

Using the CSL to Complement the OS Dispatcher in Handling Cascaded Interrupts

RG Kiran
Platform Software Group

ABSTRACT

This application report discusses how CSL's INTC module could be used to share the task of dispatching interrupts with the OS, in scenarios where the OS's interrupt dispatcher does not comprehend cascaded interrupts.

The solution involves having the CSL dispatch the cascaded interrupts alone, while completely leaving the job of dispatching primary CPU interrupts to the OS.

We also discuss an example application – of handling Cascading Interrupts in DSP/BIOS on the OMAP5912 platform.

Contents

1	Introduction	2
2	Constraints	2
	2.1 Adaptation of the CSL with the OS	2
	2.2 Multiple Interrupt Controllers	3
3	Design Details	3
	3.1 The Event Handler	3
	3.2 CSL Dispatcher	4
	3.3 Adapter Routine	4
	3.4 Initialization Code	5
	3.5 Putting it All Together	5
	3.5.1 Initialization	6
	3.5.2 The Adapter ISR	6
	3.5.3 The Event Handler	7
	3.6 Control Flow	7
Appendix A	Example: Handling Cascaded Interrupts in DSP/BIOS on OMAP5912	9
	A.1 The Example	9
	A.2 Cascading	9
	A.3 The Adapter ISR	10
	A.4 Configuration Specifics	10
	A.4.1 Static Configuration – Using the CDB File	10
	A.4.2 Dynamic Configuration: using 'HWI_dispatchPlug(...)'	12
	A.5 Control Flow	12

Trademarks are the property of their respective owners.

List of Figures

Figure A–1. Interrupt Controller Cascading on OMAP5912/C55	9
Figure A–2. Registering the Adapter ISR to HWI	10
Figure A–3. Specifying the ISR Routine	11
Figure A–4. Specifying the ISR Attributes	11

1 Introduction

Interrupt dispatching involves maintaining an association of interrupt service routines (ISR) to interrupts, and transferring control to the appropriate ISR on the reception of an interrupt.

The interrupt controller (INTC) CSL module provides an interrupt dispatcher that comprehends all the interrupts on a chip.

In the presence of an operating system (OS), interrupt dispatching is typically handled by the interrupt manager in the OS. Hence, the CSL dispatcher generally finds use only in environments where an OS is missing.

But there are scenarios where the native operating system understands the top-level (or core) interrupts, but does not cater to interrupts on cascaded interrupt controllers. This is particularly true of operating system kernels that have been written for, and makes use of, the features of a specific CPU subsystem, but have not been modified to understand all the idiosyncrasies of a full system-on-chip built around the core.

A solution is discussed here to handle such cases; where, the CSL can complement the interrupt manager of the OS by handling the cascaded interrupts.

2 Constraints

The proposed solution extends support for cascaded interrupt dispatching using the CSL while continuing to keep the CSL code and design independent of any OS.

2.1 Adaptation of the CSL with the OS

Cascaded interrupt controllers convey the reception of an interrupt on one of its lines via one (or more) of the core interrupt lines to which they are wired. Thus, the triggering of one of the cascaded interrupts would result in an interrupt on the CPU-core.

Since the OS dispatches all the core interrupts, the CSL dispatcher (that is now expected to dispatch the cascaded interrupts) would need to be hooked to the OS's interrupt dispatcher such that it gets invoked whenever a cascaded interrupt occurs. The CSL dispatcher would, thus, need to be like any other ISR to the OS.

Operating systems typically mandate that all ISRs registered with its interrupt manager conform to certain code-templates – with respect to, say, the signature of the ISR, entry/exit sequences to be followed, etc.

On the other hand, CSL aims to abstract the underlying hardware in a manner that is totally independent of any particular operating system (only provide a mechanism, do not impose a policy); so much so that, by design, CSL does not even assume the presence of an underlying operating system. The design of the CSL dispatcher is, therefore, OS-agnostic and cannot directly obey the requirements of any one operating system.

Hence, an adapter function that adapts the CSL dispatcher to the OS is defined. The adapter routine accommodates the ISR requirements of the OS and yet invokes the CSL dispatcher in the necessary manner. It should be noted that the adapter code is **not** part of the CSL library. It is to be treated as part of the OS's platform-specific code-base subject to porting.

It is actually this adapter that is directly registered with the OS interrupt manager as an ISR, to all those core-interrupts onto which cascaded interrupt controllers are wired.

2.2 Multiple Interrupt Controllers

There is a possibility that multiple core-interrupts are used to cascade other interrupt controllers. The adapter routine(s) would need to be hooked to each one of these core-interrupts. In order to be able to provide a black-box view of the cascaded interrupts, the CSL dispatcher takes in an argument that the adapter can use to identify the source of the interrupt. That way a single CSL dispatcher can take care of all the cascaded interrupt controllers.

3 Design Details

3.1 The Event Handler

The CSL maintains an association of ISRs — or event handlers, in CSL lingo — to interrupts (events) in an internal array of event-handler records.

The CSL user registers an event handler with an interrupt by using the 'CSL_intcPlugEventHandler' API as follows.

```
{
    CSL_IntcObj intcObj;
    CSL_IntcHandle hIntc;
    CSL_IntcHwSetup setup;
    CSL_IntcEventHandlerRecord isrRec;
    CSL_Status status;
    hIntc = CSL_intcOpen (
        &intcObj,
        CSL_INTC_EVENTID_I2C,
        NULL,
        &status
    );
    if (status != CSL_SOK) {
        printf("interrupt open failed!\n");
        return -1;
    }
    setup.priority = CSL_INTC_PRIORITY_DEFAULT;
    setup.sense = CSL_INTC_SENSE_FALL_EDGE;
    setup.type = CSL_INTC_TYPE_FIQ;
    CSL_intcHwSetup(hIntc, &setup);
    isrRec0.handler = my_i2c_isr; /* the event-handler for i2c interrupt */
    isrRec.arg = (void *)i2c_handle; /* an argument to be passed to the ISR */
    CSL_intcPlugEventHandler(hIntc, &isrRec);
    return 0;
}
```

NOTE: *The interrupt controller interface to the interrupt must be configured appropriately; otherwise, the controller might not register an interrupt.*

Since all cascaded interrupts are dispatched by the CSL, the user must register the ISRs for all cascaded interrupts with the CSL dispatcher.

It must be noted that the ISRs registered with the CSL dispatcher must not only comply with prescribed C-calling conventions, but must also conform to the following prototype mandated by the CSL:

```
typedef void (* CSL_IntcEventHandler)(void *);
```

Therefore an ISR-framework would look as follows:

```
void my_i2c_isr (void *arg)
{
    /* "service" the interrupt */
}
```

3.2 CSL Dispatcher

The CSL dispatcher – ‘CSL_intcDispatcher’ – is a C-callable function with the following prototype:

```
inline void CSL_intcDispatcher (int intNum);
```

where ‘intNum’ identifies the CPU interrupt that was triggered.

NOTE: *The CSL dispatcher has been made an ‘inline’ function to avoid the overhead of a call, when invoked.*

Here is how the CSL dispatcher functions:

- On invocation, it attempts to identify the exact source of the interrupt that was triggered by appropriately probing the interrupt controllers.
- It would then try to find a valid event-handler associated with the interrupt by looking into the internal array of event-handler records.
- On finding a valid event-handler, the dispatcher transfers control to it via a C-call.
- On return from the event-handler, the dispatcher performs any necessary clean up; this might, for example, include acknowledgement of the interrupt with the interrupt controllers (thereby instructing it to process the next pending interrupt and present it to the CPU).

3.3 Adapter Routine

As discussed in section 2.1, it would not be possible to design a CSL dispatcher that would suit all operating systems, since the structure of an ISR would vary from one OS to another. The use of an adapter function is required to fit the CSL dispatcher into the OS and extend the facility of dispatching to cascaded-interrupts.

The adapter routine is just another ISR from the OS’s perspective. Its primary task, however, is to invoke the CSL dispatcher appropriately.

Since there could be multiple interrupt controllers cascaded from the core interrupt lines, the adapter must identify the core-interrupt number that was triggered and convey its interrupt number via an argument to the CSL dispatcher. The CSL dispatcher would proceed with this information in locating the exact source.

Here is an example adapter for a hypothetical OS scenario:

```

/* an example OS-CSL "adapter" ISR */
void OS_adapter_isr(int intNum, void *arg, OS_IsrContext *isrContext)
{
    /*
    intNum    -> interrupt number (on the core)
    arg       -> custom argument specified during ISR hook-up (not used here)
    isrContext -> OS-specific interrupt context information
    */

    /* some OS-mandated OS-specific call */
    OS_isr_enter(isrContext);

    /* get the interrupt number, and call the CSL dispatcher with the
    interrupt number as argument */
    CSL_intcDispatcher(intNum);

    /* some OS-mandated OS-specific call */
    OS_isr_exit(isrContext);
}
    
```

The adapter should be hooked to the core-interrupts (wired to cascaded interrupt controllers) using OS-provided mechanisms. It is recommended that this be done before all CSL INTC related initialization (section 3.4).

3.4 Initialization Code

By default, during the initialization of the CSL INTC module (specifically in the call to 'CSL_intcDispatcherInit'), the CSL dispatcher gets plugged into the core's interrupt vectors so as to enable full interrupt dispatching.

In our particular case, the core's interrupt vector area is 'owned' and managed by the native OS. The user must, therefore, indicate to the CSL that it should not modify the core interrupt vectors via a flag in the INTC-Context structure passed to the 'CSL_intcInit', as follows:

```

static CSL_IntcContext  intcContext = CSL_INTC_CONTEXT_INITVAL;
static CSL_IntcDispatcherContext intcDispatcherContext =
CSL_INTC_DISPATCHERCONTEXT_INITVAL;

/* to prevent CSL from "corrupting" the interrupt vector area */
intcContext.flags |= CSL_INTC_CONTEXT_DISABLECOREVECTORWRITES;

if (CSL_intcInit(&intcContext) != CSL_SOK) {
    printf("CSL_intcInit(..) failed\n");
    return -1;
}

if (CSL_intcDispatcherInit(&intcDispatcherContext) != CSL_SOK) {
    printf("CSL_intcDispatcherInit(..) failed\n");
    return -1;
}
    
```

3.5 Putting it All Together

Below are code snippets showing the various steps. Let us assume interrupts 7 and 11 are connected to cascaded interrupt controllers. We define and hook-up the adapter ISR to the OS on these interrupts. Suppose the I2C interrupt-line was connected to the cascaded interrupt controller on core-interrupt 7. The ISR for this interrupt should therefore be registered with the CSL (using 'CSL_intcPlugEventHandler') as shown.

3.5.1 Initialization

```

int init_stuff (void)
{
    static CSL_IntcContext  intcContext = CSL_INTC_CONTEXT_INITVAL;
    static CSL_IntcDispatcherContext intcDispatcherContext =
        CSL_INTC_DISPATCHERCONTEXT_INITVAL;

    /* hook the adapter with the OS on CPU-interrupt 7, which is wired to a
       cascaded interrupt controller. */

    /* the 'OS_hookIsr' API is assumed to take as parameters, the interrupt
       number (to hook the ISR to), the ISR address and an argument to be
       passed to the ISR, respectively.
    */
    if (OS_hookIsr(7 /*intNum*/, OS_adapter_isr /*ISR*/, NULL /*arg*/)){
        printf("OS_hookIsr(7) failed\n");
        return -1;
    }

    /* hook the adapter with the OS on CPU-interrupt 11, which is wired to
       a cascaded interrupt controller */
    if (OS_hookIsr(11, OS_adapter_isr, NULL)) {
        printf("OS_hookIsr(11) failed\n");
        return -1;
    }

    /* to prevent CSL from corrupting the interrupt vector area */
    intcContext.flags |= CSL_INTC_CONTEXT_DISABLECOREVECTORWRITES;

    /* INTC module initialization */
    if (CSL_intcInit(&intcContext) != CSL_SOK) {
        printf("CSL_intcInit(..) failed\n");
        return -1;
    }

    /* does the dispatcher related initialization and enables the core-interrupts
       wired to cascaded interrupt controllers */
    if (CSL_intcDispatcherInit(&intcDispatcherContext) != CSL_SOK) {
        printf("CSL_intcDispatcherInit(..) failed\n");
        return -1;
    }

    return 0;
}

```

3.5.2 The Adapter ISR

```

void OS_adapter_isr(int intNum, void *arg, OS_IsrContext *isrContext)
{
    /*
    intNum    -> interrupt number (on the core)
    arg       -> custom argument specified during ISR hook-up
    isrContext -> OS-specific interrupt context information
    */

    /* some OS-mandated OS-specific call */
    OS_isr_enter(isrContext);
    /* get the interrupt number, and call the CSL dispatcher with the
       interrupt number as argument */
    CSL_intcDispatcher(intNum);
    /* some OS-mandated OS-specific call */
    OS_isr_exit(isrContext);
}

```

3.5.3 The Event Handler

The 'CSL_intcPlugEventHandler' in the 'isr_plug' function (in the code below) populates the CSL event-handler record for the cascaded interrupt, whereas 'OS_hookIsr' in the 'init_stuff' function (in section 3.5.1, discussed earlier) populates the OS dispatcher table for primary CPU interrupts wired to cascaded interrupt controllers.

```

/* registering the ISR with the CSL because the interrupt is on a cascaded interrupt
controller. The routine also configures the interrupt controller interface for the
interrupt */
int isr_plug(void)
{
    CSL_IntcObj intcObj;
    CSL_IntcHandle hIntc;
    CSL_IntcHwSetup setup;
    CSL_IntcEventHandlerRecord isrRec;
    CSL_Status status;

    hIntc = CSL_intcOpen (
        &intcObj,
        CSL_INTC_EVENTID_I2C,
        NULL,
        &status
    );

    if (status != CSL_SOK) {
        printf("interrupt open failed!\n");
        return -1;
    }

    setup.priority = CSL_INTC_PRIORITY_DEFAULT;
    setup.sense = CSL_INTC_SENSE_FALL_EDGE;
    setup.type = CSL_INTC_TYPE_FIQ;
    CSL_intcHwSetup(hIntc, &setup);
    isrRec0.handler = my_i2c_isr; /* the event-handler for i2c interrupt */
    isrRec.arg = (void *)i2c_handle; /* an argument to be passed to the ISR */
    CSL_intcPlugEventHandler(hIntc, &isrRec);
    return 0;
}

/* the ISR that would get called on an I2C interrupt */
void my_i2c_isr (void *arg)
{
    /* "service" the interrupt */
}

```

3.6 Control Flow

1. I2C peripheral conveys an interrupt to the cascaded interrupt controller.
2. The cascaded interrupt controller registers the interrupt, and in turn, conveys the interrupt to the CPU on core-interrupt line 7 (assuming the controller was wired to that interrupt on the core).
3. CPU initiates the interruption sequence and invokes the vector corresponding to interrupt #7. Since the OS dispatcher controls all the core interrupts, it gets invoked.
4. The OS dispatcher refers to its internal data-structures to find the corresponding ISR and transfers control to it. Since the adapter ISR ('OS_adapter_isr') is hooked to interrupt line 7, it gets called.

5. The Adapter ISR executes any necessary OS-mandated ISR-entry sequence and invokes the CSL dispatcher (which is inline, and would typically not result in a call) conveying to it that core interrupt #7 was triggered.
6. CSL dispatcher queries the interrupt controller connected to core-interrupt 7 to find the interrupt-line on the controller that triggered the interrupt. It finds that it corresponds to the I2C interrupt.
7. The CSL dispatcher checks the event-handler record corresponding to the I2C interrupt in its internal array of event handler records. Finds the address to 'my_i2c_isr' in the record and the value of the argument to be passed to it.
8. The CSL dispatcher calls 'my_i2c_isr' with the corresponding argument.
9. 'my_i2c_isr' services the interrupt before returning control to the CSL dispatcher.
10. The CSL dispatcher acknowledges the servicing of the interrupt with the interrupt controller, if required, and returns control to the Adapter ISR – the acknowledgement would instruct the controller to process the next interrupt.
11. The adapter ISR executes any necessary OS-mandated ISR-exit sequence and returns control to the OS dispatcher (that had invoked it).
12. The OS dispatcher would eventually return from the interrupt context leaving the CPU to resume execution of the interrupted code.

Appendix A Example: Handling Cascaded Interrupts in DSP/BIOS on OMAP5912

A.1 The Example

The example demonstrates how the CSL could be used along with DSP/BIOS on the C55-core of OMAP5912, such that the core-interrupts are managed and dispatched by the HWI dispatcher (of DSP/BIOS) while using the CSL INTC dispatcher for the cascaded-interrupts.

A.2 Cascading

Figure A–1 shows the wiring of the cascaded interrupt controllers (L2.0 and L2.1) with respect to the OMAP5912 C55-core. As shown, the C55-core provides 32 interrupt lines, of which core-interrupt line 3 is cascaded to L2.0 interrupt controller (providing 16 additional interrupts) and core-interrupt lines 6 and 17 are connected to interrupt controller L2.1 (which adds another 64 interrupt lines).

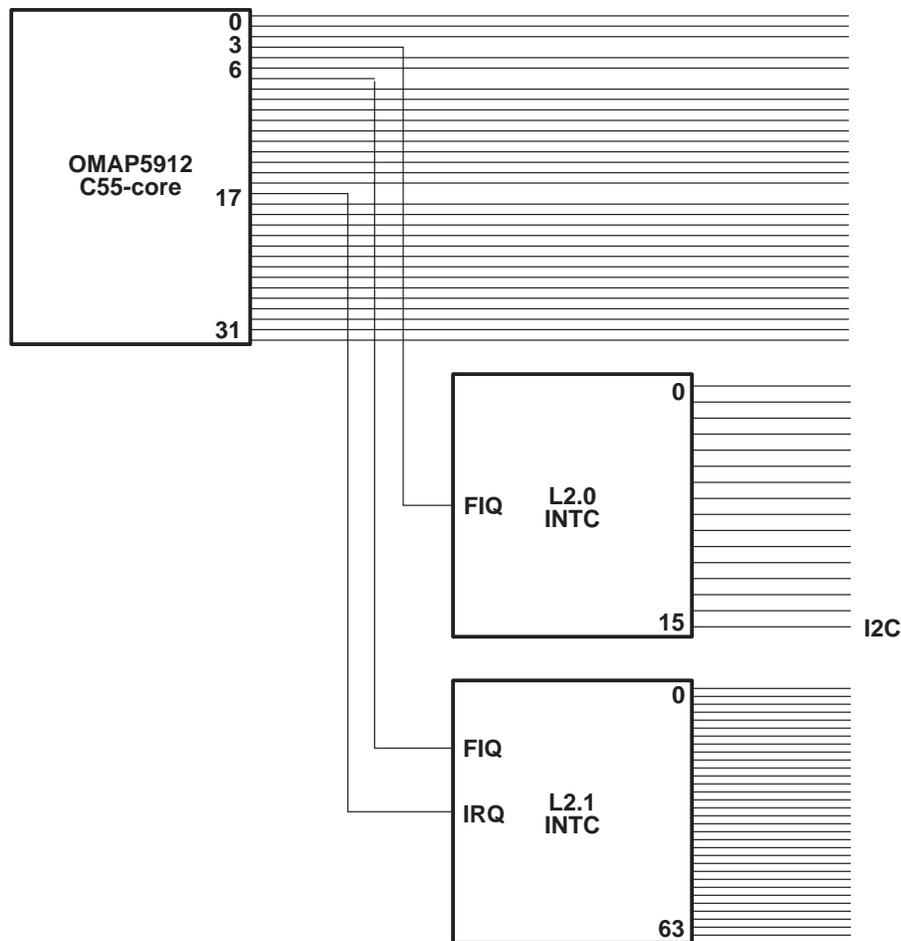


Figure A–1. Interrupt Controller Cascading on OMAP5912/C55

A.3 The Adapter ISR

The HWI module (interrupt manager) of DSP/BIOS allows for a C-function to be hooked-up as an ISR only if the HWI dispatcher is in use. The DSP/BIOS Configuration Tool could be used to statically configure whether HWI dispatcher should be used for each interrupt (section A.4.1). In order to do the hook-up dynamically, the 'HWI_dispatchPlug' function could be used (section A.4.2).

HWI would invoke the ISR with one user-specified argument. We hook-up the same adapter function to all the three core-interrupts (3, 6, 17) such that in each case the argument passed could be used to identify the interrupt number (see sections A.4.1 and A.4.2). This greatly simplifies the implementation of the adapter:

```
void bioshwi_cslintc_adapter(void *arg)
{
    /* the 'arg' actually contains the interrupt-number on the core
    -- by configuration during hook-up */
    CSL_intcDispatcher((int)arg);
}
```

A.4 Configuration Specifics

DSP/BIOS provides two separate mechanisms to hook-up the ISR to the HWI interrupt manager as detailed in the following two sections:

A.4.1 Static Configuration – Using the CDB File

Figure A–2, Figure A–3, and Figure A–4 are snapshots of steps to be followed to register the Adapter ISR to HWI on interrupt 17 using the CDB configuration file. The same steps must be repeated for interrupts 3 and 6. The only difference for interrupts 3 and 6 would be the Arg to be passed to the ISR, which would match the interrupt numbers (3 and 6 respectively).

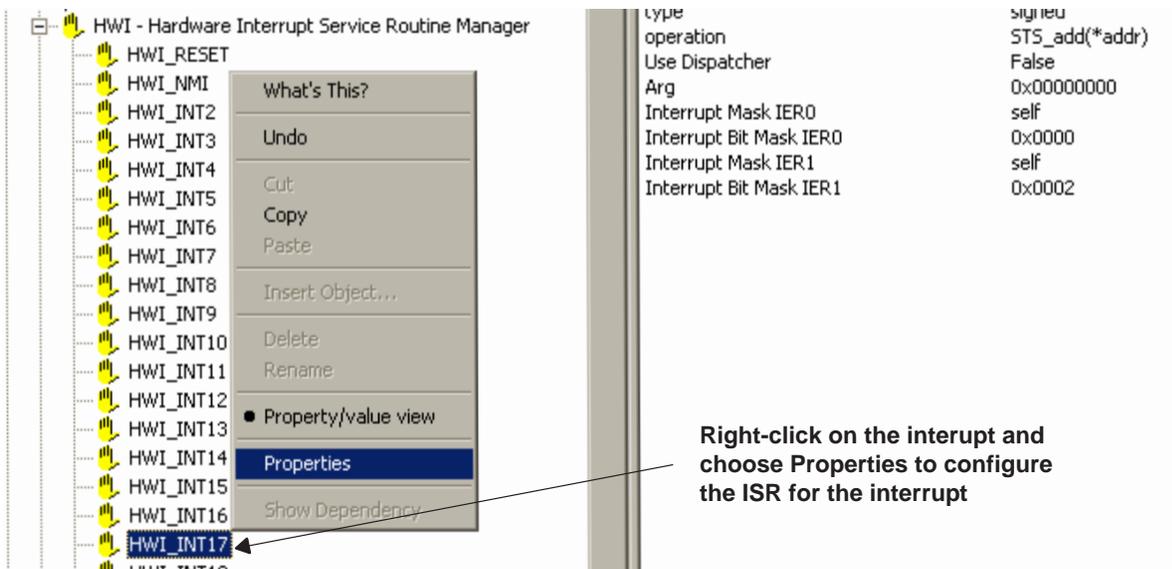


Figure A–2. Registering the Adapter ISR to HWI

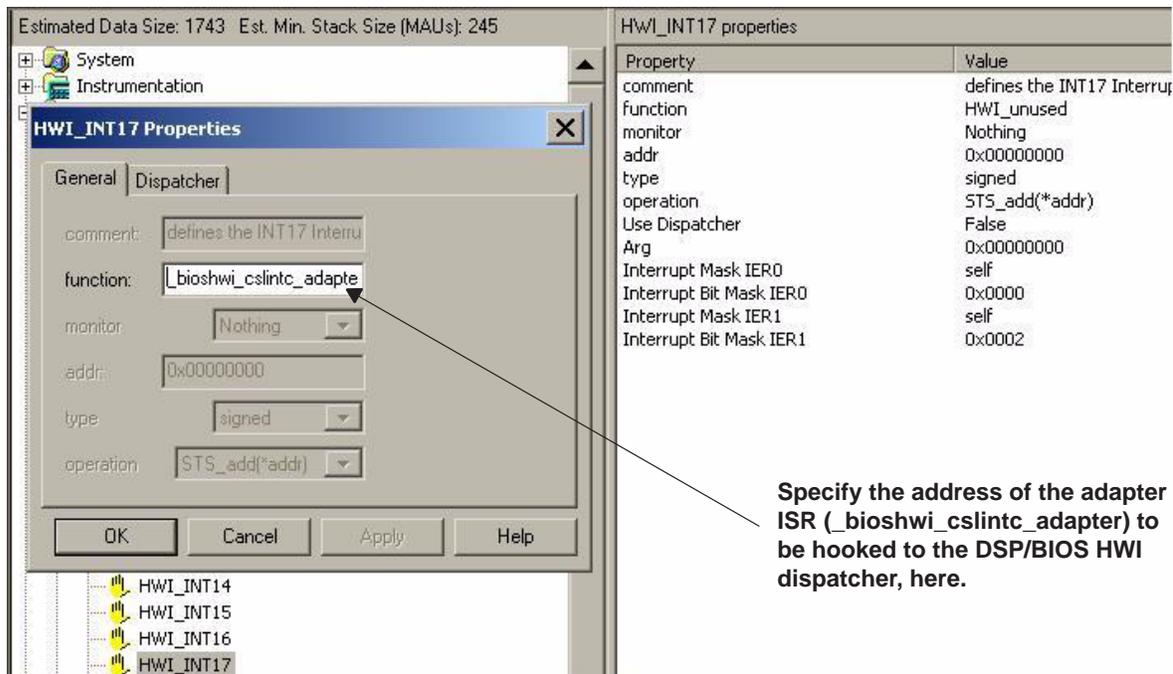


Figure A-3. Specifying the ISR Routine

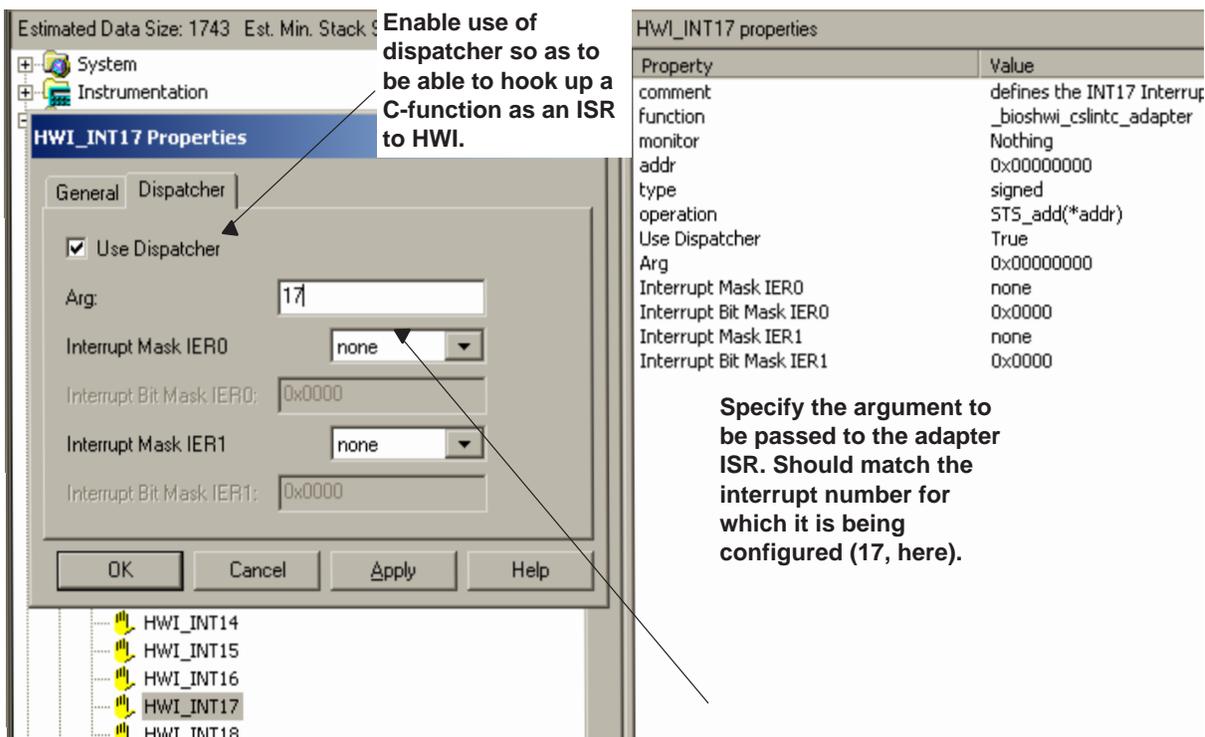


Figure A-4. Specifying the ISR Attributes

A.4.2 Dynamic Configuration: using 'HWI_dispatchPlug(...)'

To achieve the same dynamically, we use the 'HWI_dispatchPlug' API, as below:

```

/* hooking up the adapter to the cascaded-interrupts on the core */
{
    HWI_Attrs  attrs;

/* the ier{0,1}Mask members in the 'HWI_Attrs' structure are interrupt masks that the
DSP/BIOS dispatcher would apply before invoking the ISR. in our example, here, we have
chosen not to disable any interrupts */

    attrs.ier0mask = 0; /* do not disable */
    attrs.ier1mask = 0; /* any interrupt */
    attrs.arg      = (Arg)3;
    HWI_dispatchPlug(3, (Fxn)bioshwi_cslintc_adapter, &attrs);

    attrs.ier0mask = 0; /* do not disable */
    attrs.ier1mask = 0; /* any interrupt */
    attrs.arg      = (Arg)6;
    HWI_dispatchPlug(6, (Fxn)bioshwi_cslintc_adapter, &attrs);

    attrs.ier0mask = 0; /* do not disable */
    attrs.ier1mask = 0; /* any interrupt */
    attrs.arg      = (Arg)17;
    HWI_dispatchPlug(17, (Fxn)bioshwi_cslintc_adapter, &attrs);
}

```

A.5 Control Flow

1. The I2C peripheral conveys an interrupt to the L2.0 cascaded interrupt controller via interrupt line #15 of the controller
2. The L2.0 interrupt controller registers the interrupt, and in turn, conveys this interrupt to the CPU on core-interrupt line #3, routing it through the FIQ line.
3. CPU detects the interrupt and initiates the interruption sequence invoking the vector corresponding to interrupt #3. Since the DSP/BIOS HWI would have hooked its dispatcher to all the CPU-interrupts, the HWI dispatcher would get called.
4. The DSP/BIOS HWI Dispatcher ('HWI_F_dispatch') would execute necessary preamble code (like 'HWI_enter') and look for the ISR hooked to interrupt #3.
5. This would result in the execution of the Adapter ISR ('bioshwi_cslintc_adapter') since it is registered with the HWI dispatcher for interrupt 3.
6. The Adapter would invoke the CSL dispatcher ('CSL_intcDispatcher')
7. CSL dispatcher queries the L2.0 interrupt controller to find the interrupt-line on the controller that triggered the interrupt. It finds that it corresponds to the I2C interrupt.
8. The CSL dispatcher checks the event-handler record corresponding to the I2C interrupt in its internal array of event handler records. Finds the address to 'my_i2c_isr' in the record and the value of the argument to be passed to it.
9. The CSL dispatcher calls 'my_i2c_isr' with the corresponding argument
10. 'my_i2c_isr' services the interrupt before returning control to the CSL dispatcher
11. The CSL dispatcher acknowledges the servicing of the interrupt with the interrupt controller and returns control to the Adapter ISR – the acknowledgement would instruct the controller to process the next interrupt.
12. The Adapter ISR returns control to the DSP/BIOS HWI dispatcher (that had invoked it)
13. The HWI Dispatcher executes the post-amble code ('HWI_exit') and retraces the path back to interrupted context in the user's application.

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.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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