Configuring Interrupts on Keystone Devices

From Texas Instruments Wiki
Jump to: navigation, search

Contents

Objective

The objective of this wiki page is to introduce the reader to interrupts and their software setup and debugging on Keystone devices, using TI's TMS320C6678 device as an example.

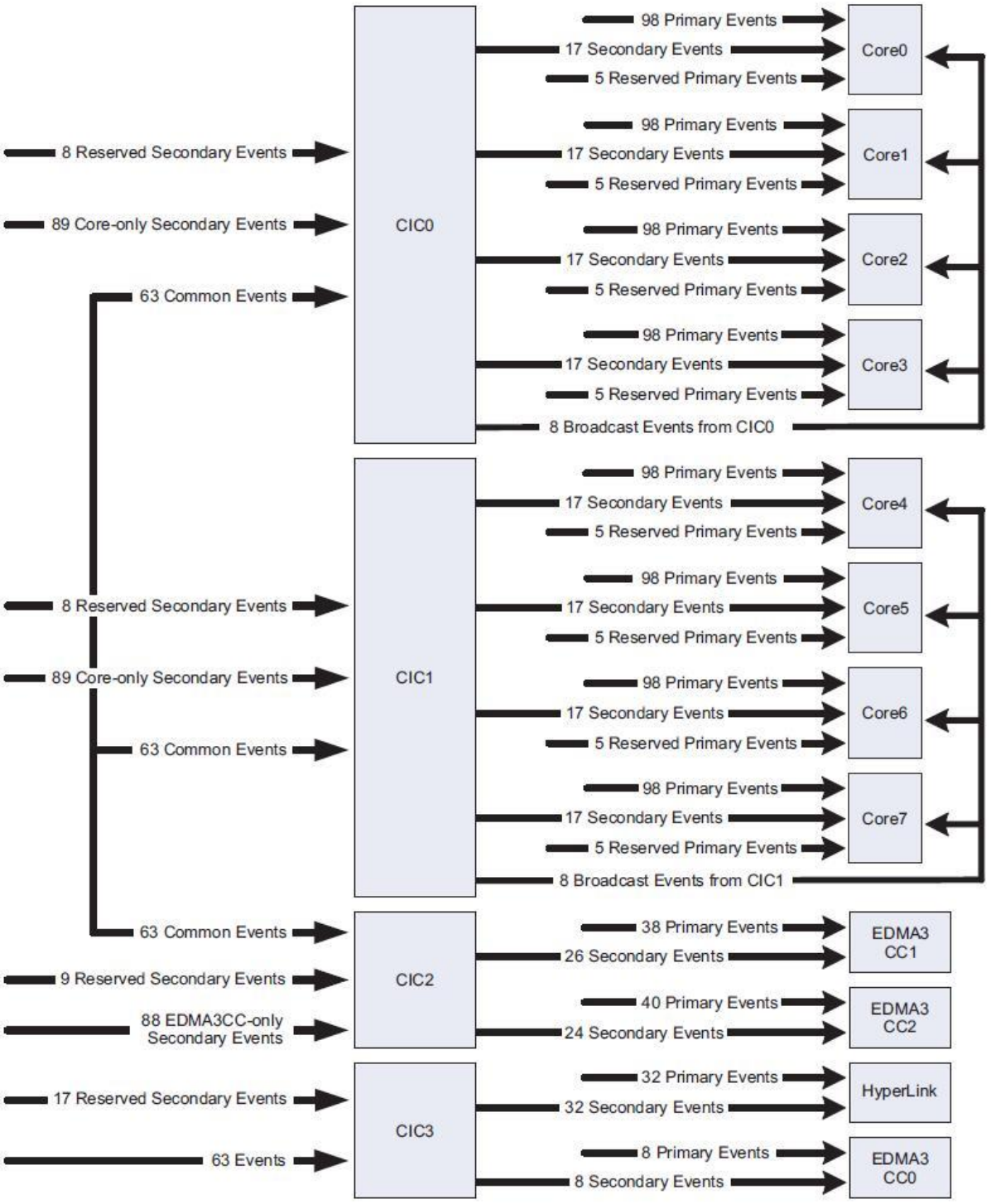
The first part consolidates some key concepts on the interrupt controller, drawn from the relevant user guides. The second part discusses the software implementation and delves into the two primary approaches for interrupt setup.

Key Concepts

The KeyStone Architecture has many peripherals and a large number of event sources. The use of events is completely dependent on a user's specific application, which drives the need for maximum flexibility; how interrupts or events are serviced is completely up to software control. Both the EDMA3 channel controllers (EDMA3CC) and the C66x CorePacs are capable of receiving events directly. However, the number of accepted events for each EDMA3CC and C66x CorePac is limited. [1]

A KeyStone device can have hundreds of events. Therefore, some of these events need to be aggregated at the chip level through the Chip-level Interrupt Controller (also known as CIC or CpIntc), before they are routed to the EDMA3CC and C66x CorePacs. It is important to note here, that CIC/CpIntc is different from the Interrupt Controller inside a C66x CorePac, which is denoted by the acronym INTC. To achieve the aggregation, multiple CICs are added to the SoC. The CIC takes chip-level events (system events) and generates host interrupts which act as event inputs to the EDMA3CC and C66x CorePac by combining and/or selecting those chip-level events. [1]

The figure below shows the 6678 interrupt topology along with the CICs and interrupt routing [3]. For more information on how the CIC operates internally please see the CIC user guide as indicated in reference [2].

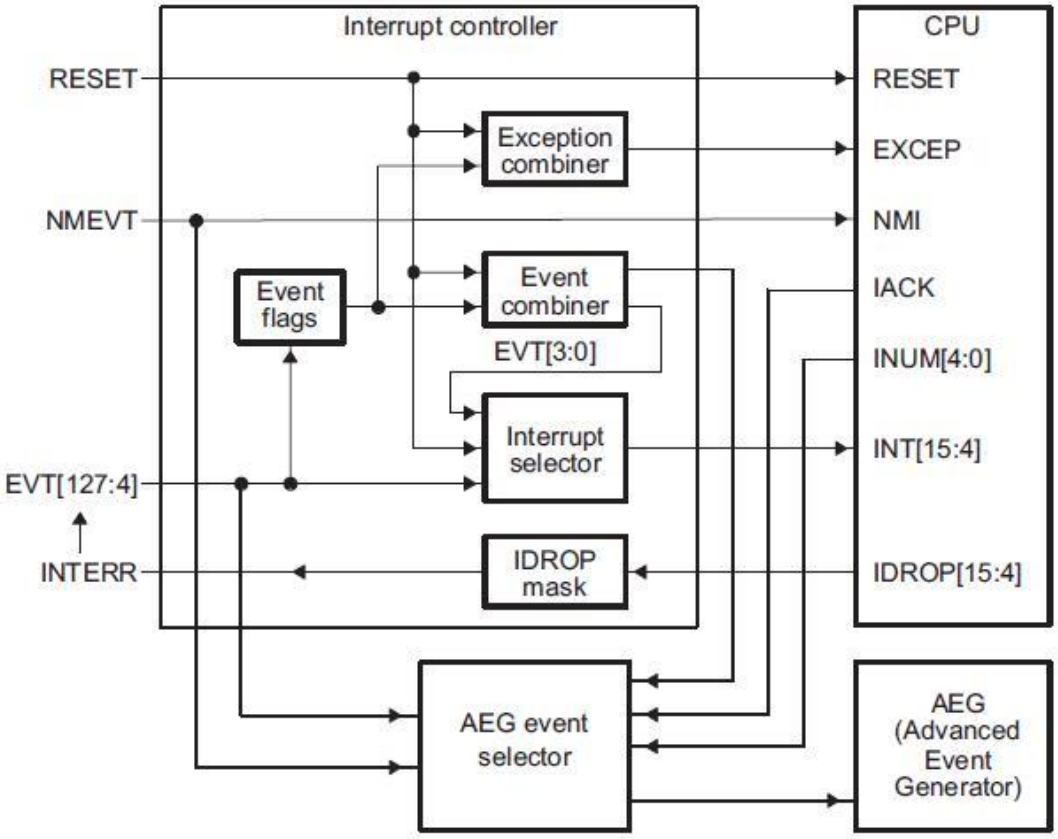

8 Reserved Secondary Events
89 Core-only Secondary Events
63 Common Events
CIC0
98 Primary Events
17 Secondary Events
5 Reserved Primary Events
Core0
Core1
Core2
Core3
8 Broadcast Events from CIC0
8 Reserved Secondary Events
89 Core-only Secondary Events
63 Common Events
CIC1
Core4
Core5
Core6
Core7
8 Broadcast Events from CIC1
63 Common Events
9 Reserved Secondary Events
88 EDMA3CC-only Secondary Events
CIC2
38 Primary Events
26 Secondary Events
40 Primary Events
24 Secondary Events
EDMA3 CC1
EDMA3 CC2
17 Reserved Secondary Events
63 Events
CIC3
32 Primary Events
32 Secondary Events
8 Primary Events
8 Secondary Events
HyperLink
EDMA3 CC0

The C66x DSP provides two types of asynchronous signaling services: Interrupts and Exceptions. Interrupts provide the means to redirect normal program flow due to the presence of an external or internal hardware signal. Exceptions are similar in that they also redirect program flow, but they are normally associated with error conditions in the system. The C66x DSP can receive 12 maskable/configurable interrupts, 1 maskable exception, and 1 unmaskable interrupt/exception. The C66x CorePac interrupt controller INTC allows up to 124 system events to be routed to the DSP interrupt/exception inputs. These 124 events can either be directly connected to the maskable interrupts, or grouped together as interrupts or exceptions. [2]

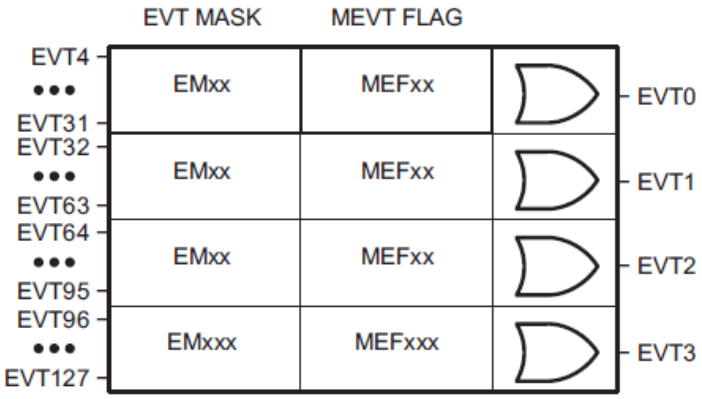
The figure below shows the CorePac INTC [2]. The 124 event IDs are represented by EVT[127:4], and the 12 maskable/configurable interrupts are represented by INTC[15:4].

The 124 system events are stored in four 32-bit registers known as the Event Flag Registers and each event is mapped to a specific flag bit. When a system event is received at INTC, the corresponding Event Flag Register bit is set accordingly. Note that the Event Flag Registers are read-only; the Event Set Register and Event Clear Register can be used to manually set or clear any of the bits in the Event Flag Register. [2]

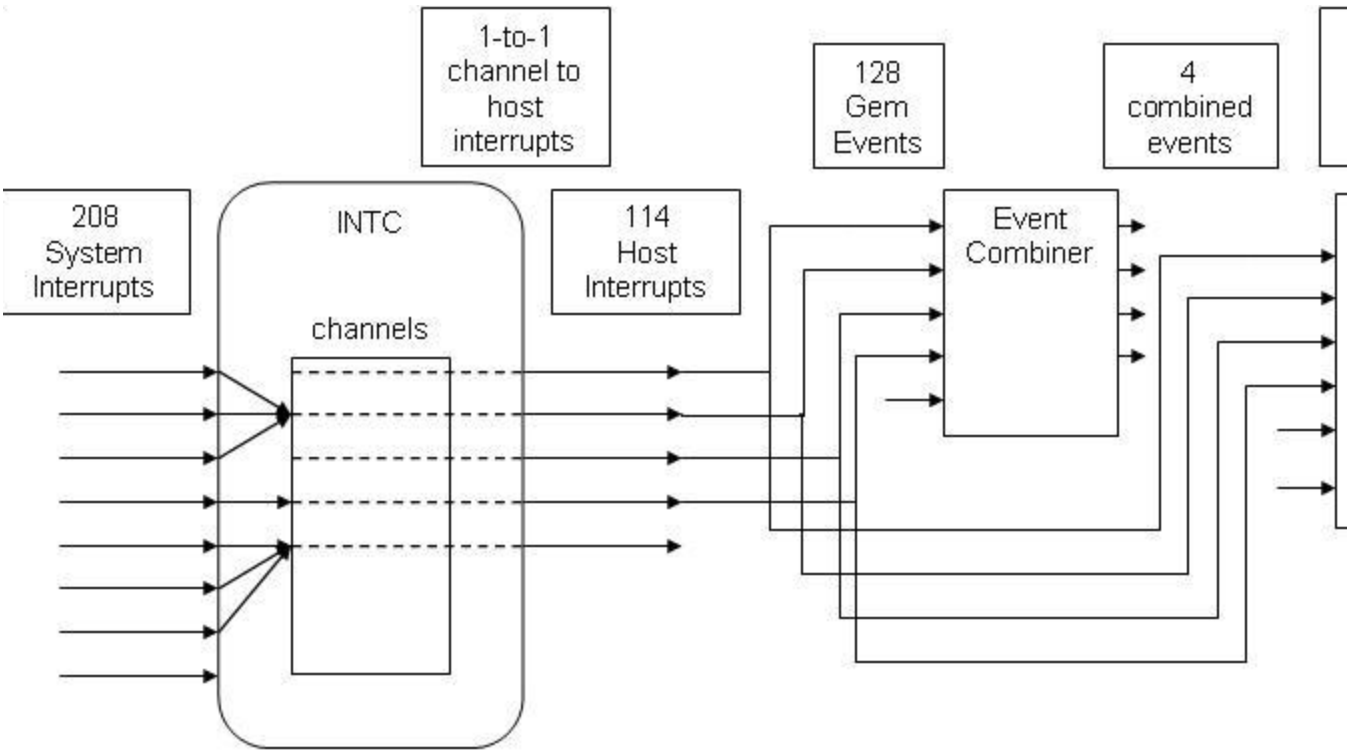
The Event Combiner allows multiple system events to be combined into a single event.


Interrupt controller
CPU
RESET
RESET
Exception combiner
EXCEP
NMEVT
NMI
Event flags
Event combiner
IACK
EVT[3:0]
INUM[4:0]
Interrupt selector
INT[15:4]
EVT[127:4]
INTERR
IDROP mask
IDROP[15:4]
AEG event selector
AEG (Advanced Event Generator)

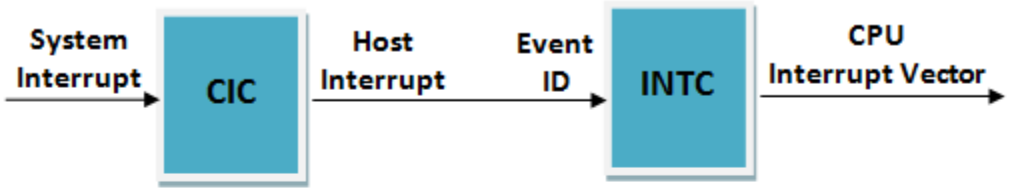
The figure below shows a logical representation of the Event Combiner in the Keystone CorePac. The 124 system events are divided into 4 groups, as shown in the figure below. The Event Combiner logic has the capability of grouping multiple event inputs to 4 possible event outputs, EVT[3:0]. These outputs are then provided as inputs to the interrupt selector and treated as additional system events. This is illustrated in the figure above [2]



The following diagram summarizes the key process flow for interrupt handling on Keystone devices.



The figure below shows a high-level block diagram and identifies some key terminology that should be kept in mind as we delve into the next section on software implementation and API usage.



CIC	Chip-level Interrupt Controller
INTC	CorePac Interrupt Controller
System Interrupt	Input to CIC
Host Interrupt	Output from CIC
Event ID	Input to INTC
CPU Interrupt Vector	Output from INTC

Software Implementation

There are two primary software components that provide APIs that can be leveraged to configure interrupts on Keystone devices. The Multicore Software Development Kit (MCSDK) contains both software packages, viz. SYS/BIOS and Platform Development Kit (PDK), that enable developers to program interrupts. MCSDK can be downloaded from the weblink provided in the references at the end of this document. [4]

Programmers are free to use either approach though there are some general guidelines that can be kept in mind when making a choice:

- If application developers plan to use SYS/BIOS RTOS on their device, then it is recommended that they leverage the relevant SYS/BIOS Interrupt APIs to configure interrupts, and use CSL APIs only where an equivalent function does not exist in the SYS/BIOS APIs. However, here it is important to note that if application developers use both SYS/BIOS and CSL APIs for the purpose of mapping interrupts and writing to the Interrupt Service Table Pointer (ISTP), there will be conflicts since both CSL and SYS/BIOS will assume that they own ISTP.
- If the system does not use the SYS/BIOS RTOS, then CSL APIs can be used to map and configure interrupts.

Using CSL APIs

The specific module in the PDK that contains the relevant APIs for interrupt programming is the Chip Support Library (CSL). CSL APIs for the CorePac Interrupt Controller aka INTC are of the form *CSL_intcxxx* and for the Chip-level Interrupt Controller aka CIC are of the form *CSL_CPINTCxxx*. We'll first focus on configuring the CorePac's INTC and later delve into APIs for configuring CIC.

Configuring CorePac's INTC

The general CSL approach to interrupt mapping for INTC is as shown in the code snippet below. In this example, event ID 63 is mapped to interrupt vector 4. If you'd like to change what event triggers the interrupt, first lookup the event ID, which is listed in the device-specific data manual (for 6678 this is listed in Table 7-38 in reference [3]). Once you have identified the event ID (let's represent this by xx), modify the code snippet below to replace *CSL_INTC_EVENTID_63* with *CSL_INTC_EVENTID_xx* as the second argument to the *CSL_intcOpen* function call. Similary, if you'd like to change the interrupt vector from 4 to another vector id number (between 4 and 15), assign the new interrupt vector represented by *xx* as *CSL_INTC_VECTID_xx* to the variable *vectId* in the code below.

```
{
   CSL_IntcObj intcObj63p;
   CSL_IntcGlobalEnableState state;

   CSL_IntcContext context;

   CSL_Status intStat;
   CSL_IntcParam vectId;

   context.numEvtEntries = 0;
   context.eventhandlerRecord = NULL;

   CSL_intcInit(&context);

   CSL_intcGlobalNmiEnable();
   intStat = CSL_intcGlobalEnable(&state);

   vectId = CSL_INTC_VECTID_4;
   hIntc63 = CSL_intcOpen (&intcObj63, CSL_INTC_EVENTID_63,
&vectId, NULL);

   EventRecord.handler = &event63Handler;
   EventRecord.arg = hIntc63;

   CSL_intcPlugEventHandler(hIntc63,&EventRecord);
   CSL_intcHwControl(hIntc63,CSL_INTC_CMD_EVTENABLE,NULL);

   CSL_IntcClose(hIntc63);
}

void event63Handler(CSL_IntcHandle hIntc){
  . . .
}
```

The *CSL_IntcContext* is the structure that is used to store the current INTC context and incorporates four elements: the event handler record, the event allocation mask, the number of event entries and an offset. The event handler record consists of the event handler, i.e. the ISR function that will be invoked when a particular event occurs, and the corresponding argument that will be passed to this ISR function. The number of event entries corresponds to the number

of events that the programmer plans to map. The event allocation mask consists of 4 arrays of 32-bit values, where each bit represents one of the 128 events. The offset map is an array of 128 elements that stores whether each of the 128 events has been mapped to a CPU interrupt or not, and whether the event handler associated is valid or not.

In the next step in the example we enable global and NMI interrupts using the *CSL_intcGlobalEnable()* and *CSL_intcGlobalNmiEnable()* APIs.

Mapping events to interrupts is achieved using the *CSL_intcOpen(...)* API. The API reserves an interrupt-event for use and returns a valid handle to the event only if the event is not currently allocated. In this example, we map event ID 63 to CPU Interrupt vector 4.

To define the ISR that should be run when this event occurs, we first update the event handler record of the INTC context. In this example, we point the event handler to "test_isr_handler", which is the function we want executed when event 63 occurs. The *CSL_intcPlugEventHandler(...)* API is then used to tie an event-handler to an event, so that when the event occurs the relevant event-handler is invoked. The *CSL_intcHwControl(...)* API is then used to enable the event, in this case event 63.

Finally the *CSL_intcClose(...)* API is called to de-allocate and release the event. Once this is called the INTC handle can no longer be used to access the event; further accesses to event resources are possible only on "opening" an event object again.

Configuring CIC

CSL APIs for configuring CIC are of the form *CSL_CPINTC*_xxx. The following code snippet shows some typical CSL APIs and the general flow for CIC configuration.

```
/* Disable all host interrupts. */
CSL_CPINTC_disableAllHostInterrupt(hnd);

/* Configure no nesting support in the CPINTC Module. */
CSL_CPINTC_setNestingMode (hnd, CPINTC_NO_NESTING);

/* We now map System Interrupt 0 - 3 to channel 3 */
CSL_CPINTC_mapSystemIntrToChannel (hnd, 0 , 2);
CSL_CPINTC_mapSystemIntrToChannel (hnd, 1 , 4);
CSL_CPINTC_mapSystemIntrToChannel (hnd, 2 , 5);
CSL_CPINTC_mapSystemIntrToChannel (hnd, 3 , 3);

/* Enable system interrupts 0 - 3 */
CSL_CPINTC_enableSysInterrupt (hnd, 0);
CSL_CPINTC_enableSysInterrupt (hnd, 1);
CSL_CPINTC_enableSysInterrupt (hnd, 2);
CSL_CPINTC_enableSysInterrupt (hnd, 3);

/* Enable Host interrupt 3 */
CSL_CPINTC_enableHostInterrupt (hnd, 3);

/* Enable all host interrupts also. */
```

```
CSL_CPINTC_enableAllHostInterrupt(hnd);
```

The comments in the code snippet provide an explanation of what most of the code tries to achieve. Note that the call to function *CSL_CPINTC_setNestingMode*, with second argument *CPINTC_NO_NESTING*, configures "nesting" to be deactivated. The term "nesting" refers to a method that enables developers to configure CIC such that when a current interrupt is being serviced, certain specified interrupts are disabled. The typical usage is to nest on the current interrupt and disable all interrupts of the same or lower priority (or channel). Nesting mode is not supported in KeyStone devices. So the information is not in the user's guide and it is disabled in the above code snippet and in CSL examples.

For a more detailed understanding of the CSL for CIC configuration, we encourage you to download the MCSDK, and import and explore the Code Composer Studio project at the MCSDK install path, under the folder 'pdk_C6678_x_x_x_xx\packages\ti\csl\example\cpintc'

For a more detailed understanding of CSL, including interrupt-related APIs, please see the API documentation in the MCSDK install path on your computer at 'pdk_C6678_x_x_x_xx\packages\ti\csl\docs\csldocs.chm'

Example

With this example, we try to leverage what we have learned so far to setup an interrupt for Hyperlink on TI's 6678 multicore device.

```
/*** --- INTC Initializations --- ***/

/* Note that hyplnk_EXAMPLE_COREPAC_VEC = 4,
hyplnk_EXAMPLE_COREPAC_INT_INPUT = 0x15, */
/* CSL_INTC_CMD_EVTCLEAR = 3, CSL_INTC_CMD_EVTENABLE = 0 */

CSL_IntcParam vectId = hyplnk_EXAMPLE_COREPAC_VEC;
Int16 eventId = hyplnk_EXAMPLE_COREPAC_INT_INPUT;
CSL_IntcGlobalEnableState   state;

/* INTC module initialization */
hyplnkExampleIntcContext.eventhandlerRecord =
hyplnkExampleEvtHdlrRecord;
hyplnkExampleIntcContext.numEvtEntries      = 2;
CSL_intcInit(&hyplnkExampleIntcContext);

/* Enable NMIs */
CSL_intcGlobalNmiEnable();

/* Enable global interrupts */
CSL_intcGlobalEnable(&state);

hyplnkExampleIntcHnd = CSL_intcOpen (&hyplnkExampleIntcObj,
eventId, &vectId, NULL);
hyplnkExampleEvtHdlrRecord[0].handler = hyplnkExampleIsr;
hyplnkExampleEvtHdlrRecord[0].arg = (void *)eventId;
```

```
CSL_intcPlugEventHandler(hyplnkExampleIntcHnd,
hyplnkExampleEvtHdlrRecord);

/* Clear the event in case it is pending */
CSL_intcHwControl(hyplnkExampleIntcHnd, CSL_INTC_CMD_EVTCLEAR,
NULL);

/* Enable event */
CSL_intcHwControl(hyplnkExampleIntcHnd, CSL_INTC_CMD_EVTENABLE,
NULL);

/*** --- CIC Initializations --- ***/

CSL_CPINTC_Handle hnd;
hnd = CSL_CPINTC_open (0);

/* Disable all host interrupts. */
CSL_CPINTC_disableAllHostInterrupt(hnd);

/* Configure no nesting support in the CPINTC Module */
CSL_CPINTC_setNestingMode (hnd, CPINTC_NO_NESTING);

/* Clear Hyperlink system interrupt number 111 */
/* We get the interrupt number from Table 7-39 in the 6678 */
/* data manual at http://www.ti.com/lit/ds/sprs691c/sprs691c.pdf
*/
CSL_CPINTC_clearSysInterrupt (hnd, CSL_INTC0_VUSR_INT_O);

/* Enable Hyperlink system interrupt number 111 on CIC0 */
CSL_CPINTC_enableSysInterrupt (hnd, CSL_INTC0_VUSR_INT_O);

/* Map System Interrupt to Channel. */
/* Note that hyplnk_EXAMPLE_INTC_OUTPUT = 32 + (11 * CoreNumber)
= 43 for Core0*/
CSL_CPINTC_mapSystemIntrToChannel (hnd, CSL_INTC0_VUSR_INT_O,
hyplnk_EXAMPLE_INTC_OUTPUT);

/* Enable the Host Interrupt */
CSL_CPINTC_enableHostInterrupt (hnd, hyplnk_EXAMPLE_INTC_OUTPUT);

CSL_CPINTC_enableAllHostInterrupt(hnd);
```

Note that in 6678 there is one-to-one mapping between host interrupt and channel, so we do not need to use the *CSL_CPINTC_mapChannelToHostInterrupt* API

Using SYS/BIOS

SYS/BIOS is installed as part of the MCSDK installation. Three API modules within SYS/BIOS are of particular interest for interrupt processing: HWI, EventCombiner and CpIntc. The HWI module is used to configure the CorePac's INTC; the EventCombiner module is used to configure the Event Combiner; and the CpIntc module is used to configure CIC to map system interrupts to host interrupts. In this section we delve into each of these modules.

HWI

This module provides APIs for managing hardware interrupts. The *ti.sysbios.family.c64p.Hwi* module provides APIs that implement HWI functions specific to the C64x+ and C66x devices.

The code snippet below shows an example of creating an HWI instance. In this example, event ID 10 is mapped to interrupt vector 5. If you'd like to change what event triggers the interrupt, first lookup the event ID, which is listed in the device-specific data manual (for 6678 this is listed in Table 7-38 in reference [3]). Once you have identified the event ID modify the code snippet below to assign the relevant HWI params element (*hwiParams.eventId*) to the new event ID. If you'd like to change the interrupt vector from 5 to another vector id number (between 4 and 15), use the new interrupt vector number as the first argument to the *Hwi_create* function call.

```
#include <xdc/runtime/Error.h>
#include <ti/sysbios/hal/Hwi.h>

Hwi_Handle myHwi;

Int main(Int argc, char* argv[])
{
    Hwi_Params hwiParams;
    Error_Block eb;

    Hwi_Params_init(&hwiParams);
    Error_init(&eb);

    // set the argument you want passed to your ISR function
    hwiParams.arg = 1;

    // set the event id of the peripheral assigned to this
interrupt
    hwiParams.eventId = 10;

    // don't allow this interrupt to nest itself
    hwiParams.maskSetting = Hwi_MaskingOption_SELF;

    //
    // Configure interrupt 5 to invoke "myIsr".
    // Automatically enables interrupt 5 by default
    // set params.enableInt = FALSE if you want to control
    // when the interrupt is enabled using Hwi_enableInterrupt()
    //

    myHwi = Hwi_create(5, myIsr, &hwiParams, &eb);

    if (Error_check(&eb)) {
        // handle the error
    }
}

Void myIsr(UArg arg)
```

```
{
    // this runs when interrupt #5 goes off
}
```

EventCombiner

The event combiner allows the user to combine up to 32 system events into a single combined event. The events 0, 1, 2, and 3 are the events associated with the event combiner. Using the EventCombiner module along with the Hwi module, allows the user to route a combined event to any of the 12 maskable CPU interrupts available on GEM. The EventCombiner supports up to 128 system events. Users can specify a function and an argument for each system event and can choose to enable whichever system events they want.

There are two ways you can setup EventCombiner. One way is to do this statically in your RTSC configuration file (.cfg) in your CCS project. The other way is to do this in your C code. Both methods are illustrated in the code snippets below.

Method 1: In .cfg file
This code snippet combines events 15 and 16, and uses interrupt vector 4

```
var EventCombiner =
xdc.useModule('ti.sysbios.family.c64p.EventCombiner');
EventCombiner.events[15].unmask = true;
EventCombiner.events[15].fxn = '&event15Fxn';
EventCombiner.events[15].arg = 0x15;
EventCombiner.events[16].unmask = true;
EventCombiner.events[16].fxn = '&event16Fxn';
EventCombiner.events[16].arg = 0x16;
```

Method 2: In C file
This code snippet combines events 4 and 5, and uses interrupt vector 8

```
eventId = 4;
EventCombiner_dispatchPlug(eventId, &event4Fxn, arg, TRUE);
EventCombiner_dispatchPlug(eventId + 1, &event5Fxn, arg, TRUE);

Hwi_Params_init(&params);
params.arg = (eventId / 32);
params.eventId = (eventId / 32);
params.enableInt = TRUE;

intVector = 8;

Hwi_create(intVector, &EventCombiner_dispatch, &params, NULL);
```

CpIntc

This module manages the CIC hardware. This module supports enabling and disabling of both system and host interrupts. This module also supports mapping system interrupts to host interrupts and host interrupts to Hwis or to the EventCombiner. These functionality are supported statically and during runtime for CICs connected to the GEM interrupt controller but only during runtime for other CICs. There is a dispatch function for handling GEM hardware interrupts triggered by a system interrupt. The Global Enable Register is enabled by default in the module startup function.

System interrupts are those interrupts generated by a hardware module in the system. These interrupts are inputs into CIC. Host interrupts are the output interrupts of CIC. There is a one-to-one mapping between channels and host interrupts therefore, the term "host interrupt" is also used for channels. Note that this modules does not support prioritization, nesting, and vectorization.

The SYS/BIOS module that provides the CpIntc APIs is 'ti.sysbios.family.c66.tci66xx.CpIntc.'

The following code snippet provides an example implementation using CpIntc APIs. The comments before each line of code explain what the code achieves.

```
// Map system interrupt 15 to host interrupt 8
CpIntc_mapSysIntToHostInt(0, 15, 8);

// Plug the function for event #15
CpIntc_dispatchPlug(15, &event15Fxn, 15, TRUE);

// Enable host interrupt #8
CpIntc_enableHostInt(0, 8); // enable host interrupt 8
```

Example

Here we put together our learning from this section and walk through a SYS/BIOS example.

```
/* Map the System Interrupt i.e. the Interrupt Destination 0
interrupt to the DIO ISR Handler. */
CpIntc_dispatchPlug(CSL_INTC0_INTDST0,
(CpIntc_FuncPtr)myDioTxCompletionIsr, (UArg)hSrioDrv, TRUE);

/* The configuration is for CPINTC0. We map system interrupt 112
to Host Interrupt 8. */
CpIntc_mapSysIntToHostInt(0, CSL_INTC0_INTDST0, 8);

/* Enable the Host Interrupt. */
CpIntc_enableHostInt(0, 8);

/* Enable the System Interrupt */
CpIntc_enableSysInt(0, CSL_INTC0_INTDST0);

/* Get the event id associated with the host interrupt. */
eventId = CpIntc_getEventId(8);
```

```
Hwi_Params_init(&params);

/* Host interrupt value*/
params.arg = 8;

/* Event id for your host interrupt */
params.eventId = eventId;

/* Enable the Hwi */
params.enableInt = TRUE;

/* This plugs the interrupt vector 4 and the ISR function. */
/* When using CpIntc, you must plug the Hwi fxn with
CpIntc_dispatch */
/* so it knows how to process the CpIntc interrupts.*/
Hwi_create(4, &CpIntc_dispatch, &params, NULL);
```

Analyzing Interrupts in Code Composer Studio

TI's Code Composer Studio provides the ability to view the device memory map and register values as you step through your code.

Here we provide a GEL file with functions that you can use to analyze your interrupt use case.

To download the GEL file please click here

Please change the extension from .txt to .gel and load in CCS. If you are new to GEL files, please see http://processors.wiki.ti.com/index.php/GEL

References

[1] CorePac User Guide http://www.ti.com/litv/pdf/sprugw0b

[2] CIC User Guide http://www.ti.com/litv/pdf/sprugw4a

[3] C6678 Data Manual http://www.ti.com/lit/ds/symlink/tms320c6678.pdf

[4] MCSDK Download Page http://www.ti.com/tool/bioslinuxmcsdk

[5] SYS/BIOS & HWI e2e Forum Post
http://e2e.ti.com/support/embedded/bios/f/355/t/94262.aspx

[6] Nesting e2e Forum Post http://e2e.ti.com/support/dsp/c6000_multi-core_dsps/f/639/t/189559.aspx

[7] CpIntc e2e Forum Post http://e2e.ti.com/support/embedded/bios/f/355/p/203249/724263.aspx#724263

Navigation menu

Personal tools

- Log in
- Request account

Namespaces

- Page
- Discussion

Variants

Views

- Read
- View source
- View history

More

Search

Search Go

Navigation

- Main Page
- All pages

- All categories
- Recent changes
- Random page
- Help

Toolbox

- What links here
- Related changes
- Special pages
- Printable version
- Permanent link
- Page information

- This page was last edited on 27 September 2012, at 19:50.
- Content is available under Creative Commons Attribution-ShareAlike unless otherwise noted.

- Privacy policy
- About Texas Instruments Wiki
- Disclaimers
- Terms of Use

- 

-