Chipcon Products from Texas Instruments

Z-Stack HAL Porting Guide

Document Number: F8W-2006-0024

Texas Instruments, Inc.
San Diego, California USA
(619) 497-3845

Revision	Description	Date
1.0	Initial release.	03/14/2007

TABLE OF CONTENTS

1. Introduction

1.1. Scope

This document provides information on how to take a Z-Stack™ release and customize the HAL layer so that Z-Stack can run on the user's own target hardware.

1.2. Usage

Each section will provide the user the information and options to customize it if possible. This document covers the following:

- HAL component folders (compiling options, customization)
- GPIO pin interface (HAL drivers, Radio, and MCU)
- External Interrupt ISR handling

1.3. Acronyms

ADC	Analog to Digital Conversion
API	Application Programming Interface.
GPIO	General Purpose Input Output
HAL	Hardware Abstraction Layer
MAC	Medium Access Control.
OSAL	Operating System Abstraction Layer.

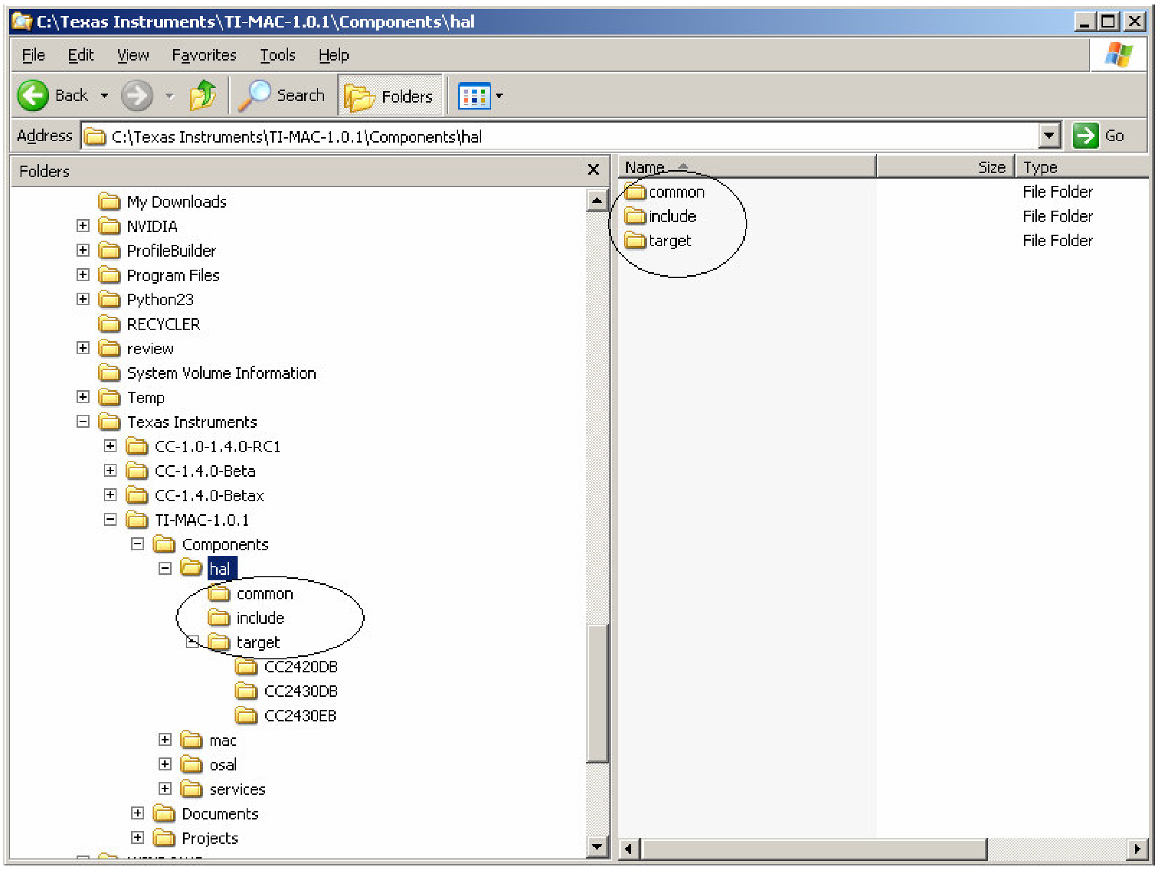
2. HAL

HAL is the layer that provides hardware services without exposing too much hardware related material so that the users can focus more on their application. Majority of the HAL is the drivers that control LEDs, LCD, ADC, KEYs, Timers and UART. These services are abstracted by simple APIs which allow users to use these services without worrying about setting up these services at GPIO level.

2.1. HAL Component Folders

HAL Component folders contain HAL drivers and HAL related files. HAL Component folders are organized into 3 different folders under **\Component\hal**.

- Common
- Include
- Target



2.1.1. HAL Common Folder

HAL common folder contains files that can be used in stack, MAC and drivers but they are HAL related in some way. There are 2 files in this folder: **hal_assert.c** and **hal_driver.c**.

2.1.1.1. hal_assert.c

This file contains macros that are used to debug and/or verify data. These macros are used by MAC (high, low level) and HAL drivers.

HAL_ASSERT ()

This macro is for use during debugging. The given expression must evaluate as "true" or else a fatal error occurs. From here, the call stack feature of the debugger can pinpoint where the problem occurred. This feature can be disabled for optimum performance and minimum code size (ideal for finalized, debugged production code). To disable, define the preprocessor symbol **HALNODEBUG** at the project level.

HAL_ASSERT_SIZE ()

This macro compares the size of the first parameter to the integer value of the second parameter. If they do not match, a compile time error for negative array size occurs (even gnu chokes on negative array size). This compare is done by creating a typedef for an array. No variables are created and no memory is consumed with this check. The created type is used for checking only and is not for use by any other code.

2.1.1.2. hal_driver.c

This file contains initializing and event processing routines for all HAL drivers. This is where hal driver related events are processed. It also contains polling mechanism for the drivers such as Timer and UART.

Hal_Init ()
This is called by osalTaskAdd () to register HAL drivers as a task in the OSAL. This will allow HAL drivers messages and events to be processed by the OSAL accordingly. User can add non hardware related initializations in here. These will be initialized by the OSAL when the HAL driver task is created.

HalDriverInit ()
This is called by main () to initialize the hardware of the drivers. User can add hardware related initializations in here. All these initializations will be done at the beginning of the program.

Hal_ProcessEvent ()
This file is registered to OSAL to handle HAL related events coming from the drivers such as keys, leds, and power saving. Users can add their own HAL related events in here. The event id must be unique and must be defined in **hal_driver.h**

For example: These are some of the predefined events

HAL Event	Description
HAL_KEY_EVENT	This event is issued by hal_key to handle key polling and de-bounce
HAL_LED_BLINK_EVENT	This event is issued by hal_led to handle blinking leds
HAL_SLEEP_TIMER_EVENT	This event is issued by Power Saving

Hal_ProcessPoll ()
This file is called by osal_start_system () in the main infinite loop to process polling events for the HAL Timer and HAL UART drivers.

2.1.2. HAL Include Folder

HAL Include folder contains header files for HAL drivers and HAL related files.

For example: These are the files for CC2430EB target under Include folder.

File	Type
hal_adc.h	Driver
hal_key.h	Driver
hal_lcd.h	Driver

hal_led.h	Driver
hal_timer.h	Driver
hal_uart.h	Driver
hal_defs.h	Macros and Type Definitions
hal_driver.h	Driver
hal_sleep.h	Sleep/Power Saving
hal_assert.h	Debug

2.1.3. HAL Target Folder

HAL Target folder contains target specific files. This folder can include drivers, board configuration, MCU information and data types that associate to the specific target only.

For example: These are the files for CC2430EB target under Target folder.

File	Type
hal_adc.c	Driver
hal_key.c	Driver
hal_lcd.c	Driver
hal_led.c	Driver
hal_timer.c	Driver
hal_uart.c	Driver
hal_mcu.c	MCU Configuration specific to the target
hal_board_cfg.h	Drivers Configuration – Contains hardware/GPIO configuration for the drivers specific to the target
hal_mac_cfg.c hal_mac_cfg.h	MAC Configuration specific to the target
hal_sleep.h	Sleep/Power Saving specific to the target
hal_target.c hal_target.h	Reserved
hal_types.h	Typedefs and keywords specific to the target

2.2. HAL Drivers Compile Options

The following compiling options allow the user to compile IN or OUT certain HAL driver. Currently when a driver is compiled OUT, the API routines are still accessible but the body of the routines will not be compiled and executed. When a routine is not called from within the application, this routine will not be linked into the application at all. This means that it not will occupy any code space

Symbol	Definition
HAL_ADC	ADC driver
HAL_LCD	LCD driver
HAL_LED	LED driver
HAL_KEY	Key driver

HAL_UART	UART driver

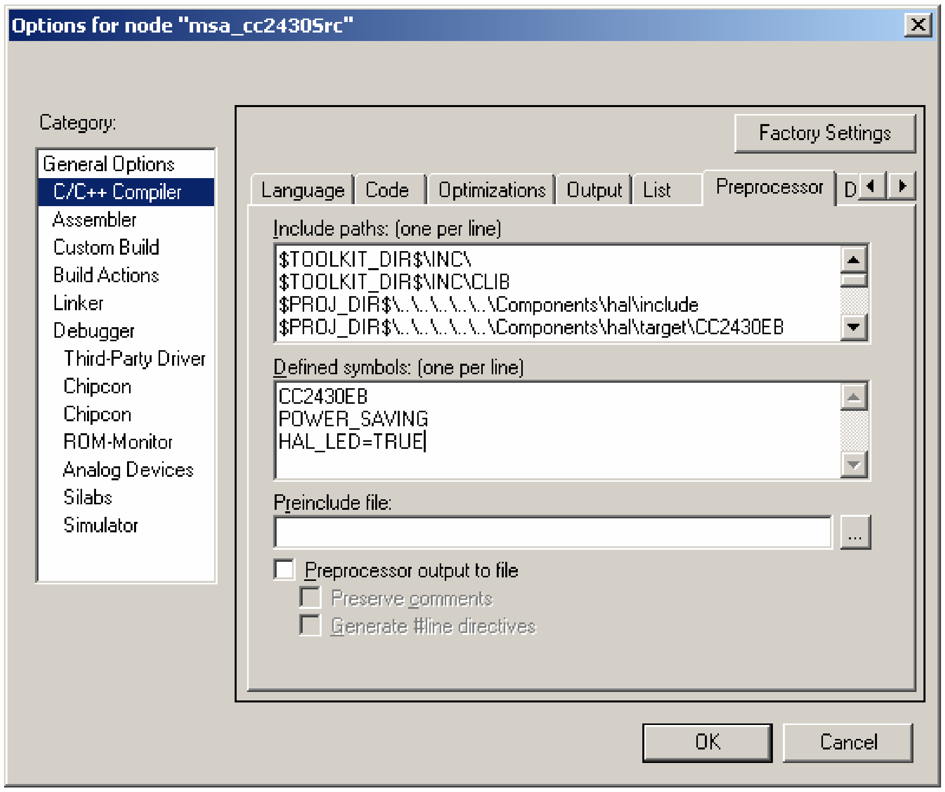
2.2.1. Default Compile Options

By default, HAL_ADC, HAL_LCD, HAL_LED, HAL_KEY and HAL_UART are set to TRUE by HAL driver. These are defined in **\Components\hal\target\XXX\hal_board_cfg.h** where XXX is the target (CC2430DB, CC2430EB, CC2420DB…etc)

2.2.2. Compiling IN Drivers

To compile the project with a certain HAL driver

1. Open a project
2. Right click on project
3. Select **Options**
4. Select **C/C++ Options**
5. Select **Preprocessor**
6. Go to **Defined Symbols**
7. Enter **HAL_XXX=TRUE** where **XXX** is ADC, LCD, LED, KEY or UART



Include paths: (one per line)

```
$TOOLKIT_DIR$\INC\
$TOOLKIT_DIR$\INC\CLIB
$PROJ_DIR$\..\..\..\..\..\Components\hal\include
$PROJ_DIR$\..\..\..\..\..\Components\hal\target\CC2430EB
```

Defined symbols: (one per line)

```
CC2430EB
POWER_SAVING
HAL_LED=TRUE
```

Preinclude file:

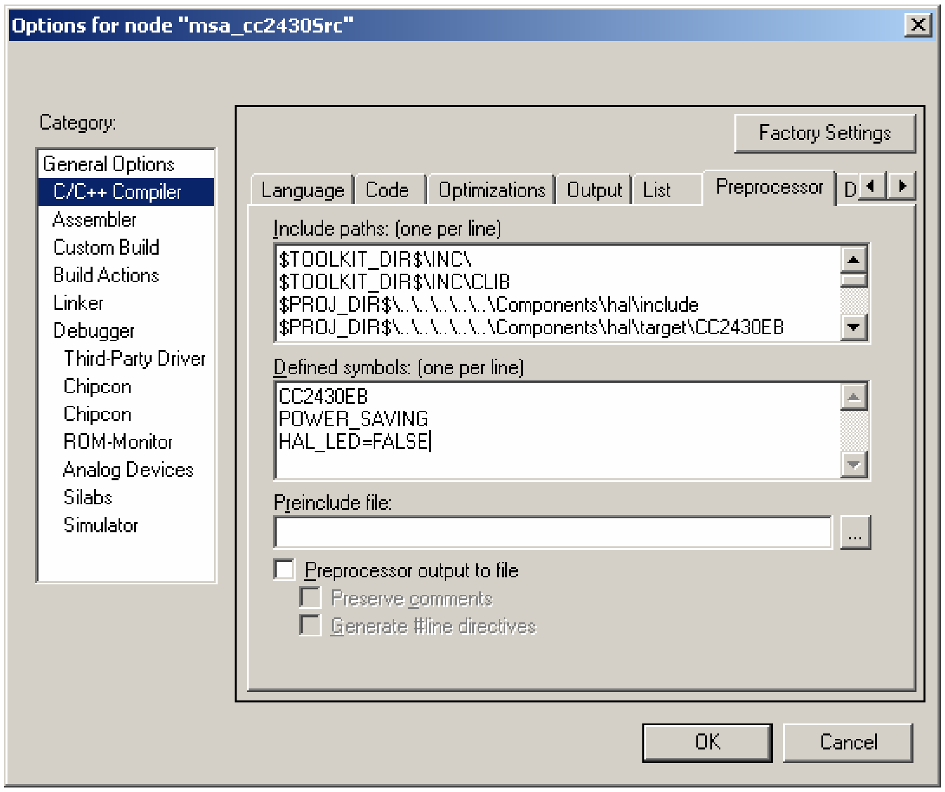
☐ Preprocessor output to file
☐ Preserve comments
☐ Generate #line directives

2.2.3. Compiling OUT Drivers

To compile a project without a certain HAL driver

1. Open a project
2. Right click on project

3. Select **Options**
4. Select **C/C++ Options**
5. Select **Preprocessor**
6. Go to **Defined Symbols**
7. Enter **HAL_XXX=FALSE** where **XXX** is ADC, LCD, LED, KEY, UART



Include paths: (one per line)

```
$TOOLKIT_DIR$\INC\
$TOOLKIT_DIR$\INC\CLIB
$PROJ_DIR$\..\..\..\..\..\Components\hal\include
$PROJ_DIR$\..\..\..\..\..\Components\hal\target\CC2430EB
```

Defined symbols: (one per line)

```
CC2430EB
POWER_SAVING
HAL_LED=FALSE
```

2.3. HAL Drivers Customization

Even though HAL drivers are flexible, there is time when the user decides to customize their HAL drivers to do a specific thing that fits their needs. Users can either modify the existing drivers to work with their hardware or simply add their own drivers as they see fit.

2.3.1. Modifying Existing HAL Drivers

When the user decides to modify the drivers to support different features that are not provided by default, the following rules should be followed:

1. All the header files located at **\Components\hal\include** should remain the same. These prototypes are called by different parts of the program and should not be modified.

2. The functionality of the each driver can be changed by modifying corresponding function in **\Components\hal\target\XXX** where **XXX** is the target. Driver files are hal_adc.c, hal_key.c, hal_lcd.c, hal_led.c, hal_timer.c, and hal_uart.c.

3. Drivers hardware configuration can be modified through **hal_board_cfg.h**

2.3.2. Adding Drivers For New Target

Users can add new drivers or custom drivers to support their own hardware.

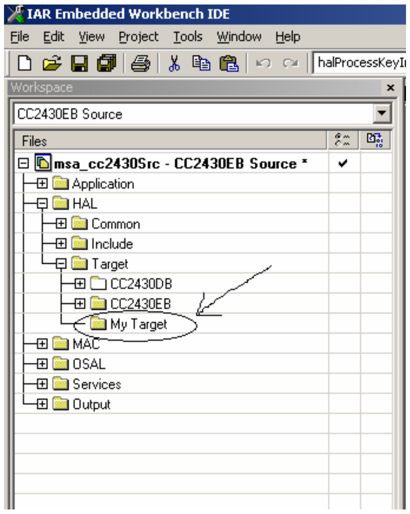
1. All the new header files go to **\Components\hal\include**.

2. All the files that contain the actual functions/routines go to **\Components\hal\target\XXX** where **XXX** is the user's defined target.

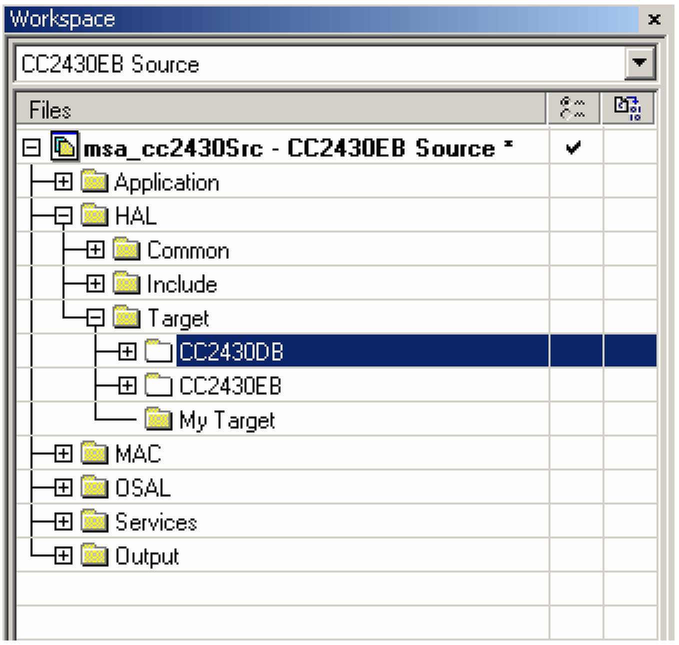
3. If GPIO pins conflict or are unwanted, certain provided drivers can be compiled OUT using compile flags. Check **HAL Drivers Compiling Options** section for more details.

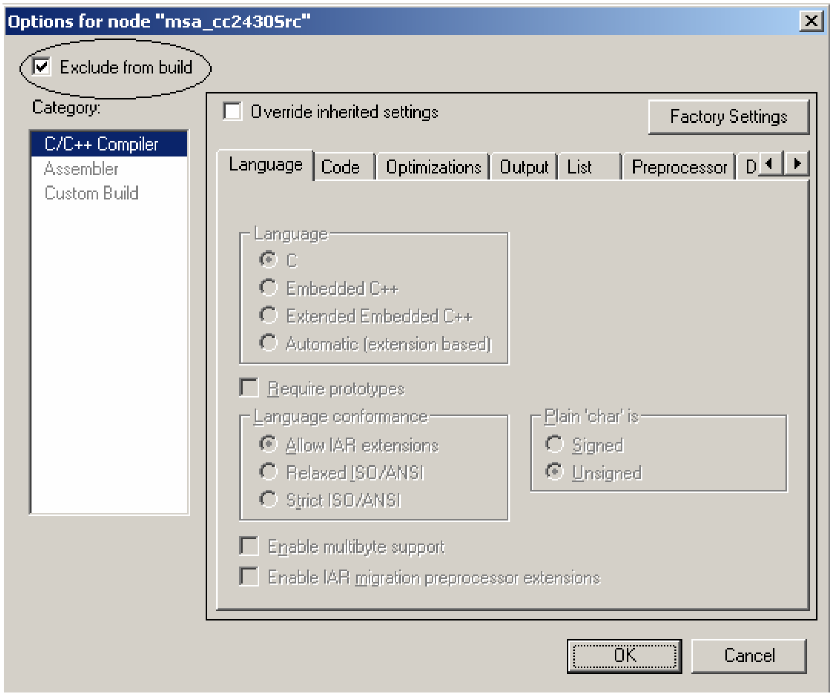
4. Verify GPIO pins are correctly setup by checking **hal_board_cfg.h** for conflicts. Board specific configuration GPIO for hal drivers is located here.

5. Project file has to be modified accordingly. New target must be added under HAL folder in the project file together with all the new files.



6. For project that includes multiple targets, un-used or old targets must be excluded from build to prevent conflicts. This can be done by right click on the target, select options and then check "Exclude from build".




Options for node "msa_cc2430Src"

[✓] Exclude from build

Category: C/C++ Compiler, Assembler, Custom Build

[ ] Override inherited settings

Language: [✓ C] Embedded C++ / Extended Embedded C++ / Automatic (extension based)

[ ] Require prototypes

Language conformance: [✓ Allow IAR extensions] Relaxed ISO/ANSI / Strict ISO/ANSI

Plain 'char' is: Signed [✓ Unsigned]

[ ] Enable multibyte support

[ ] Enable IAR migration preprocessor extensions

3. GPIO and Radio/MCU Pin Interface

GPIO and Radio/MCU pin definitions are located at different places/files depending on which purpose the pins serve. If the user decides to alter the GPIO configuration, to use a different pin-out between the MCU and Radio, these are the files in the following locations need to be modified:

3.1. HAL Drivers GPIO Pin Interface Modification

HAL drivers pin definitions are located in **\Components\hal\target\XXX\hal_board_cfg.h** where **XXX** is the target.

3.2. Radio/MCU GPIO Pin Interface Modification

Radio/MCU pin definitions are located in **\Components\hal\target\XXX\hal_mac_cfg.h** and **\Components\hal\target\XXX\hal_mac_cfg.c** where **XXX** is the target.

4. Interrupt Service Routine.

This section shows the user where to modify an existing Interrupt Service Routine (ISR) and how to create a new ISR for an interrupt.

4.1. HAL ISR Function Macro

To abstract the differences between compilers a macro defined as `HAL_ISR_FUNCTION(f,v)` in **\Components\hal\target\XXX\hal_mcu.h** (where **XXX** is the target) is used to declare the ISR. For more detail please check **hal_mcu.h**.

To use the macro, simply declare:

```
HAL_ISR_FUNCTION (prototype, vector)
{
   /* Do something when this interrupt happens!!! */
}
```

Where **prototype** is the name of the ISR and **vector** is the interrupt vector.

4.2. Modifying Existing ISR For GPIO Interrupt

To modify an existing ISR for a GPIO interrupt, simply go to the where the ISR routine is declared and modify it.

4.2.1. Example : Modifying ISR in a CC2430 project

Example: Modify **P1INT_VECTOR** interrupt service routine in a CC2430 project.

P1INT_VECTOR interrupt service routine is declared in hal_key.c.

```
HAL_ISR_FUNCTION (halKeyPort1Isr, P1INT_VECTOR)
{
  halProcessKeyInterrupt ();
}
```

4.3. Adding New ISR For GPIO Interrupt

To add a new ISR, the user must be provided with a header file that contains information of the target's microcontroller. This file provides information about the **interrupt vector definition** and/or **byte address** of the specific GPIO port that will be used. This information often is provided by the MCU manufacturer. The user also needs to know the **prototype** of the ISR.

4.3.1. Example 1: Creating an ISR for a CC2430 target

Create an ISR for timer 1 (**T1_VECTOR**)

Step 1: For **CC2430**, the interrupt vector definitions locate in **ioCC2430.h** and it partially looks like this:

```
/* ------------------------------------------------------------------------------------------------
 *                                Interrupt Vectors
 * ------------------------------------------------------------------------------------------------
 */
#define  RFERR_VECTOR   VECT(  0, 0x03 )
#define  ADC_VECTOR     VECT(  1, 0x0B )
#define  URX0_VECTOR    VECT(  2, 0x13 )
#define  URX1_VECTOR    VECT(  3, 0x1B )
#define  ENC_VECTOR     VECT(  4, 0x23 )
#define  ST_VECTOR      VECT(  5, 0x2B )
#define  P2INT_VECTOR   VECT(  6, 0x33 )
#define  UTX0_VECTOR    VECT(  7, 0x3B )
#define  DMA_VECTOR     VECT(  8, 0x43 )
#define  T1_VECTOR      VECT(  9, 0x4B )  ← Timer 1 vector
#define  T2_VECTOR      VECT( 10, 0x53 )
#define  T3_VECTOR      VECT( 11, 0x5B )
#define  T4_VECTOR      VECT( 12, 0x63 )
#define  P0INT_VECTOR   VECT( 13, 0x6B )
#define  UTX1_VECTOR    VECT( 14, 0x73 )
#define  P1INT_VECTOR   VECT( 15, 0x7B )
#define  RF_VECTOR      VECT( 16, 0x83 )
#define  WDT_VECTOR     VECT( 17, 0x8B )
```

Step 2: Use the macro, define the ISR.

```
HAL_ISR_FUNCTION (My_Timer1_ISR, T1_VECTOR)
{
 /* Do something when this timer 1 happens!!! */
}
```

4.3.2. Example 2: Creating an ISR for a MSP430 target

Create an ISR for **USART0TX_VECTOR**

Step1: For **MSP430**, the interrupt vector definitions locate in **msp430x16x.h** and it partially looks like this:

```
/************************************************************
* Interrupt Vectors (offset from 0xFFE0)
************************************************************/

#define DACDMA_VECTOR       (0 * 2u)
#define PORT2_VECTOR        (1 * 2u)
#define USART1TX_VECTOR     (2 * 2u)
#define USART1RX_VECTOR     (3 * 2u)
#define PORT1_VECTOR        (4 * 2u)
#define TIMERA1_VECTOR      (5 * 2u)
```

```
#define TIMERA0_VECTOR          (6 * 2u)
#define ADC12_VECTOR            (7 * 2u)
#define USART0TX_VECTOR         (8 * 2u)  ← UART0 Transmit vector
#define USART0RX_VECTOR         (9 * 2u)
#define WDT_VECTOR              (10 * 2u)
#define COMPARATORA_VECTOR(11 * 2u)
#define TIMERB1_VECTOR          (12 * 2u)
#define TIMERB0_VECTOR          (13 * 2u)
#define NMI_VECTOR              (14 * 2u)
#define RESET_VECTOR            (15 * 2u)
```

Step 2: Use the macro, define the ISR

```
HAL_ISR_FUNCTION (MyTx0Isr, UART0TX_VECTOR)
{
 /* Do something when this UART0TX_VECTOR happens!!! */
}
```

4.3.3. Example 3: Creating an ISR for a AVR target

Create an ISR for **INT0_vect** on an **AVR**.

Step 1: First check **iom128.h** which is provided by the manufacturer. This is how it partially looks like:

```
*==============================*/
/* Interrupt Vector Definitions */
/*==============================*/

/* NB! vectors are specified as byte addresses */

#define    RESET_vect           (0x00)
#define    INT0_vect            (0x04)  ← INT0_vect vector
#define    INT1_vect            (0x08)
#define    INT2_vect            (0x0C)
#define    INT3_vect            (0x10)
#define    INT4_vect            (0x14)
#define    INT5_vect            (0x18)
#define    INT6_vect            (0x1C)
#define    INT7_vect            (0x20)
#define    TIMER2_COMP_vect     (0x24)
#define    TIMER2_OVF_vect      (0x28)
#define    TIMER1_CAPT_vect     (0x2C)
#define    TIMER1_COMPA_vect    (0x30)
#define    TIMER1_COMPB_vect    (0x34)
#define    TIMER1_OVF_vect      (0x38)
#define    TIMER0_COMP_vect     (0x3C)
#define    TIMER0_OVF_vect      (0x40)
#define    SPI_STC_vect         (0x44)
#define    USART0_RXC_vect      (0x48)
#define    USART0_UDRE_vect     (0x4C)
#define    USART0_TXC_vect      (0x50)
#define    ADC_vect             (0x54)
#define    EE_RDY_vect          (0x58)
#define    ANA_COMP_vect        (0x5C)
```

```
Step 2: Use the macro, define the ISR

HAL_ISR_FUNCTION (MyINT0ISR, INT0_vect)
{
  /* Do something when this INT0 happens!!! */
}
```