Comments

None.

Constraints

dspId must be valid.

See Also

IPS_exit ()

7.1.3.2 IPS_exit

This function finalizes the IPS component.

Syntax

```
DSP_STATUS
IPS_exit (IN ProcessorId dspId) ;
```

Arguments

IN	ProcessorId	dspId

DSP Identifier.

Return Value

DSP_SOK	The IPS for dspId has been successfully closed.
DSP_EFAIL	General failure

Comments

None.

Constraints

dspId must be valid.

See Also

IPS_init ()

7.1.3.3 IPS_register

This function registers a callback for a specific event with the IPS component.

Syntax

```
DSP_STATUS
IPS_register (IN     ProcessorId dspId,
              IN     Uint32      eventNo,
              IN     FnIpsCbck   fnIpsCbck,
              IN OPT Pvoid       cbckArg) ;
```

Arguments

`IN` `ProcessorId` `dspId`

DSP Identifier.

`IN` `Uint32` `eventNo`

Event No to be registered.

`IN` `FnIpsCbck` `fnIpsCbck`

Callback function to be registered for the specified event.

`IN OPT` `Pvoid` `cbckArg`

Optional argument to the callback function to be registered for the specified event. This argument shall be passed to each invocation of the callback function.

Return Value

`DSP_SOK`	Operation successfully completed
`DSP_EFAIL`	General failure
`DSP_EWRONGSTATE`	IPS not initialized

Comments

None.

Constraints

The IPS component must be initialized before calling this function.

The fnIpsCbck argument must be valid.

The event must be supported by the IPS component.

See Also

`IPS_unregister ()`

7.1.3.4 IPS_unregister

This function unregisters the callback for the specific event with the IPS component.

Syntax

```
DSP_STATUS
IPS_unregister (IN     ProcessorId dspId,
                IN     Uint32      eventNo,
                IN     FnIpsCbck   fnIpsCbck) ;
```

Arguments

`IN` `ProcessorId` `dspId`

DSP Identifier.

`IN` `Uint32` `eventNo`

Event No to be used.

`IN` `FnIpsCbck` `fnIpsCbck`

Callback function to be unregistered for the specified event.

Return Value

`DSP_SOK`	Operation successfully completed
`DSP_EFAIL`	General failure
`DSP_EWRONGSTATE`	IPS not initialized

Comments

None.

Constraints

The IPS component must be initialized before calling this function.

The fnIpsCbck argument must be valid.

The event must be supported by the IPS component.

The event must have been registered with the IPS component earlier.

See Also

`IPS_register ()`

7.1.3.5 IPS_notify

This function registers a callback for a specific event with the IPS component.

Syntax

```
DSP_STATUS
IPS_notify (IN ProcessorId dspId,
            IN Uint32      eventno,
            IN Uint16      payload);
```

Arguments

IN	ProcessorId	dspId
	DSP Identifier to notified.	
IN	Uint32	eventNo
	Event No to be used.	
IN	Uint16	payload
	Data to be sent with Event.	

Return Value

DSP_SOK	Operation successfully completed
DSP_EFAIL	General failure
DSP_EWRONGSTATE	IPS not initialized

Comments

None.

Constraints

The IPS component must be initialized before calling this function.

The cbckFxn argument must be valid.

The event must be supported by the IPS component.

See Also

IPS_unregister ()

7.2 Notify

7.2.1 Typedefs & Data Structures

7.2.1.1 FnNotifyCbck

This typedef defines Signature of the callback function to be registered with the NOTIFY component.

Definition

```
typedef Void (*FnNotifyCbck) (IN Uint32 eventNo, IN OPT Pvoid arg, IN
OPT Pvoid info) ;
```

Fields

`eventNo`	Event number associated with the callback being invoked
`arg`	Fixed argument registered with the IPS component along with the callback function.
`info`	Run-time information provided to the upper layer by the notify component. This information is specific to the IPS being implemented

Comments

None.

Constraints

None.

See Also

`None.`

7.2.2 API Definition

7.2.2.1 _NOTIFY_init

This function initializes the notify component.

Syntax

```
DSP_STATUS
_NOTIFY_init (IN ProcessorId dspId) ;
```

Arguments

IN	ProcessorId	dspId

DSP Identifier.

Return Value

DSP_SOK	The notify component has been successfully opened.
DSP_EFAIL	General failure

Comments

PROC_attach () internally calls this function. Applications need not call this API.

Constraints

`dspId` must be valid.

See Also

`_NOTIFY_exit ()`

7.2.2.2 _NOTIFY_exit

This function finalizes the notify component.

Syntax

```
DSP_STATUS
_NOTIFY_exit (IN ProcessorId dspId) ;
```

Arguments

IN	ProcessorId	dspId

DSP Identifier.

Return Value

DSP_SOK	The notify component has been successfully closed.
DSP_EFAIL	General failure

Comments

PROC_detach () internally calls this function. So applications need not call this function.

Constraints

`dspId` must be valid.

See Also

`_NOTIFY_init ()`

7.2.2.3 NOTIFY_register

This function registers a callback for a specific event with the notify component.

Syntax

```
DSP_STATUS
NOTIFY_register (IN     ProcessorId dspId,
                 IN     Uint32      ipsId,
                 IN     Uint32      eventNo,
                 IN     FnIpsCbck   fnIpsCbck,
                 IN OPT Pvoid       cbckArg) ;
```

Arguments

IN	ProcessorId	dspId

DSP Identifier.

IN	Uint32	ipsId

IPS identifier.

IN	Uint32	eventNo

Event No to be registered.

IN	FnIpsCbck	fnIpsCbck

Callback function to be registered for the specified event.

IN OPT	Pvoid	cbckArg

Optional argument to the callback function to be registered for the specified event. This argument shall be passed to each invocation of the callback function.

Return Value

DSP_SOK	Operation successfully completed
DSP_EFAIL	General failure
DSP_EWRONGSTATE	IPS not initialized
DSP_EINVALIDARG	Invalid arguments.

Comments

None.

Constraints

The notify component must be initialized before calling this function.

PROC_attach has been successful before calling this function.

The fnIpsCbck and dspId argument must be valid.

The event must be supported by the IPS component.

See Also

`NOTIFY_unregister ()`

7.2.2.4 NOTIFY_unregister

This function unregisters the callback for the specific event with the IPS component.

Syntax

```
DSP_STATUS
NOTIFY_unregister (IN     ProcessorId    dspId,
                   IN     Uint32         ipsId,
                   IN     Uint32         eventNo,
                   IN     FnNotifyCbck   fnNotifyCbck,
                   IN OPT Pvoid          cbckArg) ;
```

Arguments

`IN` `ProcessorId` `dspId`

DSP Identifier.

`IN` `Uint32` `ipsId`

IPS identifier.

`IN` `Uint32` `eventNo`

Event No to be registered.

`IN` `FnIpsCbck` `fnIpsCbck`

Callback function to be registered for the specified event.

`IN OPT` `Pvoid` `cbckArg`

Optional argument to the callback function to be registered for the specified event. This argument shall be passed to each invocation of the callback function.

Return Value

`DSP_SOK`	Operation successfully completed
`DSP_EFAIL`	General failure
`DSP_EWRONGSTATE`	IPS not initialized
`DSP_EINVALIDARG`	Invalid arguments

Comments

None.

Constraints

The IPS component must be initialized before calling this function.

The fnIpsCbck argument must be valid.

The event must be supported by the IPS component.

The event must have been registered with the IPS component earlier.

The event must be supported by the NOTIFY component.

All Notifications are complete.

See Also

`NOTIFY_register ()`

7.2.2.5 NOTIFY_notify

This function registers a callback for a specific event with the IPS component.

Syntax

```
DSP_STATUS
NOTIFY_notify (IN     ProcessorId dspId,
               IN     Uint32      ipsId,
               IN     Uint32      eventno,
               IN OPT Uint32      payload);
```

Arguments

`IN` `ProcessorId` `dspId`

Processor id to which notification needs to be sent.

`IN` `Uint32` `ipsId`

IPS identifier

`IN` `Uint32` `eventNo`

Event No to be used.

`IN` `Uint16` `payload`

Data to be sent with Event.

Return Value

`DSP_SOK`	Operation successfully completed
`DSP_EFAIL`	General failure
`DSP_EWRONGSTATE`	IPS not initialized
`DSP_EINVALIDARG`	Invalid arguments.

Comments

To get the notifications, application on the dspId must register a call back function with the notify component using ipsId and event no prior to this NOTIFY_notify call.

This API notifies the call back function if the function is registered on the same ipsId and eventno.

Constraints

The IPS component must be initialized before calling this function.

The cbckFxn argument must be valid.

The event must be supported by the IPS component.

See Also

`NOTIFY_register ()`

8 Internal Discussions

8.1 Prioritizing IPS (High level design)

Registering the Events:

The IPS component will provide mechanism to register the user defined event callbacks. Users will register events with user defined priority (where 0 is the highest priority, other follow)

```
typedef struct EventInfo_tag {
      Pvoid  entryFunction ;
      Uint32 params        ;
} EventInfo ;
```

And event is registered in the list below, using event number to index into the list.
`EventInfo eventList [MAX_IPS_EVENT] ;`
So, users can not register two callback functions for single event but events can have same user defined priorities.

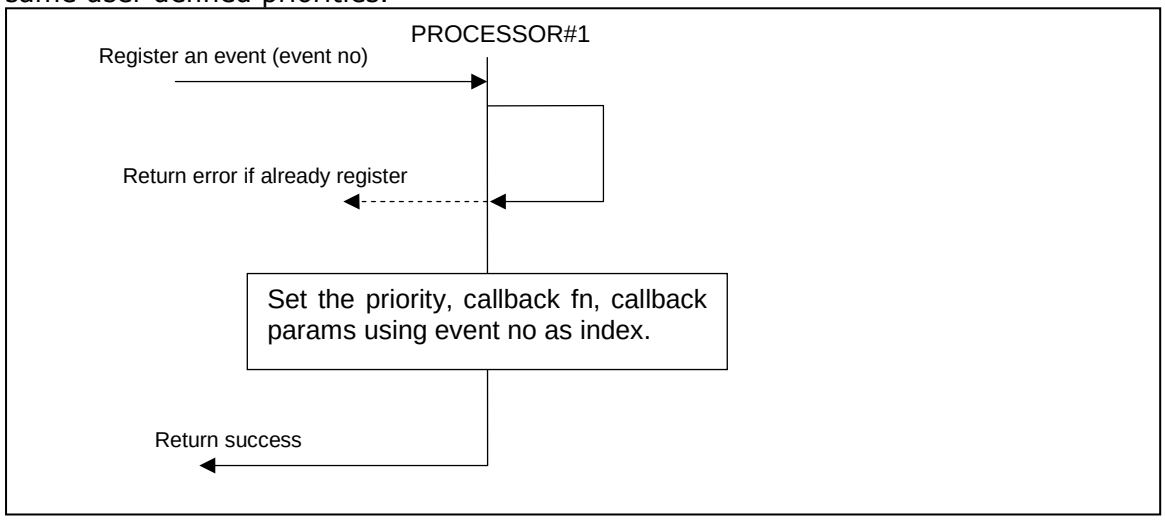

Figure 13. Register an Event

IPS Scheduler

Need:

As we are well aware of the fact that DSP/BIOS™ LINK uses only one interrupt for notification of MSGQ/CHNL events. Using only one interrupt limits DSP/BIOS™ LINK to have events prioritization scheme. The other need is, it does not provide feature to service other kind of events (For say, a user defined event).

Logic:

Since it has only one interrupt for notification, it must have a priority scheme other than interrupts priority. This can be achieved by having an event scheduler run after receiving an interrupt. Job of this scheduler will be to service the highest priority event first then rest (Assuming that each event is assigned a priority, from a defined scale for priorities). One constraint for getting the multiple events is the interrupt handler (for the only one interrupt) must be very fast (i.e. low latency time), so that other side can send more interrupts in short durations. Now there are two approaches for servicing events:

Alternative1:

This scheduler is based on an event chart. An event chart is basically a data structure containing an event list to store the occurred events and storage space for the payload attached with each event. Instead of the event list, a bitmask can be used to reduce the memory requirement then it might handle less number of events and might also complex the logic. This scheduler will be called from the ISR directly. It is an important fact that, the event list (Linklist) management takes more time ultimately increasing the interrupt latency time. So a bitmask solution is a better approach. IPS scheduler will handle only 32 events and 16Bits payload attached with each event. So we require 4Bytes + (4 *16) Bytes = 68Bytes. See Figure 1.

Here for simplicity each event has same priority number as the event No i.e. event 0 has priority 0 (Highest) and so on.

The senders of the events have to lock the event chart, update the bitmask to reflect presence of the event, attached the payload and interrupts the other side. On the other side in ISR, scheduler is called. First it locks the event chart, searches for the highest priority event. Once it finds the highest priority event it clears the events in the bitmask only after reading the payload, then calls the callback attached to the event.

As it is clear that prioritization has to be done between different events (mean different priorities) so if multiple senders want to send same event to the other side than event are send one at a time meaning second event is send only if the fist event is taken up by the scheduler on the side..

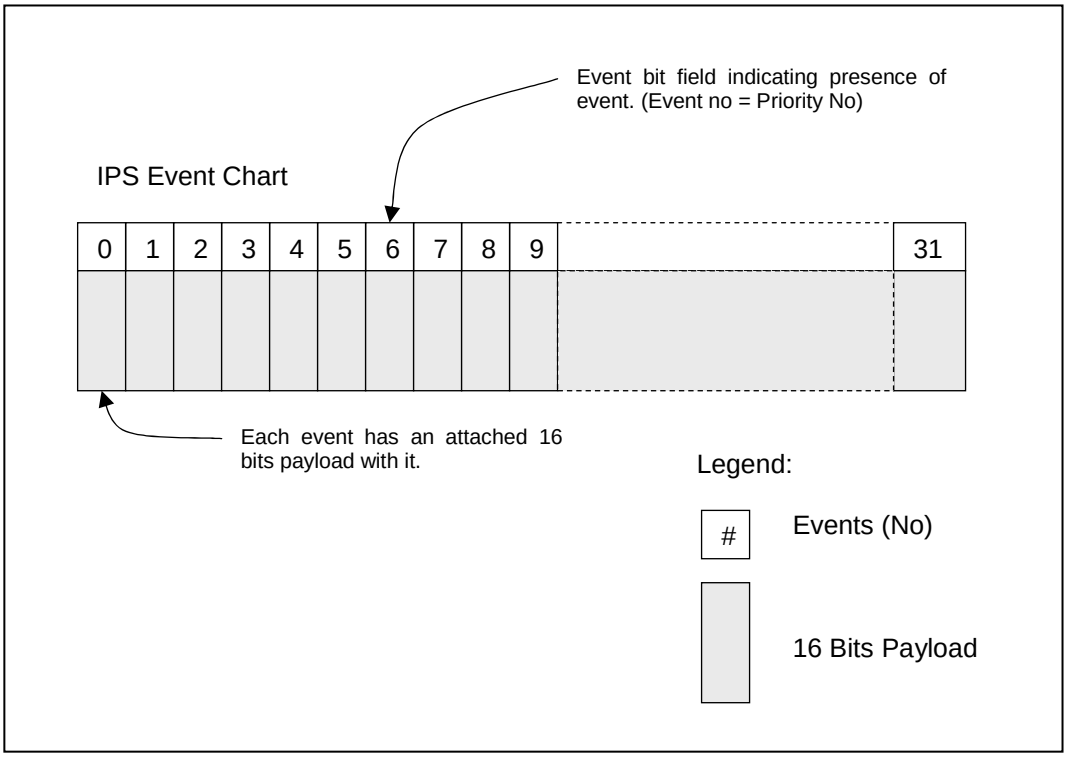

Figure 14. Event chart

The below figures shows the sequence diagram:

In Figure 2, senders on the processor 1 waits if the bit field in the event chart is already set to one, meaning the scheduler on the processor 2 is yet to service this event. This is true since priority stands good in always event different events are handle simultaneously.

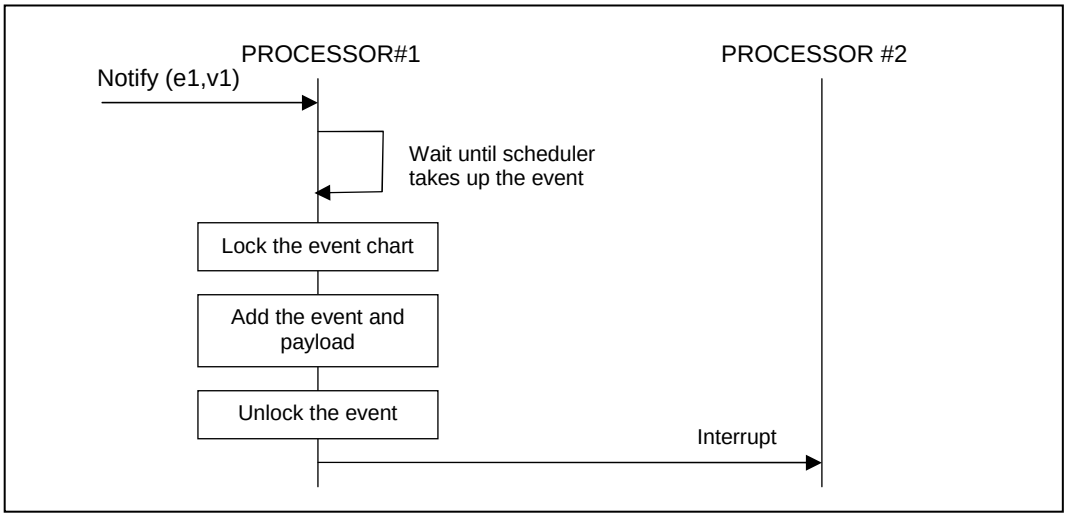

Figure 15. Sending an Event

In Figure 3, receiver of the interrupt invokes an ISR to handle the interrupt, whose main job is to call scheduler logic. Scheduler acquires the event chart lock and then checks the bitmask for finding out which event is having highest priority, then clearing it and servicing the event.

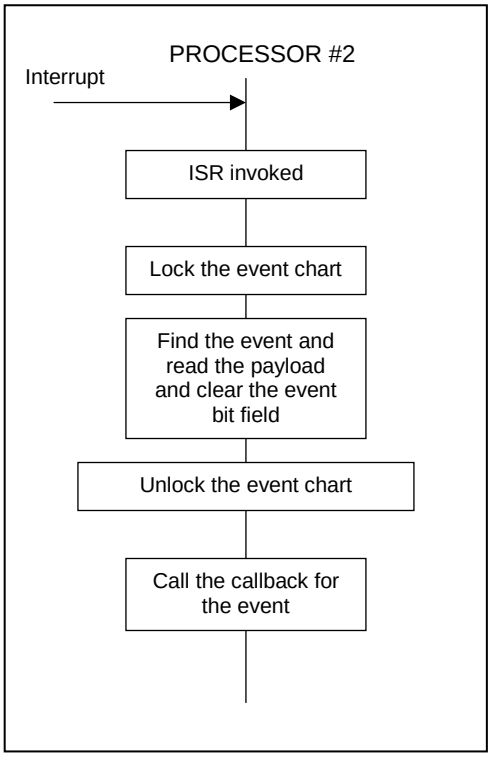

Figure 16. Receiving an Event

Advantages:

1. Does not depend upon any OS resources like DPC.
2. Provides an ISR context to every callback.

Disadvantages:

1. Event preemption can not be achieved.
2. ISR consumes times in IPS; Since ISR calls the scheduler logic.

Alternative 2:

In this scheduler all senders do not have to wait in case of same event is being process by the other side. Sender interrupts the other side with event and payload. The other side uses event no passed to determine the priority of the event from the event list. Then after determining the priority it creates a DPC according to priority (some scaling has to be done). So the OS really does the actual scheduling part of the IPS.

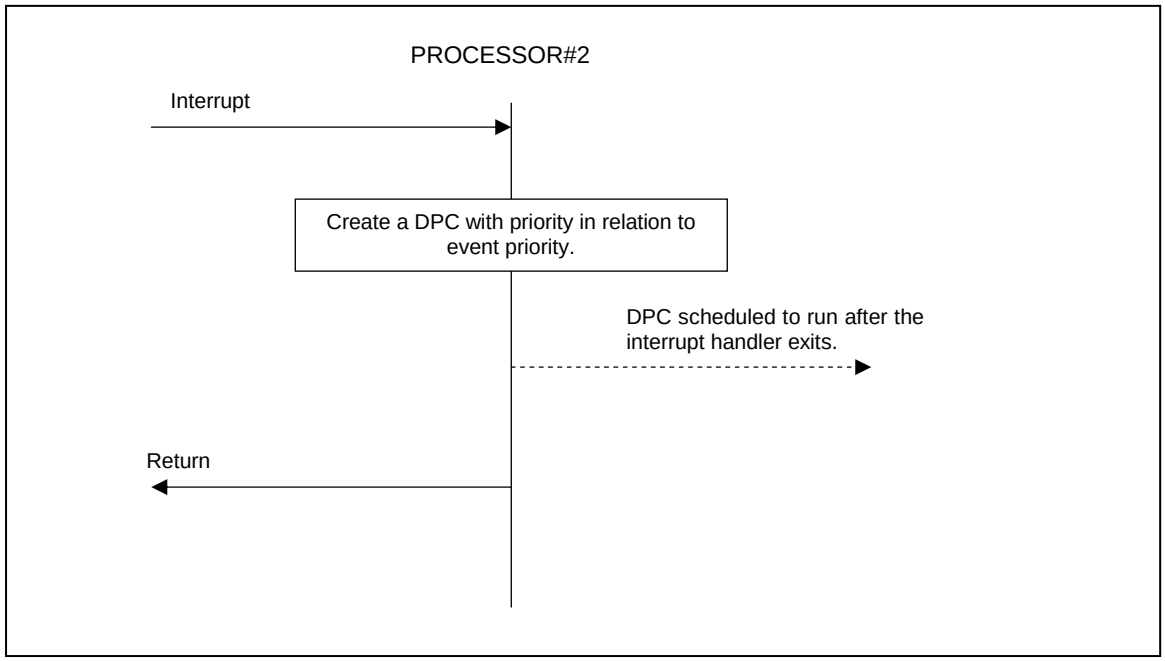

Figure 17. Sending an Event

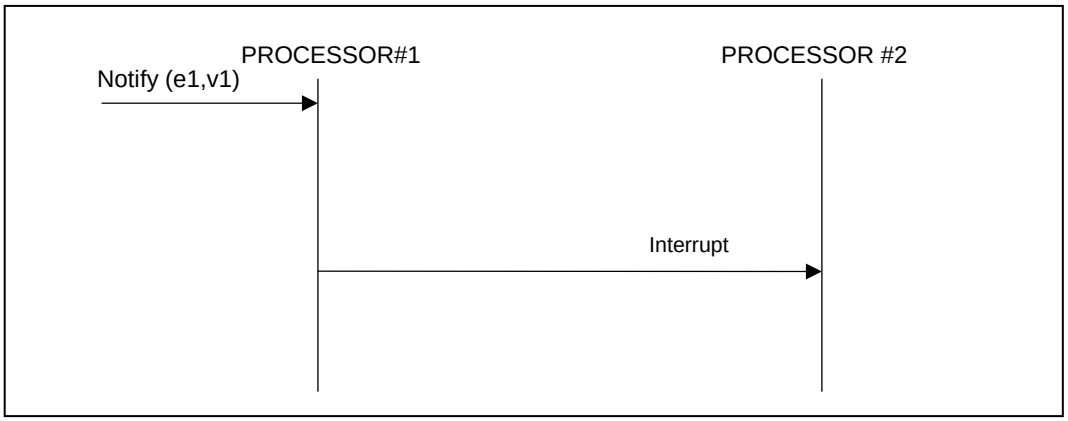

Figure 18. Receiving an Event

Disadvantages:

1. Does require OS resources like DPC, threads.
2. Does not provide an ISR context to the callbacks.

Advantages:

1. Event preemption can be achieved.
2. ISR consumes very little times in IPS; Since ISR does not call heavy scheduler logic.

Chosen Alternative:

Since, IPS should provide ISR context to every callbacks and also should be less depended upon OS. We will be concentrating on the first alternative as a solution.

8.2 Prioritizing IPS (Low level discussion)

8.2.1 Register

For registering events the following prototype and structure is needed.

Prototype for callbacks:

```
Void (*IPS_Callback) (Void * params) ;
```

Structure for storing registered events:

```
typedef struct EventList_tag {
      IPS_CallBack callback ;
      Pvoid        params   ;
} EventList ;
```

Registered event informaton list:

```
#define MAX_IPS_EVENT 32 /* Always */
EventList eventList [MAX_IPS_EVENTS] ;
```

To register an event:
1. Check if the event number is less than maximum number of event supported.
2. Check if the event is already register or not.
3. if event is not registered, then register the event.

```
IPS_Register (Uint32 eventNo,
              Uint32 priority,
              Pvoid  callback,
              Pvoid  params)
{
      /* check if the eventNo is not beyond the MAX_IPS_EVENTS. */
      if (eventNo < MAX_IPS_EVENTS) {
            /* check if the eventNo is already registered. */
            if (eventList [eventNo].priority == -1) {
                  eventList [eventNo].callback = callback ;
                  eventList [eventNo].params   = params   ;
            }
            else {
                  /* Return already registered error. */
            }
      }
      else {
            /* Return INVALID event No reached error. */
      }
```

```
}
```

8.2.2 Notify & Send

For passing the event no and payload the following structures are needed:

```
typedef struct EventChart_tag {
      Uint32 bitMask ; /* Event bit mask */
      Uint16 payloadArr [MAX_IPS_EVENT] ; /* Payload array */
      Uint16 padding [] ; /* padding for cache line alignment */
} EventChart ;

typedef struct IpsEventMap_tag {
      EventChart eventChart ; /* Event chart */
      MPCSObj    mpcsObj    ; /* lock for the chart. */
} IpsEventMap ;
```

To send the event:

1. Check if the event is registered with the IPS.
2. If registered, check if the bitfield corresponding to the event in the bitmask is set, if set then wait until it is cleared.
3. Lock the eventChart using the mpcsObj.
4. Set the bitfield corresponding to the event in the bitmask.
5. Attached the payload.
6. Send the interrupt.

```
IPS_Notify (Uint32 eventNo, Uint16 payload)
{
      /* check if the eventNo is not beyond the MAX_IPS_EVENTS. */
      if (eventNo < MAX_IPS_EVENTS) {
            /* wait until the previous event with same event no is
               serviced. */
            while (ipsEventMap.eventChart & (1 << eventNo) == 1) ;

            MPCS_Enter (ipsEventMap.mpcsObj, TRUE) ;
            SET_BIT (ipsEventMap.eventChart.bitmask, eventNo) ;
            ipsEventMap.ev    entChart.payloadArr [eventNo] = payload ;
            MPCS_Leave (ipsEventMap.mpcsObj, TRUE) ;
            Send_Interrupt () ;
      }
      else {
            /* Return INVALID event No reached error. */
      }
}
```

IPS_ISR:

2. Acknowledge the interrupt.
3. Read the eventChart bitmask to find out which events are set. From those find out which event has highest priority.
4. Read the payload attached with that event.
5. Lock the event chart.
6. Clear the event bit field in bitmask.
7. Unlock the event chart.
8. Call the callback function attached with this event.

```
IPS_ISR (Uint32 eventNo, Uint16 payload)
{
      /* Acknowledge the interrupt. */
      Clear_Interrupt () ;
      while (ipsEventMap.eventChart.bitMask > 0) {
            eventMask = ipsEventMap.eventChart.bitMask;
```

```
                for (i=0; i<32; i++) {
                      if ((eventMask & 1 << i) == 1) {
                            break ;
                      }
                }

                MPCS_Enter (ipsEventMap.mpcsObj, TRUE) ;
                payload = ipsEventMap.eventChart.payLoad [i] ;
                CLEAR_BIT (ipsEventMap.eventChart.bitmask, i) ;
                MPCS_Leave (ipsEventMap.mpcsObj, TRUE) ;
                eventList [i].callback () ;
        }
    }
```

8.3 Notify

The main job of this module is to provide the application of DSP/BIOS™ LINK, a mechanism to register their callbacks for the event notification and get notification about events. As it is known that, all events are generated through IPS module. So Notify module provides a mechanism to register applications' callbacks into the IPS module for event notification. Since IPS module does not expose any APIs for the above said feature, thus Notify module expose the functionality of IPS register and event notification to the applications.

Where does it sit?

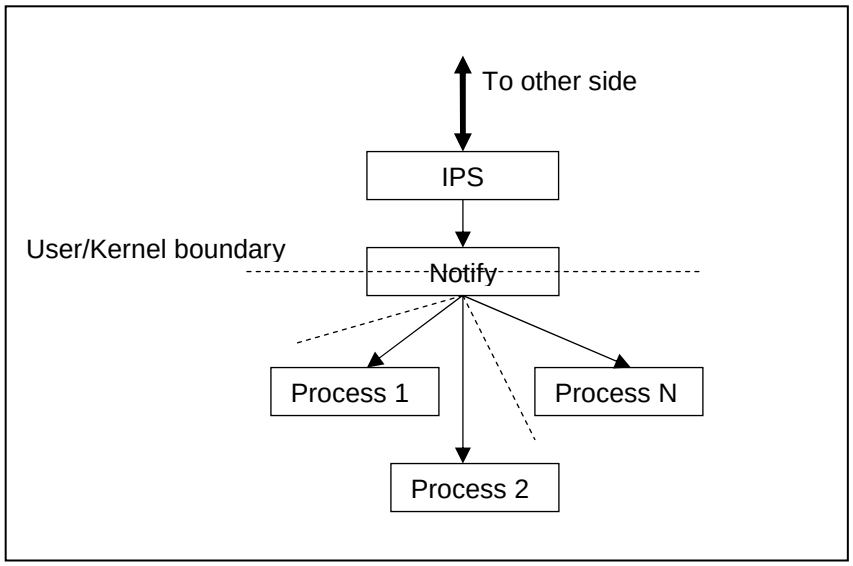

Figure 19. Location of Notify module

The Notify module plugs directly into the IPS module on the side nearer to the IPS module while the farthest side exposes the inherited features of IPS. As seen a dotted line bisects the Notify module into two parts. This is done for OSes where there is a boundary between user and kernel for example, Linux. IPS module in these types of OSes is kernel level module. So first part of Notify module talks to IPS module i.e. it is kernel side implementation. The other part, which is farthest from IPS, uses some predefined ioctl calls to talk to the kernel side of Notify module.

But in other types of OSes, Notify function behaves as transparent proxy between applications and IPS module. So this module does not have much thing to do.

So Linux is the main focal point of the interest. In Linux, Kernel can not call the user mode functions. Then how to invoke the user side IPS event callbacks from the kernel? Also how to provide the same environment as all kernel side callback gets?

Answer to these questions is POSIX signals (RealTime Signals). Let's dig deeper into the signals. Signals are handled in the same fashion as IRQs are. Both have the highest level of prioritization achievable in respective mode. The only difference is, signal are executed in user process context, who has installed the signal handler while IRQ (ISR) does not have any attachment to either kernel or user processes.

Signals can be raised from kernel and can be handled in user process. *(POSIX signals have one advantage over normal signals; they can be queued to a depth of*

max_queued_signals (variable defined in kernel). Users can change the depth via editing /proc/sys/kernel/rtsig-max file (root privileges needed).)

When a user process requests to register an event notification callback, the user side Notify module installs a signal handler for predefined signal and calls the kernel side Notify module to register the user process. Kernel side Notify module stores the user process's information and attached its own callback into the IPS module.

When IPS triggers an event, IPS calls the callback associated with the event i.e. it calls the kernel side Notify module's callback. Which uses the event no to find out which user process to be notified from the information stored at registering. In Linux case, it raises the signal.
Below figure gives the idea about the discussion.

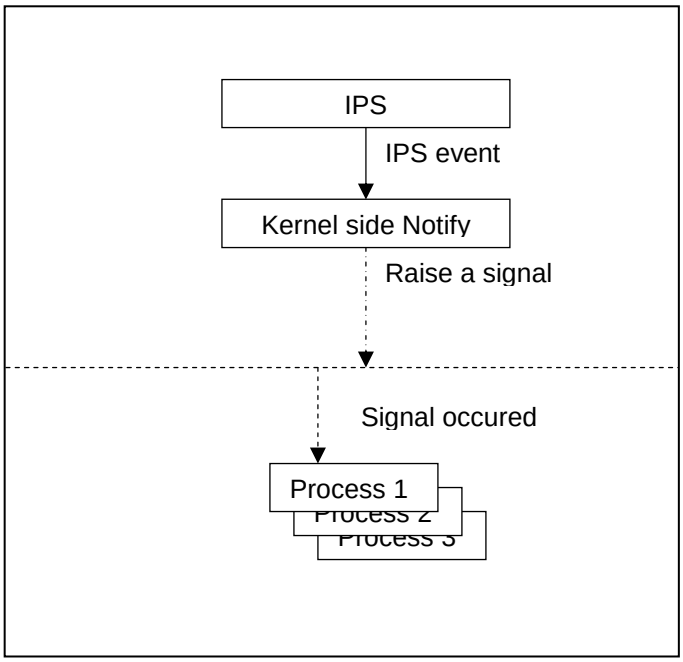

Figure 20. Event passed to the user processes

Based on the above discussion, two alternatives are discussed below:

1. Unique signal for each events:
   Here each event has one unique signal associated with it for example; event zero has first realtime signal and the last event (i.e. 31) is associated with last realtime signal. But there will be only one signal handler installed for all signals. Job of this signal handler is to direct the focus to the correct callback.
   Let's see the prototype of this signal handler:

```
Void Notify_SignalHandler (Int32 sigNo,
                           Siginfo_t * info,
                           void * extra) ;
```

   Here,

   sigNo: generated signal number.
   Info: Signal related information.
   Extra: Don't care.

   So for all signals raised `Notify_SignalHandler` will be called. Using the `sigNo` it can be found which signal was raised. On registering time, if the callback function

and parameters (passed to the callback function during call back) were stored in local array then, using `sigNo` desired callback function can be called.
For passing the IPS generated value, info parameter can be used. This parameter has a field `si_int` which can be set before raising the signal and can be read in the signal handler.
Advantages:

1. Since signals have associated priorities thus event prioritization can be effectively achieved in user processes also.
2. Only one signal handler is used for all signals which mean less code size.

Disadvantages:

1. All RT signals are used up thus user process can not use any of these signals for other purpose.
2. One signal (one event) is used for only one purpose. So this limits to, one callback for one event.

2. One signal for all events:
Here, for all events only one signal is used. Job of this signal handler is to direct the focus to the correct callback.
Let's see the prototype of this signal handler:

```
Void Notify_SignalHandler (Int32 sigNo,
                           Siginfo_t * info,
                           void * extra) ;
```

Here,

sigNo: generated signal number.
Info: Signal related information.
Extra: Don't care.

So for all signals raised `Notify_SignalHandler` will be called. Using the `sigNo` it can be found which signal was raised. On registering time, if the callback function and parameters (passed to the callback function during call back) were stored in local array then, using `sigNo` desired callback function can be called.
For passing the IPS generated value, info parameter can be used. This parameter has a field `si_int` which can be set before raising the signal and can be read in the signal handler.
Advantages:

3. All RT signals are not used so users can use remaining signals for their own purpose.

Disadvantages:

1. Effective event prioritization can not be achieved.

8.4 IPS component (Older Implementation)

The 'IPS' component on a processor is responsible for notifying its peer on the remote processor regarding an event that has occurred on its processor. This component would use the services provided on the H/W platform. It would provide APIs which shall be used by upper layers to establish communication amongst peers at that level.

8.4.1 Alternative 1: IPS component provides basic services to transfer a buffer between the GPP and DSP.

The IPS component provides services to transfer a command (containing a message or data buffer) across the physical link. Only one buffer (notify command) can be transferred at a time.

The events shall be identified by an enumeration.

The shared memory has the following basic areas (order may change):

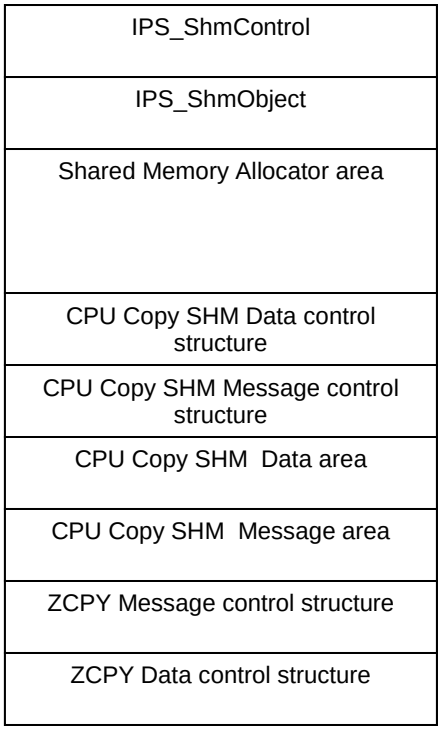

Figure 21. IPS Alternative 1: Shared memory layout

APIs:

a. IPS_Initialize
b. IPS_Register
c. IPS_Send
d. IPS_Notify
e. IPS_Dispatch
f. IPS_Finalize

Enums:

```
typedef enum {
    IPS_CmdMsg  = 0 ;
    IPS_CmdData = 1 ;
} IPS_Cmd
```

IPS structures:

```
/* IPS object and control structure */
typedef struct IPS_Object {
    IPS_CmdObj * cmdIps ;
    IPS_Callback ipsCbck [MAX_IPS_EVENTS] ;
```

```
    Pvoid         cbckArg  [MAX_IPS_EVENTS] ;
#if defined (CHNL_COMPONENT)
    IPS_CmdObj *  cmdData ;
#endif /* if defined (CHNL_COMPONENT) */
#if defined (MSGQ_COMPONENT)
    IPS_CmdObj *  cmdMsg  ;
#endif /* if defined (MSGQ_COMPONENT) */
}


/* Shared IPS control structure */
typedef struct IPS_ShmControl {
    volatile Uint32 ptrCmdToDsp ; /* Ptr./offset to cmdObj */
    volatile Uint32 ptrCmdFmDsp ; /* Ptr./offset to cmdObj */
}


/* Shared IPS command objects */
typedef struct IPS_ShmObject {
    volatile IPS_Cmd cmdToDsp ;
    volatile IPS_Cmd cmdFmDsp ;
}


/* IPS command object */
typedef struct IPS_CmdObj {
    IPS_Cmd event ;
    Uint32   bufPtr ;
    Uint32   bufSize ;
    Uint32   type ; /* Type of command (caller-specific) */
}
```

IPS functions pseudo-code:

```
/* Function to register an event callback with IPS component */
IPS_Register (event, callback, arg) {
    ipsObj->ipsCbck [event] = callback ;
    ipsObj->cbckArg [event] = arg ;
}

/* Function to send a command to the IPS component */
IPS_Send (cmdObj) {
#if defined (MSGQ_COMPONENT)
    /* Prioritize messaging over data transfer */
    if (    (ipsObj->cmdMsg == FREE)
        &&  (cmdObj->event == IPS_CmdMsg)) {
        ipsObj->cmdIps = ipsObj->cmdMsg = cmdObj ;
    }
#endif /* if defined (MSGQ_COMPONENT) */

#if defined (CHNL_COMPONENT)
    if (    (ipsObj->cmdData == FREE)
        &&  (cmdObj->event == IPS_CmdData)) {
        ipsObj->cmdData = cmdObj ;
        if (ipsObj->cmdIps == FREE) {
            ipsObj->cmdIps = cmdObj ;
        }
    }
```

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

    IPS_Dispatch (cmdObj) ;

    /* Return information about whether the message was sent */
}


/* Dispatch the command to the DSP */
IPS_Dispatch (cmdObj) {
    if (ipsShmCtrl->ptrCmdToDsp == FREE) {
        ipsShmCtrl->ptrCmdToDsp = ipsObj->cmdIps ;
        ipsShmCtrl->cmdToDsp = *(ipsObj->cmdIps) ;
        IPS_Notify () ;
        ipsObj->ipsCbck [cmdObj->event] (ipsObj->cbckArg [cmdObj-
>event]) ;
    }

    /* Return information about whether the message was sent */
}


/* Notify the DSP about command */
IPS_Notify () {
    /* Send an interrupt to the DSP */
    InterruptDsp () ;
}


/* Interrupt service routine of the IPS component */
IPS_Isr () {
    /* Get required information from the command */
    tempCmd = ptrCmdFmDsp ;
    ...
    /* Clear received command */
    ptrCmdFmDsp = FREE ;

    /* Make callbacks to registered event handlers */
    tempCmd->ipsCbck [tempCmd->event] (tempCmd->cbckArg [tempCmd-
>event) ;

    /* Dispatch any available command, if free */
    IPS_Dispatch () ;
}
```

Data transfer:

Data transfer multiplexing & issue-reclaim protocol ensures that the DSP & GPP are synchronized and ready for transfer.

It has the following in shared memory (name can be changed):

```
typedef struct CHNL_ShmControl {
    volatile Uint16  dspFreeMask ;
    volatile Uint16  gppFreeMask ;
    volatile Uint16  toDspChnlId ;
    volatile Uint16  fmDspChnlId ;
}
```

These are used by the CHNL modules on both processors to check whether the other processor has a free (issued) buffer before sending a command to the IPS module.

An enum is needed for the command type:

```
typedef enum {
    CHNL_CmdToDsp = 0 ;
    CHNL_CmdFmDsp = 1 ;
} CHNL_Cmd
```

The protocol used is similar to the existing SHM/HPI protocol for multiplexing the channels.

To send a data buffer on output channel:

(At application level):

1. Allocate a buffer (could be kernel buffer or SMA buffer).
2. Fill the buffer with contents.
3. Send the buffer to driver through `CHNL_Issue ()`.

(At driver level)

4. Set the appropriate bit for the channel in the local `outputMask`.
5. Check if the other processor has a buffer free on this channel (by looking at `dspFreeMask`).

   For copy-mode transfer: If the other processor has a buffer available on this channel, check if the output data area is free (check the `toDspChnlId` value. If invalid, output data area is free, otherwise contains the buffer for the `toDspChnlId`.). If yes, copy the buffer data into the output buffer area.
6. Attempt to dispatch the command to the IPS component: `IPS_Send ()`. For this, set the `cmdObj->type` as `CHNL_CmdToDsp`. Add the buffer chirp to the end of used queue for this channel.

To receive a data buffer on input channel:

1. Allocate a buffer (could be kernel buffer or SMA buffer).
2. Send the buffer to driver through `CHNL_Issue ()`.

(At driver level)

3. Set the appropriate bit for the channel in the shared `gppFreeMask`.
4. Add the buffer chirp to the end of channel used queue.

In the callback for SHM copy-mode:

```
/* Channel callback for SHM copy-mode */
SHMCHNL_Callback (cmdObj) {
    if (cmdObj->type == CHNL_CmdToDsp) {
        /* Get the first chirp from the toDspChnlId queue */
        ...
        /* Complete the transfer (AddIOCompletion) */
```

```
            ...
            /* Reset the appropriate bit in outputMask if the
             * queue is now empty.
             */
            ...

            /* 1. Do a round-robin search of channels to find the
             *    next channel that has a buffer free on both GPP &
             *    DSP.
             * 2. If none was found, reset the toDspChnlId to invalid
             *    value.
             * 3. If found, copy the buffer data into the output
             *    buffer area.
             * 4. Get the buffer from the channel used queue (don't
             *    remove from queue) and send to IPS.
             */
            ...
            if (new buffer available) {
                /* Fill IPS command object */
                ...
                IPS_Send (newCmdObj) ;
            }
        }
        else if (cmdObj->type == CHNL_CmdFmDsp) {
            /* Get the first chirp from the fmDspChnlId queue */
            ...
            /* Copy the buffer contents to kernel buffer */
            ...
            /* Complete the transfer (AddIOCompletion) */
            ...
            /* Reset the appropriate bit in dspFreeMask if the
             * queue is now empty.
             */
            ...
        }
    }
```

Callback for ZCPY is the same, but does not involve the buffer copy.

```
/* Channel callback for ZCPY mode */
ZCPYCHNL_Callback (cmdObj) {
    if (cmdObj->type == CHNL_CmdToDsp) {
        /* Get the first chirp from the toDspChnlId queue */
        ...
        /* Complete the transfer (AddIOCompletion) */
        ...
        /* Reset the appropriate bit in outputMask if the
         * queue is now empty.
         */
        ...

        /* 1. Do a round-robin search of channels to find the
         *    next channel that has a buffer free on both GPP &
         *    DSP.
         * 2. If none was found, reset the toDspChnlId to invalid
         *    value.
         * 3. If found, get the buffer from the channel used
```

```
             *    queue (don't remove from queue) and send to IPS.
             */
            ...
            if (new buffer available) {
                /* Fill IPS command object */
                ...
                IPS_Send (newCmdObj) ;
            }
        }
        else if (cmdObj->type == CHNL_CmdFmDsp) {
            /* Get the first chirp from the fmDspChnlId queue */
            ...
            /* Complete the transfer (AddIOCompletion) */
            ...
            /* Reset the appropriate bit in dspFreeMask if the
             * queue is now empty.
             */
            ...
        }
}
```

Message transfer:

There is no specific additional protocol needed for the message transfer within shared memory.

An enum is needed for the command type:

```
typedef enum {
    MSGQ_CmdToDsp = 0 ;
    MSGQ_CmdFmDsp = 1 ;
} MSGQ_Cmd
```

For the SHM copy-mode transfer component, following local structure is needed (name can be changed). Otherwise the flag can be directly kept global, if nothing else is needed in the structure.

```
typedef struct MQT_Object {
    Bool outputFree ;
}
```

Sequence is very similar for SHM copy-mode and zero-copy message transfer.

To send a message:

(At application level)

1. Allocate message (`SMA_Alloc ()` or `MQABUF_Alloc ()` through `MSGQ_Alloc ()`)
2. Fill the message with contents
3. Send the message to driver through `MSGQ_Put ()`.

(At driver level)

4. For copy-mode transfer: Check if the shared memory output message area is free (check the `outputFree` flag). If yes, copy the buffer data into the output message area.

5. If the queue is empty, attempt a dispatch to the IPS component: `IPS_Send ()`. For this, set the `cmdObj->type` as `MSGQ_CmdToDsp`. If the call was not successful (IPS module is already busy), add the message to the end of local message queue.

In the callback for SHM:

```
/* Remote MQT callback for SHM */
SHMRMQT_Callback (cmdObj) {
    if (cmdObj->type == MSGQ_CmdToDsp) {
        MSGQ_Free (cmdObj->bufPtr) ;
        /* Get next message from local queue and send to IPS */
        if (new message available) {
            /* Fill IPS command object */
            ...
            IPS_Send (newCmdObj) ;
        }
    }
    else if (cmdObj->type == MSGQ_CmdFmDsp) {
        /* Send the received message to local MSGQ */
        ...
    }
}
```

Callback for ZCPY is the same, but does not involve freeing the message buffer when it is sent.

```
/* ZCPY Remote MQT callback */
ZCPYRMQT_Callback (cmdObj) {
    if (cmdObj->type == MSGQ_CmdToDsp) {
        /* Get next message from local queue and send to IPS */
        if (new message available) {
            /* Fill IPS command object */
            ...
            IPS_Send (newCmdObj) ;
        }
    }
    else if (cmdObj->type == MSGQ_CmdFmDsp) {
        /* Send the received message to local MSGQ */
        ...
    }
}
```

8.4.2 Alternative 2: IPS component maintains a shared list of messages.

8.4.2.1 IPS component

The IPS component maintains lists of messages, which are shared between the GPP and the DSP. There are two unidirectional lists of messages, for messages to and from the DSP. There are similar lists for data transfer also.

The events shall be identified by an enumeration.

The IPS component shall utilize the services of a generic component that will provide critical section protection between the two processors (`CsObj` object, `CS_Enter ()`, `CS_Leave ()`).

The shared memory has the following basic areas (order may change):

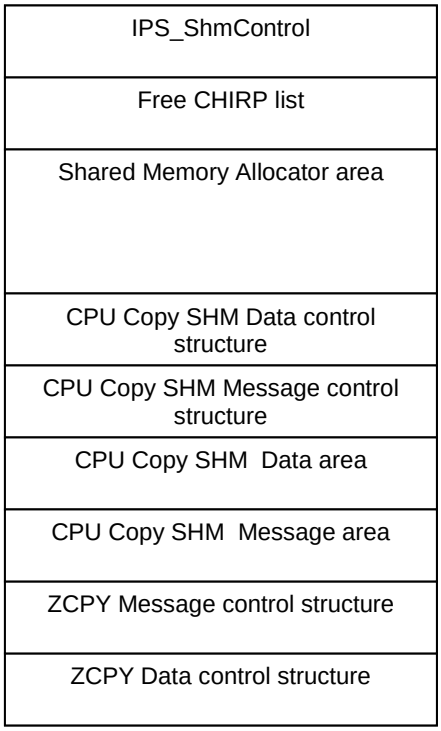

Figure 22. IPS Alternative 2: Shared memory layout

The IPS component maintains the shared lists of messages and CHIRPs within the `IPS_ShmControl` structure. In addition, it separately maintains a free lists of CHRIPs within the shared memory. This list is configured with the maximum number of outstanding requests on all channels at any time. Information for this is obtained from the static configuration.

In addition to the shared lists, shared critical section objects are maintained within the `IPS_ShmControl` structure.

Prioritization between data transfer and messaging happens on the receiving side within the ISR. While sending data or messages, the CHIRPs or messages are simply queued onto the appropriate shared lists. On the receiving side, the message list is checked first and then the data list.

A new API shall be added into the LIST component to initialize a statically instantiated List object (`LIST_Open ()`).

APIs:

a. `IPS_Initialize`
b. `IPS_Finalize`
c. `IPS_Register`
d. `IPS_Send`
e. `IPS_Notify`

Enums:

```
typedef enum {
    IPS_CmdMsg  = 0 ;
    IPS_CmdData = 1 ;
```

```
} IPS_Cmd
```

IPS structures:

```
/* IPS object and control structure */
typedef struct IPS_Object {
    IPS_ShmControl * ptrControl ;
    IPS_Callback     ipsCbck [MAX_IPS_EVENTS] ;
    Pvoid            cbckArg [MAX_IPS_EVENTS] ;
}


/* Shared IPS control structure */
typedef struct IPS_ShmControl {
    volatile Uint32 event ;         /* Indicates interrupt reason */
#if defined (CHNL_COMPONENT)
    volatile List   freeChirpList ; /* Free list of shared CHIRPs */
    volatile List   dataToDspList ; /* List of CHIRPs to DSP */
    volatile List   dataFmDspList ; /* List of CHIRPs from DSP */
    volatile CsObj  freeChirpListCs ; /* CS object for freeChirpList */
    volatile CsObj  dataToDspCs ; /* CS object for dataToDsp list */
    volatile CsObj  dataFmDspCs ; /* CS object for dataFmDsp list */
#endif /* if defined (CHNL_COMPONENT) */
#if defined (MSGQ_COMPONENT)
    volatile List   msgToDspList ; /* List of messages to DSP */
    volatile List   msgFmDspList ; /* List of messages from DSP */
    volatile CsObj  msgToDspCs ; /* CS object for msgToDsp list */
    volatile CsObj  msgFmDspCs ; /* CS object for msgFmDsp list */
#endif /* if defined (MSGQ_COMPONENT) */
}
```

IPS functions pseudo-code:

```
/* Function to initialize the IPS component */
IPS_Intialize () {
    /* Initalize the global IPS_Object object */
    /* Initalize the lists within the IPS_ShmControl area */
    /* Initalize the CS objects within the IPS_ShmControl area */
}

/* Function to finalize the IPS component */
IPS_Finalize () {
    /* Finalize the lists within the IPS_ShmControl area */
    /* Finalize the CS objects within the IPS_ShmControl area */
    /* Finalize the global IPS_Object object */
}

/* Function to register an event callback with IPS component */
IPS_Register (event, callback, arg) {
    ipsObj->ipsCbck [event] = callback ;
    ipsObj->cbckArg [event] = arg ;
}

/* Function to send a command to the IPS component */
/* The bufPtr could be a pointer to a message or a CHIRP */
IPS_Send (event, bufPtr) {
#if defined (MSGQ_COMPONENT)
```

```
    if (event == IPS_CmdMsg) {
        CS_Enter (ptrControl->msgToDspCs) ;
        LIST_PutTail (ptrControl->msgToDspList, bufPtr) ;
        CS_Leave (ptrControl->msgToDspCs) ;
    }
#endif /* if defined (MSGQ_COMPONENT) */

#if defined (CHNL_COMPONENT)
    if (event == IPS_CmdData) {
        CS_Enter (ptrControl->freeChirpListCs) ;
        LIST_GetHead (ptrControl->freeChirpList, &chirp) ;
        CS_Leave (ptrControl->freeChirpListCs) ;

        /* Initialize the CHIRP with buffer details */
        chirp = *bufPtr ;

        CS_Enter (ptrControl->dataToDspCs) ;
        LIST_PutTail (ptrControl->dataToDspList, chirp) ;
        CS_Leave (ptrControl->dataToDspCs) ;
    }
#endif /* if defined (CHNL_COMPONENT) */

    IPS_Notify (event) ;
}


/* Notify the DSP about event */
IPS_Notify (event) {
    ptrControl->event = event ;

    /* Send an interrupt to the DSP */
    InterruptDsp () ;
}


/* Interrupt service routine of the IPS component */
IPS_Isr () {
    /* Check for available messages */
#if defined (MSGQ_COMPONENT)
    if (ptrControl->event == IPS_CmdMsg) {
        while (!LIST_IsEmpty (ptrControl->msgFmDspList)) {
            CS_Enter (ptrControl->msgFmDspCs) ;
            LIST_GetHead (ptrControl->msgFmDspList, &ptr) ;
            CS_Leave (ptrControl->msgFmDspCs) ;
            LIST_PutTail (&tempList, ptr) ;
        }

        /* Make callback to registered event handler */
        ipsObj->ipsCbck [IPS_CmdMsg] (ipsObj->cbckArg [IPS_CmdMsg],
&tempList) ;
    }
#endif /* if defined (MSGQ_COMPONENT) */


    /* Check for available data buffers */
#if defined (CHNL_COMPONENT)
    if (ptrControl->event == IPS_CmdData) {
        while (!LIST_IsEmpty (ptrControl->dataFmDspList)) {
```

```
                CS_Enter (ptrControl->dataFmDspCs) ;
                LIST_GetHead (ptrControl->dataFmDspList, &ptr) ;
                CS_Leave (ptrControl->dataFmDspCs) ;

                LIST_PutTail (&tempList, ptr) ;
            }

            /* Make callback to registered event handler */
            ipsObj->ipsCbck [IPS_CmdData] (ipsObj->cbckArg [IPS_CmdData],
                                           &tempList) ;

            /* Send the used chirps back to the free list. */
            while (!LIST_IsEmpty (ptrControl->tempList)) {
                LIST_GetHead (tempList, &chirp) ;

                CS_Enter (ptrControl->freeChirpListCs) ;
                LIST_PutTail (ptrControl->freeChirpList, &chirp) ;
                CS_Leave (ptrControl->freeChirpListCs) ;
            }
        }
    #endif /* if defined (CHNL_COMPONENT) */
    }
```

8.4.2.2 Data transfer:

While the description for data transfer in this section gives the pseudo-code for the GPP-side components, the counterpart on the DSP shall have exactly the same code except for some of the fields in control structures used for transferring the data buffers.

Data transfer issue-reclaim protocol ensures that the DSP & GPP are synchronized and ready for transfer.

A pointer exchange completes for a GPP output channel, when it sends a buffer to the DSP and receives one in exchange from the DSP. This is ensured by the GPP output channel only sending a buffer to the DSP if it is already ready on its corresponding input channel. When the DSP receives this buffer from the GPP, it sends its corresponding exchange buffer to the GPP. At this point, pointer exchange is complete and the `CHNL_Reclaim ()` succeeds on both processors.

Similarly, a pointer exchange completes for the GPP input channel when it receives a buffer from the DSP. This is ensured by the DSP output channel only sending a buffer to the GPP if it is already ready on that channel. When the GPP receives this buffer, it issues its ready buffer to the DSP. At this point, pointer exchange is complete and the `CHNL_Reclaim ()` succeeds on both processors.

The CHIRP structure shall need to be modified to include the channel ID to which the CHIRP belongs.

The `LDRV_CHNL_AddIoRequest ()` function needs to be modified to not schedule a DPC. The DPC is now completely owned by the DATA component.

Zero-copy data transfer:

For the ZCPY DATA component, following shared structure is needed (name can be changed):

```
typedef struct DATA_ShmControl {
    volatile Uint16  dspFreeMask ; /* Mask indicating DSP free buffer*/
    volatile Uint16  gppFreeMask ; /* Mask indicating GPP free buffer*/
```

```
}
```

Following local structure is also needed (name can be changed):

```
typedef struct DATA_Object {
    Uint16  outputMask ; /* Indicates ready o/p channels */
    Uint16  lastOutput ; /* Last o/p chnl on which data was sent */
}
```

These are used by the CHNL modules on both processors to check whether the other processor has a free (issued) buffer before sending a command to the IPS module.

The protocol used is similar to the existing SHM/HPI protocol for multiplexing the channels. The behavior is same for buffer exchange on both input and output channels.

To exchange buffers on output channel:

(At application level):

1. Allocate a buffer (SMA buffer).
2. Fill the buffer with contents.
3. Send the buffer to driver through `CHNL_Issue ()`.

(At LDRV CHNL level)

4. Add the buffer chirp to the end of `requestList` and send the request to the DATA driver.

(At DATA driver level)

5. Set the appropriate bit for the channel in the local `outputMask`.
6. Schedule the output data DPC.

(At DATA driver level)

7. In the DPC, process requests for multiple channels at the same time. Check for all ready channels (channels having GPP as well as DSP free). For each such ready channel, get a request CHIRP and send to the SHMIPS.
8. On receiving a callback from the SHMIPS component, for each chirp received from the DSP, get a chirp from the `requestList` for the corresponding channel and send it to the SHMIPS component for completing the pointer exchange. Also indicate to the upper layer that IO is complete on the channel by calling `LDRV_CHNL_AddIOCompletion ()`.

(At LDRV CHNL level)

9. `LDRV_CHNL_GetIOCompletion ()` completes and returns.

(At application level):

10. `CHNL_Reclaim ()` completes and returns.

- *The DSP may issue a buffer after the GPP had already checked for it (in which case step 7 does not result in a command being issued to the IPS component). In this condition, the remote processor interrupts the local processor after setting appropriate event in the IPS control structure. This results in a callback to the registered handler, and the protocol ensures that the buffer gets dispatched to the IPS component.*

To exchange buffers on input channel:

(At application level):

1. Allocate a buffer (SMA buffer).
2. Fill the buffer with contents.
3. Send the buffer to driver through `CHNL_Issue ()`.

(At LDRV CHNL level)

4. Add the buffer chirp to the end of `requestList` and send the request to the DATA driver.

(At DATA driver level)

5. Set the appropriate bit for the channel in the shared `gppFreeMask`. Notify the DSP that the GPP is ready on the input channel.
6. On receiving a callback from the SHMIPS component, for each chirp received from the DSP, get a chirp from the `requestList` for the corresponding channel and send it to the SHMIPS component for completing the pointer exchange. Also indicate to the upper layer that IO is complete on the channel by calling `LDRV_CHNL_AddIOCompletion ()`.

(At LDRV CHNL level)

7. `LDRV_CHNL_GetIOCompletion ()` completes and returns.

(At application level):

8. `CHNL_Reclaim ()` completes and returns.

Pseudo-code:

```
/* Channel issue function for ZCPY mode */
ZCPYCHNL_Issue (...) {
    if (IS_OUTPUT_CHNL (chnldId)) {
        /* Set the local outputMask bit for the channel ID */
        SET_BIT (outputMask, chnlId) ;
        DPC_Schedule (outDataDpc) ;
    }
    else { /* Input channel */
        /* Set the bit to indicate buffer free on this channel */
        SET_BIT (chnlShmControl->gppFreeMask, chnlId) ;

        /* Inform the DSP about the free buffer */
        IPS_Notify (IPS_CmdData) ;
    }
}

/* Output channel processing for ZCPY mode */
outDataDpc () {
    /* Process requests for multiple channels at the same time */
    do {
        /* Get channel ID from round-robin search for ready channel */
        /* This function checks for chnl Ids where both GPP and DSP
         * are ready
         */
        FindReadyOutput (&chnlId) ;
```

```
            if (chnlId != INVALID_ID) {
                chirp = LDRV_CHNL_NextRequestChirp (procId, chnlId) ;

                /* Dispatch the IPS command */
                IPS_Send (IPS_CmdData, chirp) ;
            }
        }
        while (chnlId != INVALID_ID) ;
    }

    /* Find a ready output channel */
    FindReadyOutput (ptrChnlId) {
        /* Find a channel ID where both GPP and DSP are ready by looking
         * at the chnlShmControl->dspFreeMask and outputMask
         * Return the channel ID found through the parameter.
         */
    }
```

Callback for ZCPY mode (Note that the actual processing may be done inside a DPC scheduled from the callback. This is implementation-specific):

```
/* Channel callback for ZCPY mode */
ZCPYCHNL_Callback (arg, chirp) {
    /* Issue the buffer for pointer exchange */
    if (chirp != NULL)
        reqChirp = LDRV_CHNL_GetRequestChirp (procId,
                                              chirp->chnlId) ;
        IPS_Send (IPS_CmdData, reqChirp) ;

        /* Indicate to upper layer that IO is complete on this chirp */
        *reqChirp = *chirp;
        AddIoCompletion (..., chirp) ;
    }

    /* Schedule DPC for o/p channel processing in case command was
     * sent by other processor only to indicate availability of free
     * buffer on DSP input channel.
     */
    DPC_Schedule (dataDpc) ;
}
```

Other functions like `CancelIO ()` can be implemented by clearing the pending IO list.

Processor-copy data transfer:

For the PCPY DATA component, following shared structure is needed (name can be changed):

```
typedef struct DATA_ShmControl {
    volatile Uint16  dspFreeMask ; /* Mask indicating DSP free buffer*/
    volatile Uint16  gppFreeMask ; /* Mask indicating GPP free buffer*/
    volatile Uint16  toDspFree ;   /* Mask indicating free toDsp     */
                                   /* buffer area */
    volatile Uint16  fmDspFree ;   /* Mask indicating free fmDsp     */
                                   /* buffer area */
```

```
}
```

Following local structure is also needed (name can be changed):

```
typedef struct DATA_Object {
    Uint16  outputMask ; /* Indicates ready o/p channels */
    Uint16  lastOutput ; /* Last o/p chnl on which data was sent */
}
```

These are used by the CHNL modules on both processors to check whether the other processor has a free (issued) buffer before sending a command to the IPS module.

For processor copy data transfer, there can be a separate shared memory area maintained for the buffer copy for each channel. This shall enable transfer on multiple channels at the same time.

In case multiple buffer areas are not to be reserved for the channels, transfer on only one channel is possible at a time. In that case, the `bufFree` field in the shared memory control structure is a Boolean indicating whether the output buffer area is free or used.

The protocol used is similar to the existing SHM/HPI protocol for multiplexing the channels.

To exchange buffers on output channel:

(At application level):

1. Allocate a buffer (kernel buffer).
2. Fill the buffer with contents.
3. Send the buffer to driver through `CHNL_Issue ()`.

(At LDRV CHNL level)

4. Add the buffer chirp to the end of `requestList` and send the request to the DATA driver.

(At DATA driver level)

5. Set the appropriate bit for the channel in the local `outputMask`.
6. Schedule the output data DPC.

(At DATA driver level)

7. In the DPC, process requests for multiple channels at the same time. Check for all ready channels (channels having GPP as well as DSP free). For each such ready channel, further check if its output data area is free (a transfer is not already in progress on that channel). If free, get a request CHIRP, copy the buffer from request buffer to its SHM area and send to the SHMIPS.
8. On receiving a callback from the SHMIPS component, for each chirp received from the DSP, get a chirp from the `requestList` for the corresponding channel. Copy the buffer from SHM area to the request buffer and inform the DSP about the cleared shared input buffer area. Send the request CHIRP to the SHMIPS component for completing the pointer exchange. Also indicate to the upper layer that IO is complete on the channel by calling `LDRV_CHNL_AddIOCompletion ()`.

(At LDRV CHNL level)

9. `LDRV_CHNL_GetIOCompletion ()` completes and returns.

(At application level):

10. `CHNL_Reclaim ()` completes and returns.

- *The DSP may issue a buffer after the GPP had already checked for it (in which case step 7 does not result in a command being issued to the IPS component). In this condition, the remote processor interrupts the local processor after setting appropriate event in the IPS control structure. This results in a callback to the registered handler, and the protocol ensures that the buffer gets dispatched to the IPS component.*

To exchange buffers on input channel:

(At application level):

1. Allocate a buffer (SMA buffer).
2. Fill the buffer with contents.
3. Send the buffer to driver through `CHNL_Issue ()`.

(At LDRV CHNL level)

4. Add the buffer chirp to the end of `requestList` and send the request to the DATA driver.

(At DATA driver level)

5. Set the appropriate bit for the channel in the shared `gppFreeMask`. Notify the DSP that the GPP is ready on the input channel.
6. On receiving a callback from the SHMIPS component, for each chirp received from the DSP, get a chirp from the `requestList` for the corresponding channel. Copy the buffer from SHM area to the request buffer and inform the DSP about the cleared shared input buffer area. Send the request CHIRP to the SHMIPS component for completing the pointer exchange. Also indicate to the upper layer that IO is complete on the channel by calling `LDRV_CHNL_AddIOCompletion ()`.

(At LDRV CHNL level)

7. `LDRV_CHNL_GetIOCompletion ()` completes and returns.

(At application level):

8. `CHNL_Reclaim ()` completes and returns.

Pseudo-code:

```
/* Channel issue function for SHM copy-mode.
 * This function assumes a reserved shared memory buffer area for each
 * channel to allow transfers on multiple channels within the same
 * interrupt
 */
SHMCHNL_Issue (...) {
    if (IS_OUTPUT_CHNL (chnldId)) {
        /* Set the local outputMask bit for the channel ID */
        SET_BIT (outputMask, chnlId) ;
        DPC_Schedule (outDataDpc) ;
    }
    else { /* Input channel */
        /* Set the bit to indicate buffer free on this channel */
```

```
            SET_BIT (chnlShmControl->gppFreeMask, chnlId) ;
            /* Inform the DSP about the free buffer */
            IPS_Notify (IPS_CmdData) ;
        }
    }

    /* DPC for output channel processing for PCPY mode */
    outDataDpc (...) {
        /* Process requests for multiple channels at the same time */
        do {
            /* Get channel ID from round-robin search for ready channel */
            /* This function checks for chnl Ids where both GPP and DSP
             * are ready
             */
            FindReadyOutput (&chnlId) ;
            if (chnlId != INVALID_ID) {
                /* TEST_AND_SET_BIT is atomic for multiple processors*/
                if (TEST_AND_SET_BIT (pcpyShmCtrl->toDspFree,
                                      chnlId) == TRUE) {
                    chirp = LDRV_CHNL_GetRequestChirp (procId, chnlId) ;

                    /* Copy buffer from request buffer to SHM area */
                    MEM_Copy (shmBuffers[chnlId], chirp->buffer, ...) ;

                    /* Dispatch the IPS command */
                    IPS_Send (IPS_CmdData, chirp) ;
                }
            }
        }
        while (chnlId != INVALID_ID) ;
    }

    /* Find a ready output channel */
    FindReadyOutput (ptrChnlId) {
        /* Find a channel ID where both GPP and DSP are ready by looking
         * at the chnlShmControl->dspFreeMask and outputMask
         * Return the channel ID found through the parameter.
         */
    }
```

Callback for SHM copy-mode (Note that the actual input processing may be done inside a separate DPC scheduled from the callback. This is implementation-specific):

```
    /* Channel callback for SHM copy-mode */
    SHMCHNL_Callback (arg, list) {
        /* Process list of complete CHIRPs. Multiple channels may be
         * ready for data transfer.
         */
        LIST_First (list, &curChirp) ;
        if (curChirp != NULL) {
            /* Issue the buffer for pointer exchange */
            reqChirp = LDRV_CHNL_GetRequestChirp (procId,
                                                  curChirp->chnlId) ;

            /* Copy buffer from SHM area to request buffer */
            MEM_Copy (reqChirp->buffer, shmBuffers [chirp->chnlId], ...) ;

            /* Inform DSP about cleared shared input buffer area */
```

```
            /* CLEAR_BIT is atomic over multiple processors */
            CLEAR_BIT (pcpyShmCtrl->fmDspFree, chnlId) ;

            /* Combine notification and issuing of the exchange buffer */
            IPS_Send (IPS_CmdData, reqChirp) ;

            /* Indicate to upper layer that IO is complete on this chirp */
            AddIoCompletion (..., reqChirp) ;

            /* Process the next chirp in the list */
            LIST_Next (list, &curChirp, &nextChirp) ;
            if (nextChirp != NULL) {
                curChirp = nextChirp ;
            }
        }

        /* Schedule DPC for o/p channel processing in case command was
         * sent by other processor only to indicate availability of free
         * buffer on DSP input channel.
         */
        DPC_Schedule (outDataDpc) ;
    }
```

8.4.2.3 Message transfer

While the description for message transfer in this section gives the pseudo-code for the GPP-side components, the counterpart on the DSP shall have exactly the same code except for some of the fields in control structures used for transferring the messages.

Zero-copy message transfer:

For zero-copy mode, there is no specific additional protocol needed for the message transfer within shared memory.

To send a message:

(At application level)

1. Allocate message (`SMA_Alloc ()` or `MQABUF_Alloc ()` through `MSGQ_Alloc ()`)
2. Fill the message with contents
3. Send the message to driver through `MSGQ_Put ()`.

(At driver level)

4. Directly send the command to the IPS component: `IPS_Send ()`.

To receive a message:

(At application level)

1. Attempt to receive a message from the DSP using `MSGQ_Get ()`.

(At driver level)

2. MQT receives a callback from the SHMIPS component with the received message(s).
3. The MQT sends the message(s) received from the SHMIPS component to the appropriate local message queue(s) using `MSGQ_Put ()`.

(At application level)

4. Get the message received from the DSP (`MSGQ_Get ()` returns).
5. Free the received message using `MSGQ_Free ()`.

- *The MQT may receive the message from the DSP before the application requests for it. In this case, the order of the steps 1 and (2 and 3) will be reversed.*

Pseudo-code:

```
/* Message send function for ZCPY mode */
ZCPYRMQT_Send (...) {
    /* Dispatch the IPS command */
    IPS_Send (IPS_CmdMsg, msg) ;
}
```

In the callback for ZCPY mode (Note that the actual processing may be done inside a DPC scheduled from the callback. This is implementation-specific):

```
/* Remote MQT callback for SHM */
ZCPYRMQT_Callback (arg, list) {
    /* Process list of received messages */
    while (!LIST_IsEmpty (list)) {
        LIST_GetHead (list, &msg) ;
        /* Put the message on the appropriate local MSGQ */
        MSGQ_Put (localMsgq, msg, ...) ;
    }
}
```

Processor-copy message transfer:

For the PCPY MQT component, following shared structure is needed (name can be changed):

```
typedef struct PCPYMQT_ShmControl {
    volatile Uint16  toDspFree ; /* Indicates free toDsp msg area */
    volatile Uint16  fmDspFree ; /* Indicates free fmDsp msg area */
}
```

Following local structure is also needed (name can be changed):

```
typedef struct MQT_Object {
    List * toDsp ;
}
```

To send a message:

(At application level)

1. Allocate message (`SMA_Alloc ()` or `MQABUF_Alloc ()` through `MSGQ_Alloc ()`)
2. Fill the message with contents
3. Send the message to driver through `MSGQ_Put ()`.

(At driver level)

4. Add the message to the end of local message list.

5. Schedule a DPC to transfer the messages.
6. In the output message DPC: For all messages in the local message list:

   If the output message area is free:

   - Set the `toDspFree` field in the `pcpyShmCtrl` structure to claim the output message area.
   - Get the first pending output message from the `toDsp` local list.
   - Copy the message into the output message area.
   - Send the shared memory message to the IPS component.
   - Free the user message.

To receive a message:

(At application level)

1. Attempt to receive a message from the DSP using `MSGQ_Get ()`.

(At driver level)

2. MQT receives a callback from the SHMIPS component with the received message.
3. MQT allocates a new message using `MSGQ_Alloc ()`.
4. It copies the contents of received message into the newly allocated message.
5. It clears the `fmDspFree` field in the `pcpyShmCtrl` structure to release the input message area and sends a notification to the SHMIPS.
6. Finally, it sends the message to the appropriate local message queue using `MSGQ_Put ()`.

(At application level)

7. Get the message received from the DSP (`MSGQ_Get ()` returns).
8. Free the received message using `MSGQ_Free ()`.

❑ *The MQT may receive the message from the DSP before the application requests for it. In this case, the order of the steps 1 and (2 to 6) will be reversed.*

Pseudo-code:

```
/* Message send function for SHM copy-mode */
/* This function assumes a reserved shared memory buffer area for
 * a message */
SHMRMQT_Send (...) {
    LIST_PutTail (toDsp, msg) ;
    DPC_Schedule (msgDpc) ;
}

/* DPC function for sending PCPY mode messages */
msgDpc (...) {
    /* TEST_AND_SET is atomic over multiple processors */
    if (TEST_AND_SET (pcpyShmCtrl->toDspFree) == TRUE) {
        LIST_GetHead (toDsp, &msg) ;
```

```
            if (msg != NULL) {
                /* Copy message to SHM area */
                ...
                /* Dispatch the IPS command */
                IPS_Send (IPS_CmdMsg, shmMsg) ;
                MSGQ_Free (msg) ;
            }
        }
    }
```

In the callback for SHM mode (Note that the actual processing could possibly be done inside a DPC scheduled from the callback. This is implementation-specific):

```
/* Remote MQT callback for SHM */
SHMRMQT_Callback (arg, msg) {
    if (msg != NULL) {
        MSGQ_Alloc (&newMsg, ...) ;
        newMsg = msg ;

        /* Inform DSP about cleared shared input message area */
        /* CLEAR is atomic over multiple processors */
        CLEAR (pcpyShmCtrl->fmDspFree) ;
        IPS_Notify (IPS_CmdMsg) ;

        /* Put the message on the appropriate local MSGQ */
        MSGQ_Put (localMsgq, newMsg, ...) ;
    }
    else { /* Will not be done always if interrupts are optimized */
        DPC_Schedule (msgDpc) ;
    }
}
```

8.4.3 Chosen alternative

The alternative 2 has been chosen for the shared memory link IPS component design, since it provides a high-performance and code-size optimized design for CPU copy as well as zero-copy data transfer and message transfer.