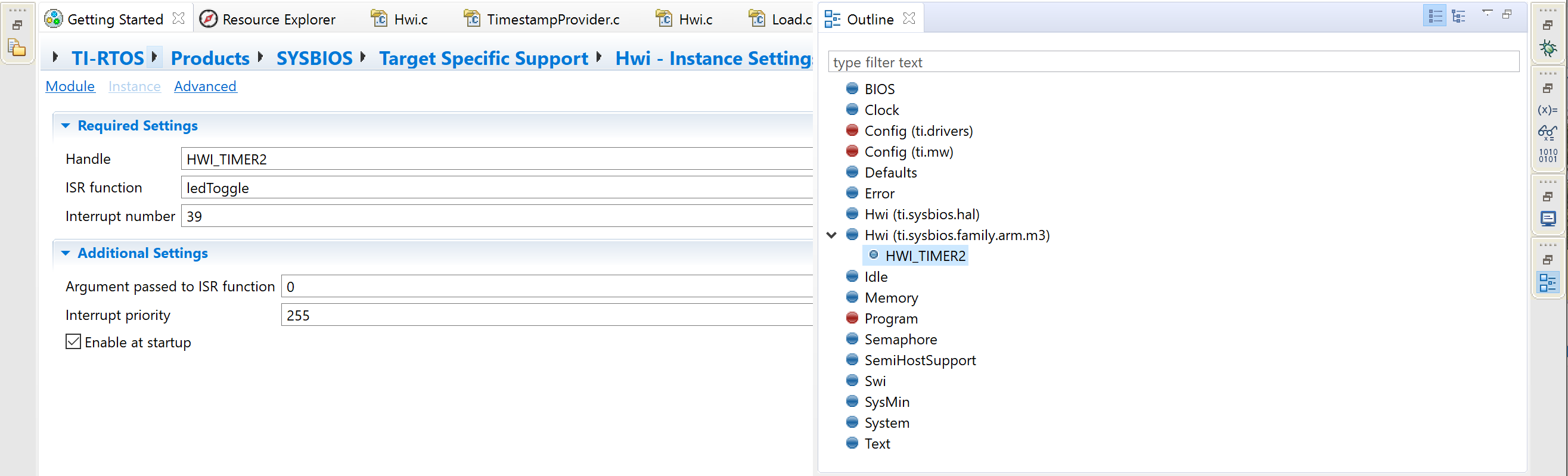
**Date Submitted: 11/10/18 2:52 PM**

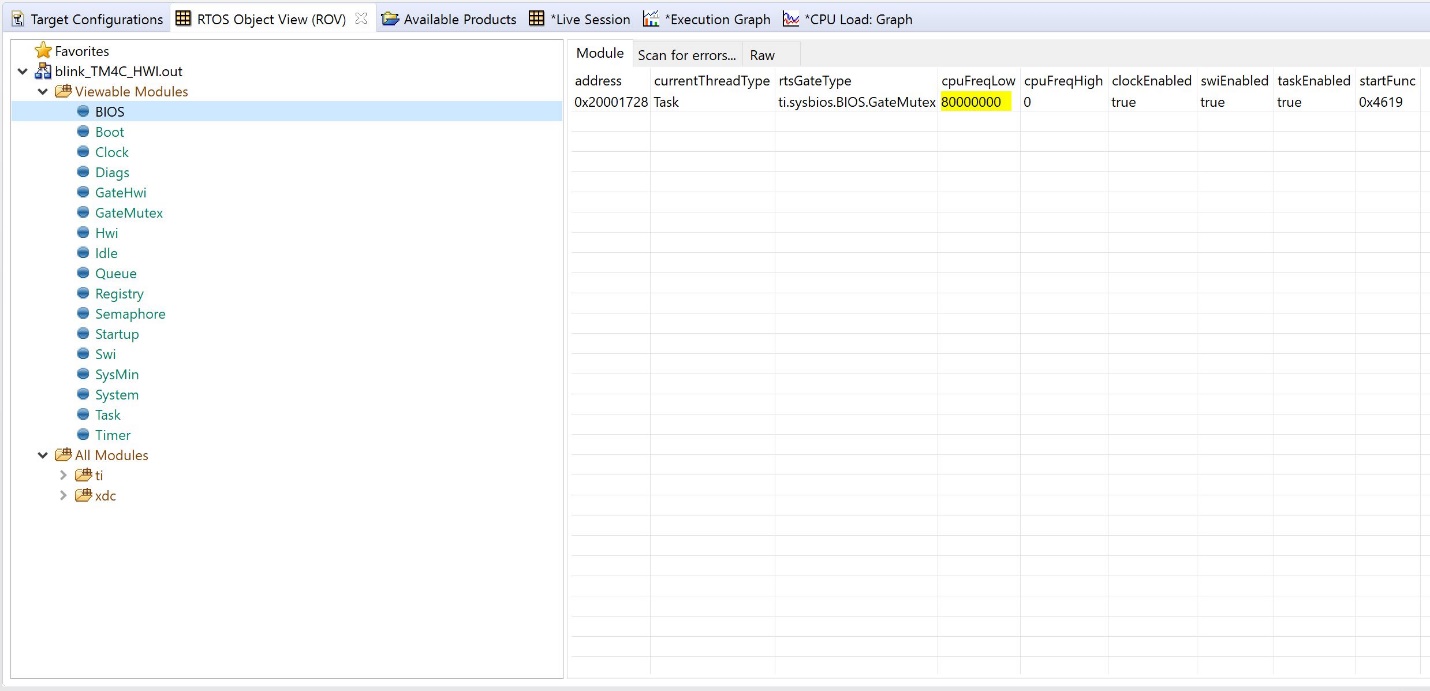
**Task 01:**

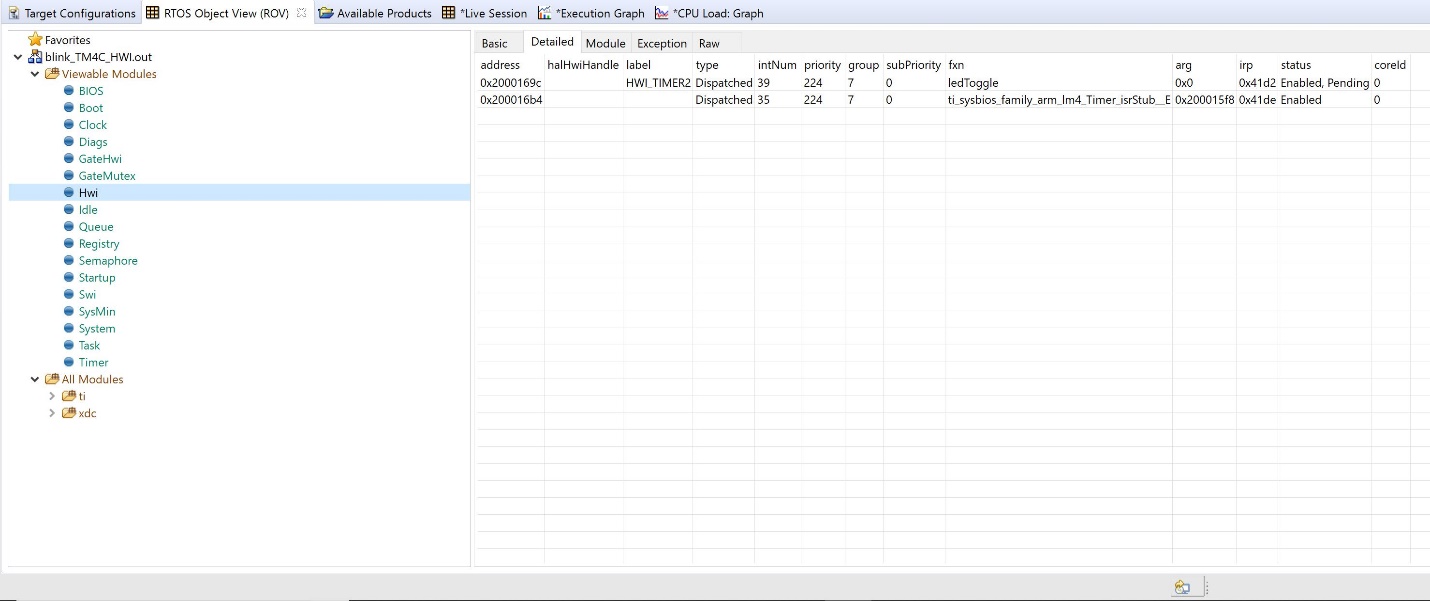
Youtube Link: <https://www.youtube.com/watch?v=NPuNQk14Nmg>

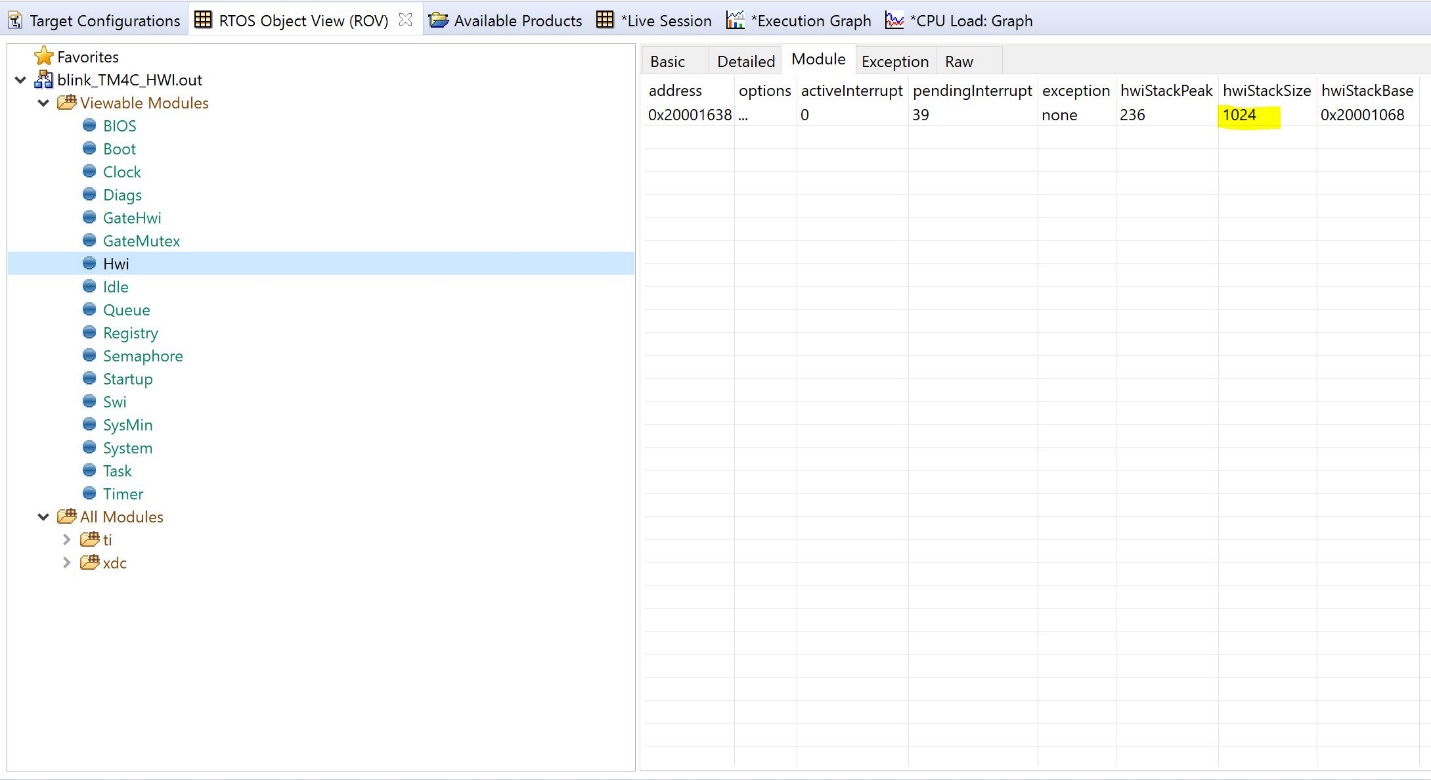


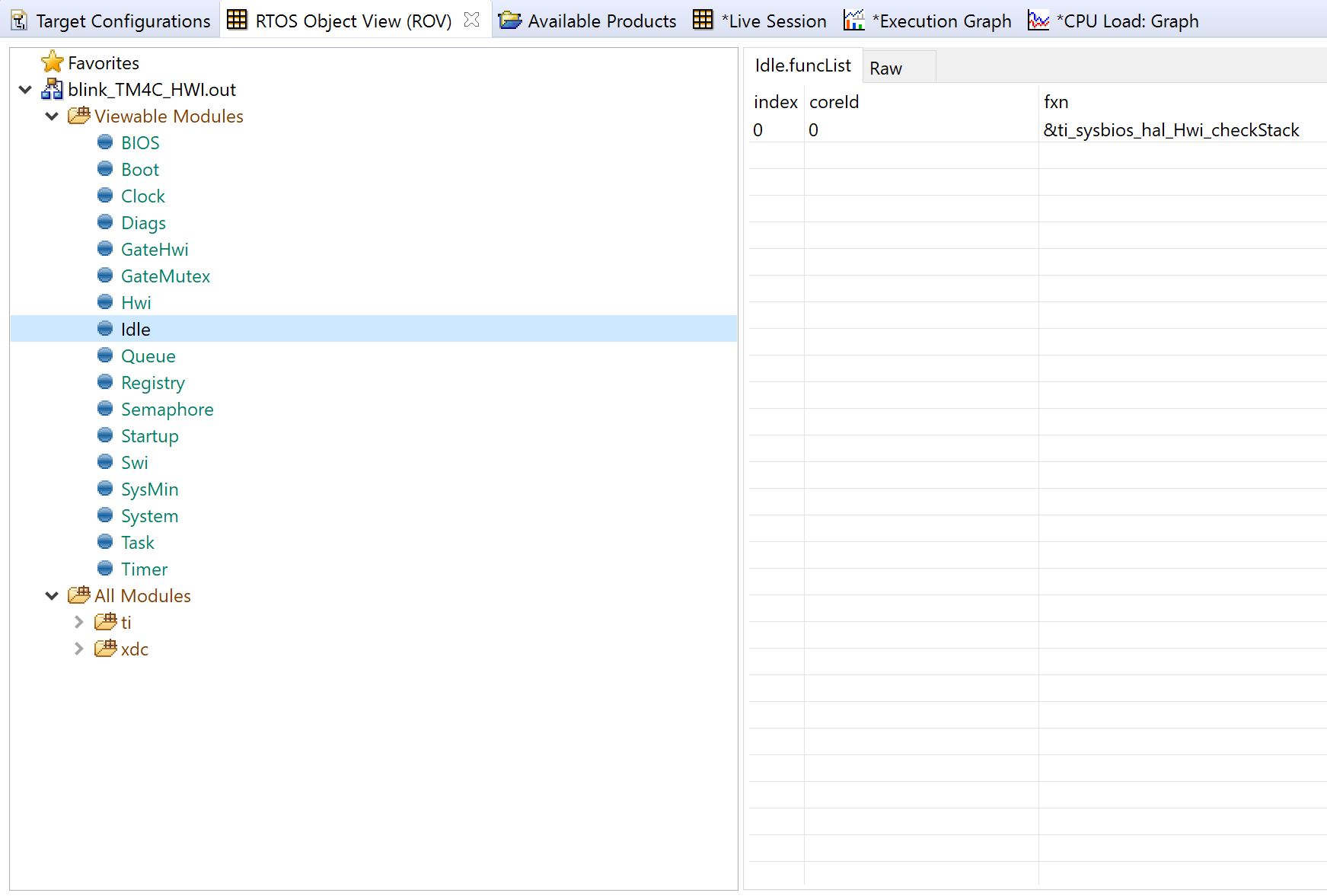
Modified empty.cfg file adding the HWI\_TIMER2 instance.

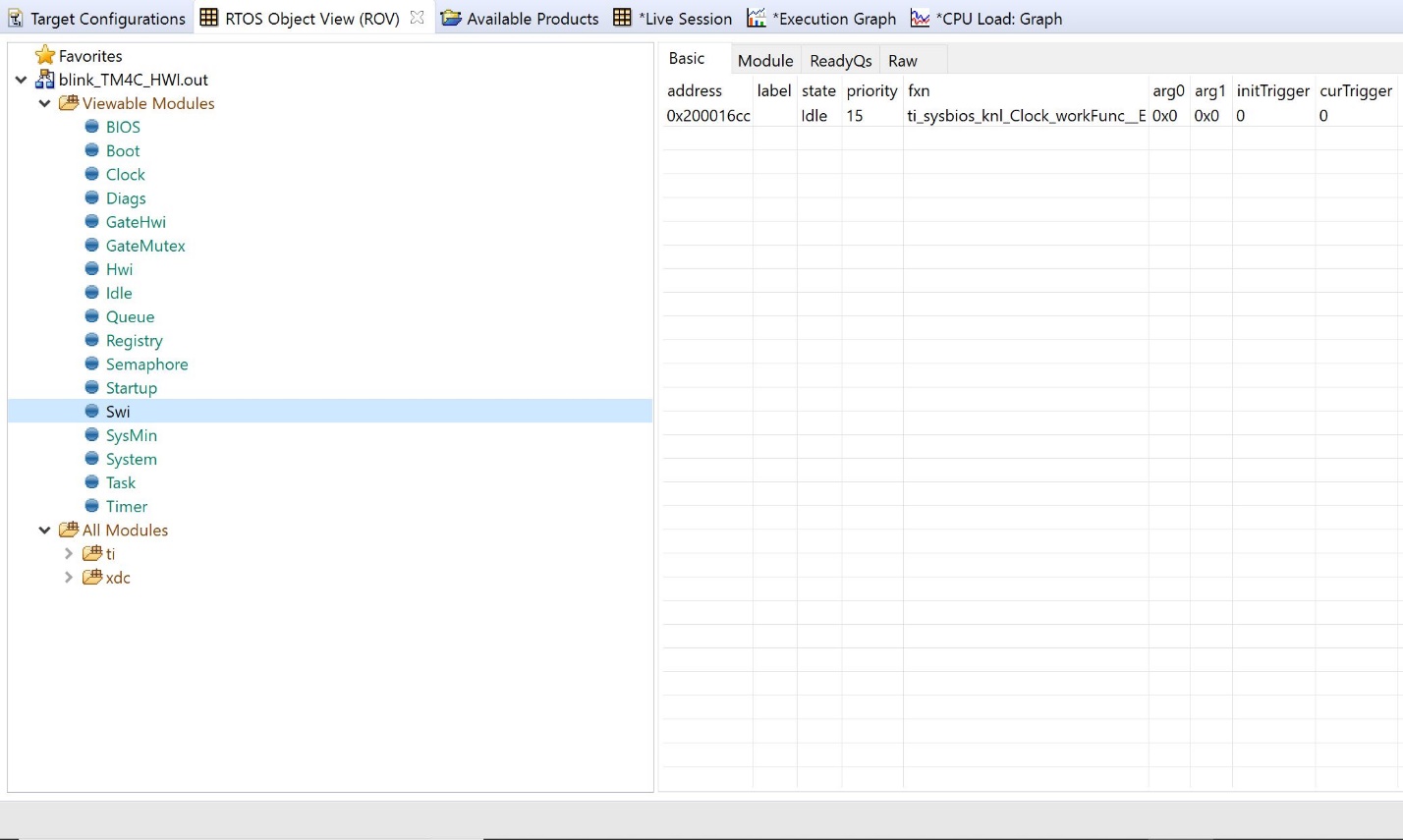
Below are the variables displayed by ROV:

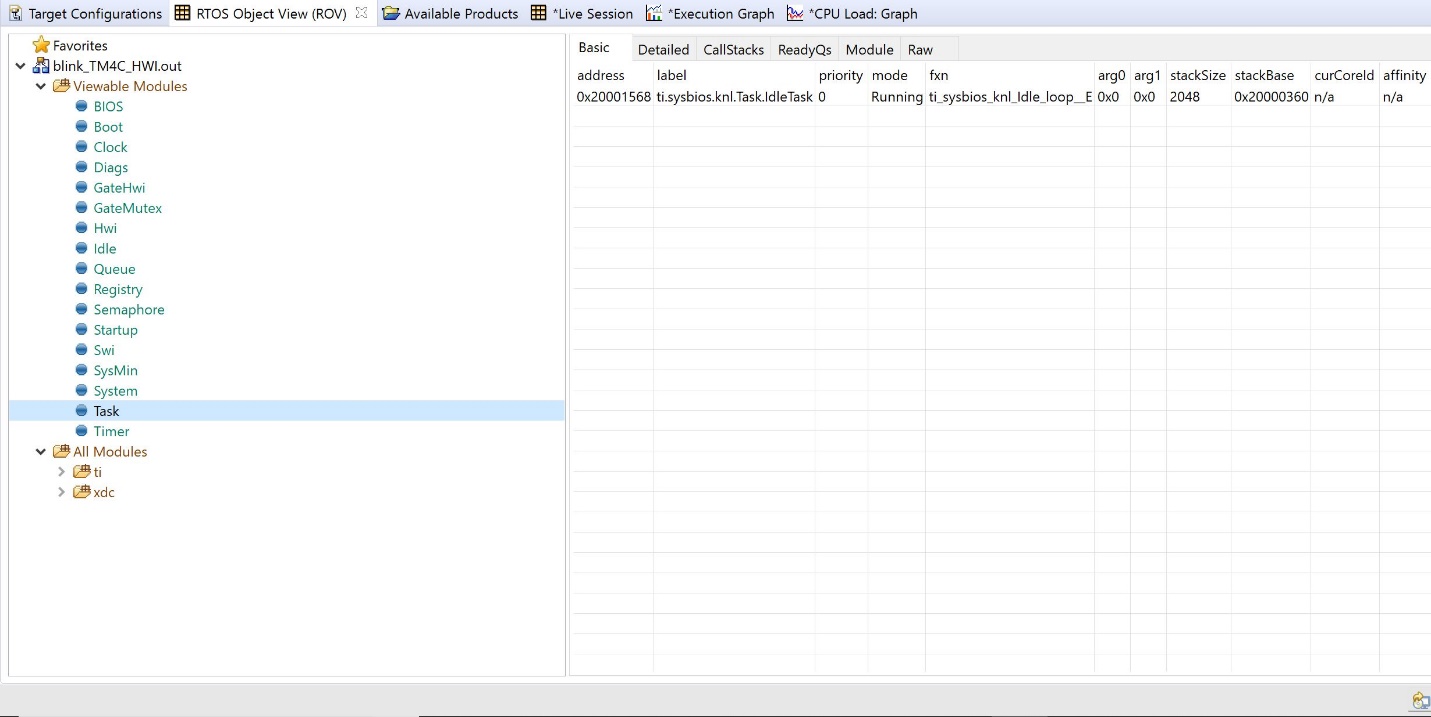


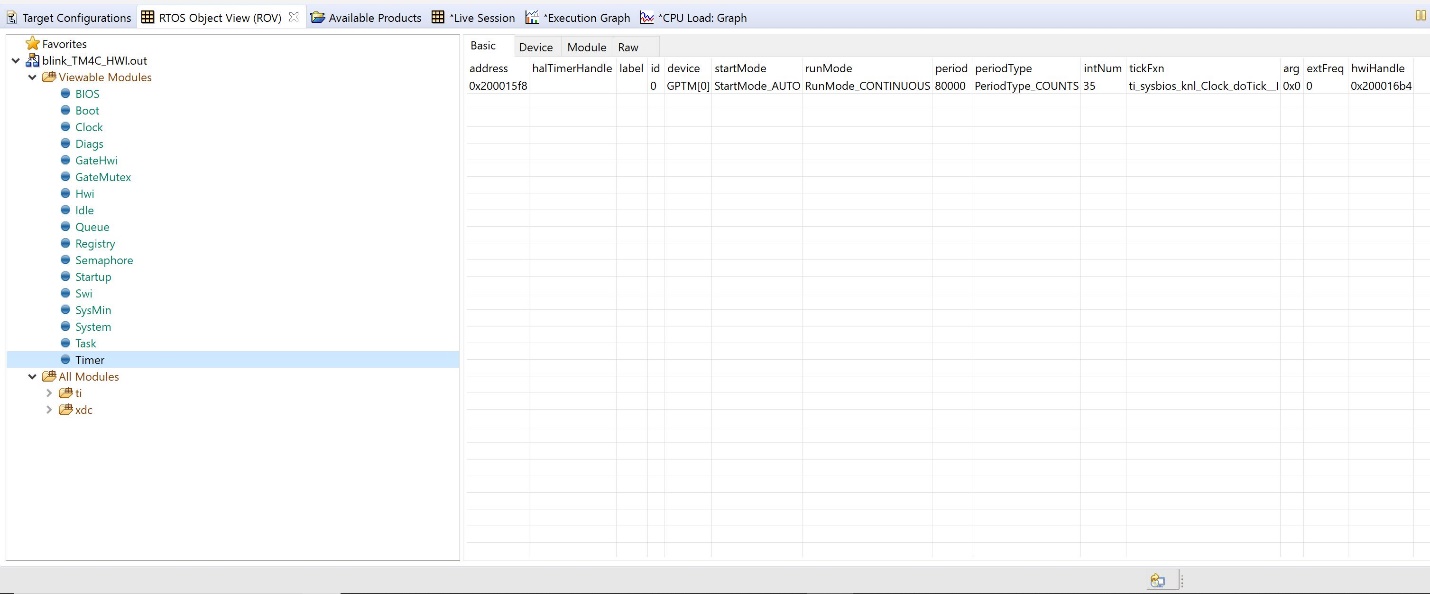












There is not an execution graph to display. Due to the fact that in Lab 5, we aren’t logging HWIs and all the information is being logged in the HWI.

Modified Code:

/\*

\* Copyright (c) 2015 Texas Instruments Incorporated

\* All rights reserved.

\*

\* Redistribution and use in source and binary forms, with or without

\* modification, are permitted provided that the following conditions

\* are met:

\*

\* \* Redistributions of source code must retain the above copyright

\* notice, this list of conditions and the following disclaimer.

\*

\* \* Redistributions in binary form must reproduce the above copyright

\* notice, this list of conditions and the following disclaimer in the

\* documentation and/or other materials provided with the distribution.

\*

\* \* Neither the name of Texas Instruments Incorporated nor the names of

\* its contributors may be used to endorse or promote products derived

\* from this software without specific prior written permission.

\*

\* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"

\* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,

\* THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR

\* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR

\* CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,

\* EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,

\* PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;

\* OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,

\* WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR

\* OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,

\* EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

\*/

/\*

\* ======== Hwi.c ========

\*/

**#include** <xdc/std.h>

**#include** <xdc/runtime/Error.h>

**#include** <xdc/runtime/Memory.h>

**#include** <xdc/runtime/Startup.h>

**#include** <xdc/runtime/System.h>

**#include** <xdc/runtime/Log.h>

**#include** <xdc/runtime/Assert.h>

**#include** <ti/sysbios/interfaces/IHwi.h>

**#define** ti\_sysbios\_knl\_Task\_\_internalaccess

**#include** <ti/sysbios/knl/Task.h>

**#include** <ti/sysbios/knl/Swi.h>

**#include** <ti/sysbios/BIOS.h>

**#include** "package/internal/Hwi.xdc.h"

**extern** Char \***ti\_sysbios\_family\_xxx\_Hwi\_switchToIsrStack**();

**extern** Void **ti\_sysbios\_family\_xxx\_Hwi\_switchToTaskStack**(Char \*oldTaskSP);

**extern** Void **ti\_sysbios\_family\_xxx\_Hwi\_switchAndRunFunc**(Void (\*func)());

**extern** UInt32 ti\_sysbios\_family\_arm\_m3\_Hwi\_dispatchTable[];

**#define** Hwi\_switchToIsrStack ti\_sysbios\_family\_xxx\_Hwi\_switchToIsrStack

**#define** Hwi\_switchToTaskStack ti\_sysbios\_family\_xxx\_Hwi\_switchToTaskStack

**#define** Hwi\_switchAndRunFunc ti\_sysbios\_family\_xxx\_Hwi\_switchAndRunFunc

**#ifdef** ti\_sysbios\_family\_arm\_m3\_Hwi\_dispatcherTaskSupport\_\_D

**#define** TASK\_DISABLE Task\_disable

**#define** TASK\_RESTORE Task\_restoreHwi

**#else**

**#define** TASK\_DISABLE Hwi\_taskDisable

**#define** TASK\_RESTORE Hwi\_taskRestoreHwi

**#endif**

**#ifdef** ti\_sysbios\_family\_arm\_m3\_Hwi\_dispatcherSwiSupport\_\_D

**#define** SWI\_DISABLE Swi\_disable

**#define** SWI\_RESTORE Swi\_restoreHwi

**#else**

**#define** SWI\_DISABLE Hwi\_swiDisable

**#define** SWI\_RESTORE Hwi\_swiRestoreHwi

**#endif**

/\*

\* ======== Hwi\_Module\_startup ========

\*/

Int **Hwi\_Module\_startup**(Int phase)

{

**int** i;

/\*

\* perform the BIOS specific interrupt disable operation so

\* main is entered with interrupts disabled.

\*/

Hwi\_disable();

**#ifndef** ti\_sysbios\_hal\_Hwi\_DISABLE\_ALL\_HOOKS

**for** (i = 0; i < Hwi\_hooks.length; i++) {

**if** (Hwi\_hooks.elem[i].registerFxn != NULL) {

Hwi\_hooks.elem[i].registerFxn(i);

}

}

**#endif**

/\*

\* Initialize the pointer to the isrStack.

\*

\* The dispatcher's SP must be aligned on a long word boundary

\*/

Hwi\_module->isrStack =

(Char \*)(((UInt32)(Hwi\_module->isrStackBase) & 0xfffffff8) +

(UInt32)Hwi\_module->isrStackSize - 2 \* **sizeof**(Int));

Hwi\_module->taskSP = (Char \*)-1; /\* signal that we're executing \*/

/\* on the ISR stack \*/

**#ifdef** ti\_sysbios\_family\_arm\_m3\_Hwi\_FIX\_MSP432\_DISPATCH\_TABLE\_ADDRESS

/\* map dispatchTable's SRAM\_DATA address into its corresponding SRAM\_CODE memory region \*/

UInt32 dispatchTable = (UInt32)Hwi\_module->dispatchTable;

dispatchTable = (dispatchTable & 0x00fffff) | 0x01000000;

Hwi\_module->dispatchTable = (Void \*)dispatchTable;

**#endif**

**for** (i = 0; i < Hwi\_Object\_count(); i++) {

Hwi\_postInit(Hwi\_Object\_get(NULL, i), NULL);

}

**return** Startup\_DONE;

}

/\*

\* ======== Hwi\_Instance\_init ========

\*/

Int **Hwi\_Instance\_init**(Hwi\_Object \*hwi, Int intNum,

Hwi\_FuncPtr fxn, **const** Hwi\_Params \*params,

Error\_Block \*eb)

{

Int status;

hwi->intNum = intNum;

/\* check vector table entry for already in use vector \*/

**if** (\*((UInt32 \*)Hwi\_module->vectorTableBase + intNum) !=

(UInt32)Hwi\_nullIsrFunc) {

Error\_raise(eb, Hwi\_E\_alreadyDefined, intNum, 0);

**return** (1);

}

**#ifndef** ti\_sysbios\_hal\_Hwi\_DISABLE\_ALL\_HOOKS

**if** (Hwi\_hooks.length > 0) {

/\* Allocate environment space for each hook instance. \*/

hwi->hookEnv = Memory\_calloc(Hwi\_Object\_heap(),

Hwi\_hooks.length \* **sizeof**(Ptr), 0, eb);

**if** (hwi->hookEnv == NULL) {

**return** (2);

}

}

**#endif**

Hwi\_disableInterrupt(intNum);

hwi->fxn = fxn;

hwi->arg = params->arg;

/\*

\* the -1 sentinel priority is the default passed by hal Hwi\_create().

\* Translate it to 255, which is our default priority.

\*/

**if** (params->priority == -1) {

hwi->priority = 255;

}

**else** {

hwi->priority = params->priority;

}

/\*

\* hwi->irp field is overloaded during initialization

\* to reduce Hwi Object footprint

\*

\* for postInit(), encode irp with enableInt

\* and useDispatcher info.

\*/

hwi->irp = 0;

/\* encode 'enableInt' in bit 0 \*/

**if** (params->enableInt) {

hwi->irp = 0x1;

}

/\* encode 'useDispatcher' in bit 1 \*/

**if** (params->useDispatcher) {

hwi->irp |= 0x2;

}

status = Hwi\_postInit(hwi, eb);

**if** (Error\_check(eb)) {

**return** (3 + status);

}

**return** (0);

}

/\*

\* ======== Hwi\_postInit ========

\* Function to be called during module startup to complete the

\* initialization of any statically created or constructed Hwi.

\* returns (0) and clean 'eb' on success

\* returns 'eb' \*and\* 'n' for number of successful createFxn() calls iff

\* one of the createFxn() calls fails

\*/

Int **Hwi\_postInit** (Hwi\_Object \*hwi, Error\_Block \*eb)

{

UInt intNum;

**#ifndef** ti\_sysbios\_hal\_Hwi\_DISABLE\_ALL\_HOOKS

Int i;

**for** (i = 0; i < Hwi\_hooks.length; i++) {

hwi->hookEnv[i] = (Ptr)0;

**if** (Hwi\_hooks.elem[i].createFxn != NULL) {

Hwi\_hooks.elem[i].createFxn((IHwi\_Handle)hwi, eb);

**if** (Error\_check(eb)) {

**return** (i);

}

}

}

**#endif**

/\*

\* Bypass dispatcher for zero-latency interrupts

\* and interrupts with useDispatcher == FALSE

\*/

/\* save intNum for use by Hwi\_enableInterrupt \*/

intNum = hwi->intNum;

**if** (((hwi->irp & 0x2) == 0) ||

(hwi->priority < Hwi\_disablePriority)) {

Hwi\_plug(hwi->intNum, (Void \*)(UArg)hwi->fxn);

/\* encode useDispatcher == FALSE as a negative intNum \*/

hwi->intNum = 0 - hwi->intNum;

}

**else** {

**if** (Hwi\_numSparseInterrupts) {

Int i;

UInt32 \*sparseInterruptTableEntry = Hwi\_module->dispatchTable;

Char \*vectorPtr;

Bool found = FALSE;

/\* find an unused sparseTableEntry \*/

**for** (i = 0; i < (Int)Hwi\_numSparseInterrupts; i++) {

**if** (sparseInterruptTableEntry[2] == 0) {

found = TRUE;

**break**;

}

**else** {

sparseInterruptTableEntry += 3;

}

}

**if** (found) {

/\* point it to the Hwi object \*/

sparseInterruptTableEntry[2] = (UInt32)hwi;

/\* plug the vector table with the sparseTable entry \*/

vectorPtr = (Char \*)sparseInterruptTableEntry;

vectorPtr += 1; /\* make it a thumb func vector \*/

Hwi\_plug(hwi->intNum, (Void \*)vectorPtr);

}

**else** {

Error\_raise(eb, Hwi\_E\_hwiLimitExceeded, 0, 0);

**#ifndef** ti\_sysbios\_hal\_Hwi\_DISABLE\_ALL\_HOOKS

**return** (Hwi\_hooks.length); /\* unwind all Hwi\_hooks \*/

**#else**

**return** (0);

**#endif**

}

}

**else** {

Hwi\_Object \*\*dispatchTable = (Hwi\_Object \*\*)Hwi\_module->dispatchTable;

dispatchTable[intNum] = hwi;

Hwi\_plug(intNum, (Void \*)(UArg)Hwi\_dispatch);

}

}

Hwi\_setPriority(intNum, hwi->priority);

**if** ((hwi->irp & 0x1) != 0) {

Hwi\_enableInterrupt(intNum);

}

hwi->irp = 0;

**return** (0);

}

/\*

\* ======== Hwi\_Instance\_finalize ========

\*/

Void **Hwi\_Instance\_finalize**(Hwi\_Object \*hwi, Int status)

{

UInt intNum;

**if** (status == 1) { /\* failed Hwi\_create \*/

**return**;

}

**#ifndef** ti\_sysbios\_hal\_Hwi\_DISABLE\_ALL\_HOOKS

**if** (Hwi\_hooks.length > 0) {

Int i, cnt;

**if** (status == 0) {

cnt = Hwi\_hooks.length;

}

**else** {

cnt = status - 3;

}

**for** (i = 0; i < cnt; i++) {

**if** (Hwi\_hooks.elem[i].deleteFxn != NULL) {

Hwi\_hooks.elem[i].deleteFxn((IHwi\_Handle)hwi);

}

}

Memory\_free(Hwi\_Object\_heap(), hwi->hookEnv,

Hwi\_hooks.length \* **sizeof**(Ptr));

}

**#endif**

intNum = hwi->intNum;

Hwi\_disableInterrupt(intNum);

Hwi\_plug(intNum, (Void \*)(UArg)Hwi\_nullIsrFunc);

**if** (Hwi\_numSparseInterrupts) {

Int i;

UInt32 \*sparseInterruptTableEntry = Hwi\_module->dispatchTable;

**for** (i = 0; i < (Int)Hwi\_numSparseInterrupts; i++) {

**if** (sparseInterruptTableEntry[2] == (UInt32)hwi) {

sparseInterruptTableEntry[2] = 0;

**break**;

}

**else** {

sparseInterruptTableEntry += 3;

}

}

}

**else** {

Hwi\_Object \*\*dispatchTable = (Hwi\_Object \*\*)Hwi\_module->dispatchTable;

dispatchTable[intNum] = 0;

}

}

/\*

\* ======== Hwi\_initNVIC ========

\* run as a Startup.firstFxn

\*/

**#ifdef** \_\_IAR\_SYSTEMS\_ICC\_\_

**extern** UInt32 \_\_vector\_table[];

**#define** Hwi\_resetVectors \_\_vector\_table

**#else**

**extern** **const** UInt32 ti\_sysbios\_family\_arm\_m3\_Hwi\_resetVectors[];

**#define** Hwi\_resetVectors ti\_sysbios\_family\_arm\_m3\_Hwi\_resetVectors

**#endif**

Void **Hwi\_initNVIC**()

{

UInt intNum;

UInt32 \*ramVectors;

/\* configure Vector Table Offset Register \*/

Hwi\_nvic.VTOR = (UInt32)Hwi\_module->vectorTableBase;

/\* copy ROM vector table contents to RAM vector table \*/

**if** (Hwi\_nvic.VTOR != (UInt32)Hwi\_resetVectors) {

ramVectors = Hwi\_module->vectorTableBase;

**for** (intNum = 0; intNum < Hwi\_NUM\_INTERRUPTS; intNum++) {

**if** (intNum < 15) {

ramVectors[intNum] = Hwi\_resetVectors[intNum];

}

**else** {

ramVectors[intNum] = (UInt32)Hwi\_nullIsrFunc;

}

}

}

/\* Set the configured PRIGROUP value \*/

Hwi\_nvic.AIRCR = (Hwi\_nvic.AIRCR & 0xffff00ff) + (Hwi\_priGroup << 8) + 0x05fa0000;

/\* set pendSV interrupt priority to Hwi\_disablePriority \*/

Hwi\_nvic.SHPR[10] = Hwi\_disablePriority;

/\* set CCR per user's preference \*/

Hwi\_nvic.CCR = Hwi\_ccr;

**#if** (defined(\_\_TI\_VFP\_SUPPORT\_\_) || \

(defined(\_\_VFP\_FP\_\_) && !defined(\_\_SOFTFP\_\_)) || \

defined(\_\_ARMVFP\_\_))

/\* disable lazy stacking mode fp indications in control register \*/

Hwi\_nvic.FPCCR &= ~0xc0000000; /\* clear ASPEN and LSPEN bits \*/

**#endif**

}

/\*

\* ======== cc26xxRomInitNVIC ========

\* Fix for SDOCM00114681: broken Hwi\_initNVIC() function.

\* Installed rather than Hwi.initNVIC for CC26xx ROM build

\* when Hwi.resetVectorAddress is not 0x00000000.

\*/

Void **Hwi\_cc26xxRomInitNVIC**()

{

UInt intNum;

UInt32 \*ramVectors;

/\* first, leverage Hwi\_initNVIC() to save .text in flash \*/

Hwi\_initNVIC();

/\* then fix the first 15 vectors \*/

ramVectors = Hwi\_module->vectorTableBase;

**for** (intNum = 0; intNum < 15; intNum++) {

ramVectors[intNum] = Hwi\_resetVectors[intNum];

}

}

/\*

\* ======== Hwi\_startup ========

\*/

Void **Hwi\_startup**()

{

Hwi\_enable();

}

**#ifdef** \_\_ti\_\_

/\*

\* ======== Hwi\_disableFxn ========

\*/

UInt Hwi\_disableFxn()

{

**return** (\_set\_interrupt\_priority(Hwi\_disablePriority));

}

/\*

\* ======== Hwi\_enableFxn ========

\*/

UInt Hwi\_enableFxn()

{

**return** \_set\_interrupt\_priority(0);

}

/\*

\* ======== Hwi\_restoreFxn ========

\*/

Void Hwi\_restoreFxn(UInt key)

{

\_set\_interrupt\_priority(key);

}

**#else**

/\*

\* ======== Hwi\_disableFxn ========

\*/

UInt **Hwi\_disableFxn**()

{

UInt key;

**#if** defined(\_\_IAR\_SYSTEMS\_ICC\_\_)

**asm** **volatile** (

**#else** /\* !\_\_IAR\_SYSTEMS\_ICC\_\_ \*/

**\_\_asm\_\_** **\_\_volatile\_\_** (

**#endif**

"mrs %0, basepri\n\t"

"msr basepri, %1"

: "=&r" (key)

: "r" (ti\_sysbios\_family\_arm\_m3\_Hwi\_disablePriority)

);

**return** key;

}

/\*

\* ======== Hwi\_restoreFxn ========

\*/

Void **Hwi\_restoreFxn**(UInt key)

{

**#if** defined(\_\_IAR\_SYSTEMS\_ICC\_\_)

**asm** **volatile** (

**#else** /\* !\_\_IAR\_SYSTEMS\_ICC\_\_ \*/

**\_\_asm\_\_** **\_\_volatile\_\_** (

**#endif**

"msr basepri, %0"

:: "r" (key)

);

}

/\*

\* ======== Hwi\_enableFxn ========

\*/

UInt **Hwi\_enableFxn**()

{

UInt key;

**#if** defined(\_\_IAR\_SYSTEMS\_ICC\_\_)

**asm** **volatile** (

**#else** /\* !\_\_IAR\_SYSTEMS\_ICC\_\_ \*/

**\_\_asm\_\_** **\_\_volatile\_\_** (

**#endif**

"movw r12, #0\n\t"

"mrs %0, basepri\n\t"

"msr basepri, r12"

: "=r" (key)

:: "r12"

);

**return** key;

}

**#endif**

/\*

\* ======== Hwi\_disableInterrupt ========

\*/

UInt **Hwi\_disableInterrupt**(UInt intNum)

{

UInt oldEnableState, index, mask;

**if** (intNum >= 16) {

index = (intNum-16) >> 5;

mask = 1 << ((intNum-16) & 0x1f);

oldEnableState = Hwi\_nvic.ISER[index] & mask;

Hwi\_nvic.ICER[index] = mask;

}

**else** **if** (intNum == 15) {

oldEnableState = Hwi\_nvic.STCSR & 0x00000002;

Hwi\_nvic.STCSR &= ~0x00000002; /\* disable SysTick Int \*/

}

**else** {

oldEnableState = 0;

}

/\* suppress inlining of this function \*/

**#if** (defined(\_\_ti\_\_) || defined(\_\_IAR\_SYSTEMS\_ICC\_\_))

**asm** (" nop");

**#else**

**\_\_asm\_\_** (" nop");

**#endif**

**return** oldEnableState;

}

/\*

\* ======== Hwi\_enableInterrupt ========

\*/

UInt **Hwi\_enableInterrupt**(UInt intNum)

{

UInt oldEnableState, index, mask;

**if** (intNum >= 16) {

index = (intNum-16) >> 5;

mask = 1 << ((intNum-16) & 0x1f);

oldEnableState = Hwi\_nvic.ISER[index] & mask;

Hwi\_nvic.ISER[index] = mask;

}

**else** **if** (intNum == 15) {

oldEnableState = Hwi\_nvic.STCSR & 0x00000002;

Hwi\_nvic.STCSR |= 0x00000002; /\* enable SysTick Int \*/

}

**else** {

oldEnableState = 0;

}

/\* suppress inlining of this function \*/

**#if** (defined(\_\_ti\_\_) || defined(\_\_IAR\_SYSTEMS\_ICC\_\_))

**asm** (" nop");

**#else**

**\_\_asm\_\_** (" nop");

**#endif**

**return** oldEnableState;

}

/\*

\* ======== Hwi\_restoreInterrupt ========

\*/

Void **Hwi\_restoreInterrupt**(UInt intNum, UInt key)

{

UInt index, mask;

**if** (intNum >= 16) {

index = (intNum-16) >> 5;

mask = 1 << ((intNum-16) & 0x1f);

**if** (key) {

Hwi\_nvic.ISER[index] = mask;

}

**else** {

Hwi\_nvic.ICER[index] = mask;

}

}

**else** **if** (intNum == 15) {

**if** (key) {

Hwi\_nvic.STCSR |= 0x00000002; /\* enable SysTick Int \*/

}

**else** {

Hwi\_nvic.STCSR &= ~0x00000002; /\* disable SysTick Int \*/

}

}

/\* suppress inlining of this function \*/

**#if** (defined(\_\_ti\_\_) || defined(\_\_IAR\_SYSTEMS\_ICC\_\_))

**asm** (" nop");

**#else**

**\_\_asm\_\_** (" nop");

**#endif**

}

/\*

\* ======== Hwi\_clearInterrupt ========

\*/

Void **Hwi\_clearInterrupt**(UInt intNum)

{

UInt index, mask;

**if** (intNum >= 16) {

index = (intNum-16) >> 5;

mask = 1 << ((intNum-16) & 0x1f);

Hwi\_nvic.ICPR[index] = mask;

}

**else** {

}

}

/\*

\* ======== Hwi\_getHandle ========

\*/

Hwi\_Handle **Hwi\_getHandle**(UInt intNum)

{

UInt32 \*func;

func = (UInt32 \*)Hwi\_module->vectorTableBase + intNum;

**if** (Hwi\_numSparseInterrupts) {

**if** (\*func != (UInt32)Hwi\_nullIsrFunc) {

Char \*vectorPtr = (Char \*)func;

vectorPtr -= 1;

UInt32 \*sparseTableEntry = (UInt32 \*)vectorPtr;

/\* Hwi handle is the 3rd word in the sparseTableEntry array \*/

**return** ((Hwi\_Handle)sparseTableEntry[2]);

}

**else** {

**return** (NULL);

}

}

**else** {

Hwi\_Object \*\*dispatchTable = (Hwi\_Object \*\*)Hwi\_module->dispatchTable;

**return** (dispatchTable[intNum]);

}

}

/\*

\* ======== Hwi\_plug ========

\*/

Void **Hwi\_plug**(UInt intNum, Void \*fxn)

{

UInt32 \*func;

func = (UInt32 \*)Hwi\_module->vectorTableBase + intNum;

/\* guard against writing to static const vector table in flash \*/

**if** (\*func != (UInt32)fxn) {

\*func = (UInt32)fxn;

}

}

/\*

\* ======== switchFromBootStack ========

\* Indicate that we are leaving the boot stack and

\* are about to switch to a task stack.

\*

\* Must be (and is) called with interrupts disabled from

\* Task\_startup().

\*/

Void **Hwi\_switchFromBootStack**()

{

/\* split thread and handler stacks \*/

Hwi\_initStacks(Hwi\_module->isrStack);

Hwi\_module->taskSP = 0;

}

/\*

\* ======== Hwi\_getStackInfo ========

\* Used to get Hwi stack usage info.

\*/

Bool **Hwi\_getStackInfo**(Hwi\_StackInfo \*stkInfo, Bool computeStackDepth)

{

Char \*isrSP;

Bool stackOverflow;

/\* Copy the stack base address and size \*/

stkInfo->hwiStackSize = Hwi\_module->isrStackSize;

stkInfo->hwiStackBase = Hwi\_module->isrStackBase;

isrSP = stkInfo->hwiStackBase;

/\* Check for stack overflow \*/

stackOverflow = (\*isrSP != (Char)0xbe ? TRUE : FALSE);

**if** (computeStackDepth && !stackOverflow) {

/\* Compute stack depth \*/

**do** {

} **while**(\*isrSP++ == (Char)0xbe);

stkInfo->hwiStackPeak = stkInfo->hwiStackSize - (SizeT)(--isrSP - (Char \*)stkInfo->hwiStackBase);

}

**else** {

stkInfo->hwiStackPeak = 0;

}

**return** stackOverflow;

}

/\*

\* ======== Hwi\_setPriority ========

\* Set an interrupt's priority.

\*

\* @param(intNum) ID of interrupt

\* @param(priority) priority

\*/

Void **Hwi\_setPriority**(UInt intNum, UInt priority)

{

/\* User interrupt (id >= 16) priorities are set in the IPR registers \*/

**if** (intNum >= 16) {

Hwi\_nvic.IPR[intNum-16] = priority;

}

/\* System interrupt (id >= 4) priorities are set in the SHPR registers \*/

**else** **if** (intNum >= 4) {

Hwi\_nvic.SHPR[intNum-4] = priority;

}

/\* System interrupts (id < 4) priorities are fixed in hardware \*/

}

/\*

\* ======== Hwi\_reconfig ========

\* Reconfigure a dispatched interrupt.

\*/

Void **Hwi\_reconfig**(Hwi\_Object \*hwi, Hwi\_FuncPtr fxn, **const** Hwi\_Params \*params)

{

UInt intNum;

/\* compensate for encoded intNum \*/

**if** (hwi->intNum < 0) {

intNum = 0 - hwi->intNum;

}

**else** {

intNum = hwi->intNum;

}

Hwi\_disableInterrupt(intNum);

hwi->fxn = fxn;

hwi->arg = params->arg;

/\*

\* the -1 sentinel priority is the default passed by hal Hwi\_create().

\* Translate it to 255, which is our default priority.

\*/

**if** (params->priority == -1) {

hwi->priority = 255;

}

**else** {

hwi->priority = params->priority;

}

Hwi\_setPriority(intNum, hwi->priority);

**if** (params->enableInt) {

Hwi\_enableInterrupt(intNum);

}

}

/\*

\* ======== Hwi\_getfunc ========

\*/

Hwi\_FuncPtr **Hwi\_getFunc**(Hwi\_Object \*hwi, UArg \*arg)

{

\*arg = hwi->arg;

**return** (hwi->fxn);

}

/\*

\* ======== Hwi\_setfunc ========

\*/

Void **Hwi\_setFunc**(Hwi\_Object \*hwi, Hwi\_FuncPtr fxn, UArg arg)

{

hwi->fxn = fxn;

hwi->arg = arg;

}

/\*

\* ======== Hwi\_getIrp ========

\*/

Hwi\_Irp **Hwi\_getIrp**(Hwi\_Object \*hwi)

{

**return** (hwi->irp);

}

/\*

\* ======== Hwi\_getHookContext ========

\*/

Ptr **Hwi\_getHookContext**(Hwi\_Object \*hwi, Int id)

{

**return** (hwi->hookEnv[id]);

}

/\*

\* ======== Hwi\_setHookContext ========

\*/

Void **Hwi\_setHookContext**(Hwi\_Object \*hwi, Int id, Ptr hookContext)

{

hwi->hookEnv[id] = hookContext;

}

/\*

\* ======== Hwi\_post ========

\*/

Void **Hwi\_post**(UInt intNum)

{

**if** (intNum >= 16) {

Hwi\_nvic.STI = intNum - 16;

}

}

/\*

\* ======== Hwi\_excSetBuffers ========

\*/

Void **Hwi\_excSetBuffers**(Ptr excContextBuffer, Ptr excStackBuffer)

{

UInt coreId = 0;

Hwi\_module->excContext[coreId] = excContextBuffer;

Hwi\_module->excStack[coreId] = excStackBuffer;

}

/\*

\* ======== Hwi\_excHandler ========

\*/

Void **Hwi\_excHandler**(UInt \*excStack, UInt lr)

{

Hwi\_module->excActive[0] = TRUE;

/\* spin here if no exception handler is plugged \*/

**while** (Hwi\_excHandlerFunc == NULL) {

;

}

Hwi\_excHandlerFunc(excStack, lr);

}

/\*

\* ======== Hwi\_excHandlerMin ========

\*/

Void **Hwi\_excHandlerMin**(UInt \*excStack, UInt lr)

{

Hwi\_ExcContext excContext;

UInt excNum;

UInt coreId = 0;

Hwi\_module->excActive[coreId] = TRUE;

**if** (Hwi\_module->excContext[coreId] == NULL) {

Hwi\_module->excContext[coreId] = &excContext;

}

Hwi\_excFillContext(excStack);

/\* Force MAIN threadtype so hooks can safely call System\_printf \*/

BIOS\_setThreadType(BIOS\_ThreadType\_Main);

/\* Call user's exception hook \*/

**if** (Hwi\_excHookFuncs[coreId] != NULL) {

Hwi\_excHookFuncs[coreId](Hwi\_module->excContext[coreId]);

}

excNum = Hwi\_nvic.ICSR & 0xff;

/\* distinguish between an interrupt and an exception \*/

**if** ( excNum < 15 ) {

Error\_raise(0, Hwi\_E\_exception, excNum, excStack[14]);

}

**else** {

Error\_raise(0, Hwi\_E\_noIsr, excNum, excStack[14]);

}

}

/\*

\* ======== Hwi\_excHandlerMax ========

\*/

Void **Hwi\_excHandlerMax**(UInt \*excStack, UInt lr)

{

Hwi\_ExcContext excContext;

UInt excNum;

UInt coreId = 0;

Hwi\_module->excActive[coreId] = TRUE;

**if** (Hwi\_module->excContext[coreId] == NULL) {

Hwi\_module->excContext[coreId] = &excContext;

}

Hwi\_excFillContext(excStack);

/\* Force MAIN threadtype so we can safely call System\_printf \*/

BIOS\_setThreadType(BIOS\_ThreadType\_Main);

Hwi\_disable();

excNum = Hwi\_nvic.ICSR & 0xff;

**switch** (excNum) {

**case** 2:

Hwi\_excNmi(excStack); /\* NMI \*/

**break**;

**case** 3:

Hwi\_excHardFault(excStack); /\* Hard Fault \*/

**break**;

**case** 4:

Hwi\_excMemFault(excStack); /\* Mem Fault \*/

**break**;

**case** 5:

Hwi\_excBusFault(excStack); /\* Bus Fault \*/

**break**;

**case** 6:

Hwi\_excUsageFault(excStack);/\* Usage Fault \*/

**break**;

**case** 11:

Hwi\_excSvCall(excStack); /\* SVCall \*/

**break**;

**case** 12:

Hwi\_excDebugMon(excStack); /\* Debug Mon \*/

**break**;

**case** 7:

**case** 8:

**case** 9:

**case** 10:

**case** 13:

Hwi\_excReserved(excStack, excNum); /\* reserved \*/

**break**;

**default**:

Hwi\_excNoIsr(excStack, excNum); /\* no ISR \*/

**break**;

}

Hwi\_excDumpRegs(lr);

/\* Call user's exception hook \*/

**if** (Hwi\_excHookFuncs[coreId] != NULL) {

Hwi\_excHookFuncs[coreId](Hwi\_module->excContext[coreId]);

}

System\_printf("Terminating execution...\n");

Hwi\_enable();

BIOS\_exit(0);

}

/\*

\* ======== Hwi\_excFillContext ========

\*/

Void **Hwi\_excFillContext**(UInt \*excStack)

{

Hwi\_ExcContext \*excContext;

Char \*stack = NULL;

SizeT stackSize = 0;

UInt coreId = 0;

excContext = Hwi\_module->excContext[coreId];

excContext->threadType = BIOS\_getThreadType();

/\* copy registers from stack to excContext \*/

excContext->r0 = (Ptr)excStack[8]; /\* r0 \*/

excContext->r1 = (Ptr)excStack[9]; /\* r1 \*/

excContext->r2 = (Ptr)excStack[10]; /\* r2 \*/

excContext->r3 = (Ptr)excStack[11]; /\* r3 \*/

excContext->r4 = (Ptr)excStack[0]; /\* r4 \*/

excContext->r5 = (Ptr)excStack[1]; /\* r5 \*/

excContext->r6 = (Ptr)excStack[2]; /\* r6 \*/

excContext->r7 = (Ptr)excStack[3]; /\* r7 \*/

excContext->r8 = (Ptr)excStack[4]; /\* r8 \*/

excContext->r9 = (Ptr)excStack[5]; /\* r9 \*/

excContext->r10 = (Ptr)excStack[6]; /\* r10 \*/

excContext->r11 = (Ptr)excStack[7]; /\* r11 \*/

excContext->r12 = (Ptr)excStack[12]; /\* r12 \*/

excContext->sp = (Ptr)(UInt32)(excStack+16); /\* sp \*/

excContext->lr = (Ptr)excStack[13]; /\* lr \*/

excContext->pc = (Ptr)excStack[14]; /\* pc \*/

excContext->psr = (Ptr)excStack[15]; /\* psr \*/

**switch** (excContext->threadType) {

**case** BIOS\_ThreadType\_Task: {

**if** (BIOS\_taskEnabled) {

excContext->threadHandle = (Ptr)Task\_self();

stack = (Task\_self())->stack;

stackSize = (Task\_self())->stackSize;

}

**break**;

}

**case** BIOS\_ThreadType\_Swi: {

**if** (BIOS\_swiEnabled) {

excContext->threadHandle = (Ptr)Swi\_self();

stack = Hwi\_module->isrStackBase;

stackSize = Hwi\_module->isrStackSize;

}

**break**;

}

**case** BIOS\_ThreadType\_Hwi: {

excContext->threadHandle =

(Ptr)Hwi\_getHandle((UInt)(excContext->psr) & 0xff);

stack = Hwi\_module->isrStackBase;

stackSize = Hwi\_module->isrStackSize;

**break**;

}

**case** BIOS\_ThreadType\_Main: {

excContext->threadHandle = NULL;

stack = Hwi\_module->isrStackBase;

stackSize = Hwi\_module->isrStackSize;

**break**;

}

}

excContext->threadStackSize = stackSize;

excContext->threadStack = (Ptr)stack;

/\* copy thread's stack contents if user has provided a buffer \*/

**if** (Hwi\_module->excStack[coreId] != NULL) {

Char \*from, \*to;

from = stack;

to = (Char \*)Hwi\_module->excStack[coreId];

**while** (stackSize--) {

\*to++ = \*from++;

}

}

excContext->ICSR = (Ptr)Hwi\_nvic.ICSR;

excContext->MMFSR = (Ptr)(UInt32)Hwi\_nvic.MMFSR;

excContext->BFSR = (Ptr)(UInt32)Hwi\_nvic.BFSR;

excContext->UFSR = (Ptr)(UInt32)Hwi\_nvic.UFSR;

excContext->HFSR = (Ptr)Hwi\_nvic.HFSR;

excContext->DFSR = (Ptr)Hwi\_nvic.DFSR;

excContext->MMAR = (Ptr)Hwi\_nvic.MMAR;

excContext->BFAR = (Ptr)Hwi\_nvic.BFAR;

excContext->AFSR = (Ptr)Hwi\_nvic.AFSR;

}

/\*

\* ======== Hwi\_excNmi ========

\*/

Void **Hwi\_excNmi**(UInt \*excStack)

{

Error\_Block eb;

Error\_init(&eb);

Error\_raise(&eb, Hwi\_E\_NMI, NULL, 0);

}

/\*

\* ======== Hwi\_excHardFault ========

\*/

Void **Hwi\_excHardFault**(UInt \*excStack)

{

Char \*fault;

Error\_Block eb;

Error\_init(&eb);

**if** (Hwi\_nvic.HFSR & 0x40000000) {

Error\_raise(&eb, Hwi\_E\_hardFault, "FORCED", 0);

Hwi\_excUsageFault(excStack);

Hwi\_excBusFault(excStack);

Hwi\_excMemFault(excStack);

**return**;

}

**else** **if** (Hwi\_nvic.HFSR & 0x80000000) {

Error\_raise(&eb, Hwi\_E\_hardFault, "DEBUGEVT", 0);

Hwi\_excDebugMon(excStack);

**return**;

}

**else** **if** (Hwi\_nvic.HFSR & 0x00000002) {

fault = "VECTBL";

}

**else** {

fault = "Unknown";

}

Error\_raise(&eb, Hwi\_E\_hardFault, fault, 0);

}

/\*

\* ======== Hwi\_excMemFault ========

\*/

Void **Hwi\_excMemFault**(UInt \*excStack)

{

Char \*fault;

Error\_Block eb;

Error\_init(&eb);

**if** (Hwi\_nvic.MMFSR) {

**if** (Hwi\_nvic.MMFSR & 0x10) {

fault = "MSTKERR: Stacking Error (RD/WR failed), Stack Push";

}

**else** **if** (Hwi\_nvic.MMFSR & 0x08) {

fault = "MUNSTKERR: Unstacking Error (RD/WR failed), Stack Pop";

}

**else** **if** (Hwi\_nvic.MMFSR & 0x02) {

fault = "DACCVIOL: Data Access Violation (RD/WR failed)";

}

**else** **if** (Hwi\_nvic.MMFSR & 0x01) {

fault = "IACCVIOL: Instruction Access Violation";

}

**else** {

fault = "Unknown";

}

Error\_raise(&eb, Hwi\_E\_memFault, fault, Hwi\_nvic.MMAR);

}

}

/\*

\* ======== Hwi\_excBusFault ========

\*/

Void **Hwi\_excBusFault**(UInt \*excStack)

{

Char \*fault;

Error\_Block eb;

Error\_init(&eb);

**if** (Hwi\_nvic.BFSR) {

**if** (Hwi\_nvic.BFSR & 0x10) {

fault = "STKERR: Bus Fault caused by Stack Push";

}

**else** **if** (Hwi\_nvic.BFSR & 0x08) {

fault = "UNSTKERR: Bus Fault caused by Stack Pop";

}

**else** **if** (Hwi\_nvic.BFSR & 0x04) {

fault = "IMPRECISERR: Delayed Bus Fault, exact addr unknown";

}

**else** **if** (Hwi\_nvic.BFSR & 0x02) {

fault = "PRECISERR: Immediate Bus Fault, exact addr known";

}

**else** **if** (Hwi\_nvic.BFSR & 0x01) {

fault = "IBUSERR: Instruction Access Violation";

}

**else** {

fault = "Unknown";

}

Error\_raise(&eb, Hwi\_E\_busFault, fault, Hwi\_nvic.BFAR);

}

}

/\*

\* ======== Hwi\_excUsageFault ========

\*/

Void **Hwi\_excUsageFault**(UInt \*excStack)

{

Char \*fault;

Error\_Block eb;

Error\_init(&eb);

**if** (Hwi\_nvic.UFSR) {

**if** (Hwi\_nvic.UFSR & 0x0001) {

fault = "UNDEFINSTR: Undefined instruction";

}

**else** **if** (Hwi\_nvic.UFSR & 0x0002) {

fault = "INVSTATE: Invalid EPSR and instruction combination";

}

**else** **if** (Hwi\_nvic.UFSR & 0x0004) {

fault = "INVPC: Invalid PC";

}

**else** **if** (Hwi\_nvic.UFSR & 0x0008) {

fault = "NOCP: Attempting to use co-processor";

}

**else** **if** (Hwi\_nvic.UFSR & 0x0100) {

fault = "UNALIGNED: Unaligned memory access";

}

**else** **if** (Hwi\_nvic.UFSR & 0x0200) {

fault = "DIVBYZERO";

}

**else** {

fault = "Unknown";

}

Error\_raise(&eb, Hwi\_E\_usageFault, fault, 0);

}

}

/\*

\* ======== Hwi\_excSvCall ========

\*/

Void **Hwi\_excSvCall**(UInt \*excStack)

{

UInt8 \*pc;

Error\_Block eb;

Error\_init(&eb);

pc = (UInt8 \*)excStack[14];

Error\_raise(&eb, Hwi\_E\_svCall, pc[-2], 0);

}

/\*

\* ======== Hwi\_excDebugMon ========

\*/

Void **Hwi\_excDebugMon**(UInt \*excStack)

{

Char \*fault;

Error\_Block eb;

Error\_init(&eb);

**if** (Hwi\_nvic.DFSR) {

**if** (Hwi\_nvic.DFSR & 0x00000010) {

fault = "EXTERNAL";

}

**else** **if** (Hwi\_nvic.DFSR & 0x00000008) {

fault = "VCATCH";

}

**else** **if** (Hwi\_nvic.DFSR & 0x00000004) {

fault = "DWTTRAP";

}

**else** **if** (Hwi\_nvic.DFSR & 0x00000002) {

fault = "BKPT";

}

**else** **if** (Hwi\_nvic.DFSR & 0x00000001) {

fault = "HALTED";

}

**else** {

fault = "Unknown";

}

Error\_raise(&eb, Hwi\_E\_debugMon, fault, 0);

}

}

/\*

\* ======== Hwi\_excReserved ========

\*/

Void **Hwi\_excReserved**(UInt \*excStack, UInt excNum)

{

Error\_Block eb;

Error\_init(&eb);

Error\_raise(&eb, Hwi\_E\_reserved, "Exception #:", excNum);

}

/\*

\* ======== Hwi\_excNoIsr ========

\*/

Void **Hwi\_excNoIsr**(UInt \*excStack, UInt excNum)

{

Error\_Block eb;

Error\_init(&eb);

Error\_raise(&eb, Hwi\_E\_noIsr, excNum, excStack[14]);

}

**#if** defined(\_\_GNUC\_\_)

**#pragma** GCC diagnostic ignored "-Wint-to-pointer-cast"

**#endif**

**#if** defined(\_\_GNUC\_\_)

**#pragma** GCC diagnostic warning "-Wint-to-pointer-cast"

**#endif**

/\*

\* ======== Hwi\_excDumpRegs ========

\*/

Void **Hwi\_excDumpRegs**(UInt lr)

{

Hwi\_ExcContext \*excp;

Char \*ttype;

UInt coreId = 0;

Char \*name;

excp = Hwi\_module->excContext[coreId];

**switch** (lr) {

**case** 0xfffffff1:

System\_printf("Exception occurred in ISR thread at PC = 0x%08x.\n", excp->pc);

**break**;

**case** 0xfffffff9:

**case** 0xfffffffd:

System\_printf("Exception occurred in background thread at PC = 0x%08x.\n", excp->pc);

**break**;

**default**:

System\_printf("Bogus Exception return value: %08x.\n", lr);

**break**;

}

**switch** (excp->threadType) {

**case** BIOS\_ThreadType\_Task: {

ttype = "Task";

**if** (BIOS\_taskEnabled) {

name = Task\_Handle\_name(excp->threadHandle);

}

**else** {

name = "Not known";

}

**break**;

}

**case** BIOS\_ThreadType\_Swi: {

ttype = "Swi";

**if** (BIOS\_swiEnabled) {

name = Swi\_Handle\_name(excp->threadHandle);

}

**else** {

name = "Not known";

}

**break**;

}

**case** BIOS\_ThreadType\_Hwi: {

ttype = "Hwi";

name = Hwi\_Handle\_name(excp->threadHandle);

**break**;

}

**case** BIOS\_ThreadType\_Main: {

ttype = "Main";

name = "main()";

**break**;

}

}

**if** (name == NULL) {

name = "(unnamed)";

}

System\_printf("Core %d: Exception occurred in ThreadType\_%s.\n", coreId, ttype);

System\_printf("%s name: %s, handle: 0x%x.\n", ttype, name, excp->threadHandle);

System\_printf("%s stack base: 0x%x.\n", ttype, excp->threadStack);

System\_printf("%s stack size: 0x%x.\n", ttype, excp->threadStackSize);

System\_printf("R0 = 0x%08x R8 = 0x%08x\n", excp->r0, excp->r8);

System\_printf("R1 = 0x%08x R9 = 0x%08x\n", excp->r1, excp->r9);

System\_printf("R2 = 0x%08x R10 = 0x%08x\n", excp->r2, excp->r10);

System\_printf("R3 = 0x%08x R11 = 0x%08x\n", excp->r3, excp->r11);

System\_printf("R4 = 0x%08x R12 = 0x%08x\n", excp->r4, excp->r12);

System\_printf("R5 = 0x%08x SP(R13) = 0x%08x\n", excp->r5, excp->sp);

System\_printf("R6 = 0x%08x LR(R14) = 0x%08x\n", excp->r6, excp->lr);

System\_printf("R7 = 0x%08x PC(R15) = 0x%08x\n", excp->r7, excp->pc);

System\_printf("PSR = 0x%08x\n", excp->psr);

System\_printf("ICSR = 0x%08x\n", Hwi\_nvic.ICSR);

System\_printf("MMFSR = 0x%02x\n", Hwi\_nvic.MMFSR);

System\_printf("BFSR = 0x%02x\n", Hwi\_nvic.BFSR);

System\_printf("UFSR = 0x%04x\n", Hwi\_nvic.UFSR);

System\_printf("HFSR = 0x%08x\n", Hwi\_nvic.HFSR);

System\_printf("DFSR = 0x%08x\n", Hwi\_nvic.DFSR);

System\_printf("MMAR = 0x%08x\n", Hwi\_nvic.MMAR);

System\_printf("BFAR = 0x%08x\n", Hwi\_nvic.BFAR);

System\_printf("AFSR = 0x%08x\n", Hwi\_nvic.AFSR);

}

/\*

\* ======== Hwi\_flushVnvic ========

\*/

Void **Hwi\_flushVnvic**()

{

}

/\*

\* ======== Hwi\_dispatchC ========

\* "Top Half" of Configurable IRQ interrupt dispatcher.

\*/

UInt **Hwi\_dispatchC**(Hwi\_Irp irp, UInt32 dummy1, UInt32 dummy2, Hwi\_Object \*hwi)

{

/\*

\* Enough room is reserved above the isr stack to handle

\* as many as 16 32-bit stack resident local variables.

\* If the dispatcher requires more than this, you must

\* handle this in Hwi\_Module\_startup().

\*/

BIOS\_ThreadType prevThreadType;

UInt tskKey = 1;

UInt swiKey = 1;

**#ifndef** ti\_sysbios\_hal\_Hwi\_DISABLE\_ALL\_HOOKS

Int i;

**#endif**

Hwi\_disable();

**if** (Hwi\_dispatcherTaskSupport) {

tskKey = TASK\_DISABLE();

}

**if** (Hwi\_dispatcherSwiSupport) {

swiKey = SWI\_DISABLE();

}

/\* set thread type to Hwi \*/

prevThreadType = BIOS\_setThreadType(BIOS\_ThreadType\_Hwi);

**if** (Hwi\_numSparseInterrupts == 0) {

UInt intNum = (Hwi\_nvic.ICSR & 0x000000ff);

Hwi\_Object \*\*dispatchTable = (Hwi\_Object \*\*)Hwi\_module->dispatchTable;

hwi = dispatchTable[intNum];

}

/\* IRP tracking is always enabled for M3 \*/

hwi->irp = irp;

**#ifndef** ti\_sysbios\_hal\_Hwi\_DISABLE\_ALL\_HOOKS

/\* call the begin hooks \*/

**for** (i = 0; i < Hwi\_hooks.length; i++) {

**if** (Hwi\_hooks.elem[i].beginFxn != NULL) {

Hwi\_hooks.elem[i].beginFxn((IHwi\_Handle)hwi);

}

}

**#endif**

Log\_write5(Hwi\_LM\_begin, (IArg)hwi, (IArg)hwi->fxn,

(IArg)prevThreadType, (IArg)hwi->intNum, hwi->irp);

/\* call the user's isr \*/

**if** (Hwi\_dispatcherAutoNestingSupport) {

Hwi\_enable();

(hwi->fxn)(hwi->arg);

Hwi\_disable();

}

**else** {

(hwi->fxn)(hwi->arg);

}

Log\_write1(Hwi\_LD\_end, (IArg)hwi);

**#ifndef** ti\_sysbios\_hal\_Hwi\_DISABLE\_ALL\_HOOKS

/\* call the end hooks \*/

**for** (i = 0; i < Hwi\_hooks.length; i++) {

**if** (Hwi\_hooks.elem[i].endFxn != NULL) {

Hwi\_hooks.elem[i].endFxn((IHwi\_Handle)hwi);

}

}

**#endif**

/\* restore thread type \*/

BIOS\_setThreadType(prevThreadType);

/\* encode both tskKey and swiKey in return \*/

**return** ((tskKey << 8) + swiKey);

}

/\*

\* ======== Hwi\_doSwiRestore ========

\* Run swi scheduler. (Executes on Hwi stack)

\*/

Void **Hwi\_doSwiRestore**(UInt swiTskKey)

{

/\* Run Swi scheduler \*/

**if** (Hwi\_dispatcherSwiSupport) {

SWI\_RESTORE(swiTskKey & 0xff); /\* Run Swi scheduler \*/

}

}

/\*

\* ======== Hwi\_doTaskRestore ========

\* Run task scheduler. (Executes on Task stack)

\*/

Void **Hwi\_doTaskRestore**(UInt swiTskKey)

{

/\* Run Task scheduler \*/

**if** (Hwi\_dispatcherTaskSupport) {

TASK\_RESTORE(swiTskKey >> 8); /\* returns with ints disabled \*/

}

}