

Hardware Abstraction Layer (HAL)

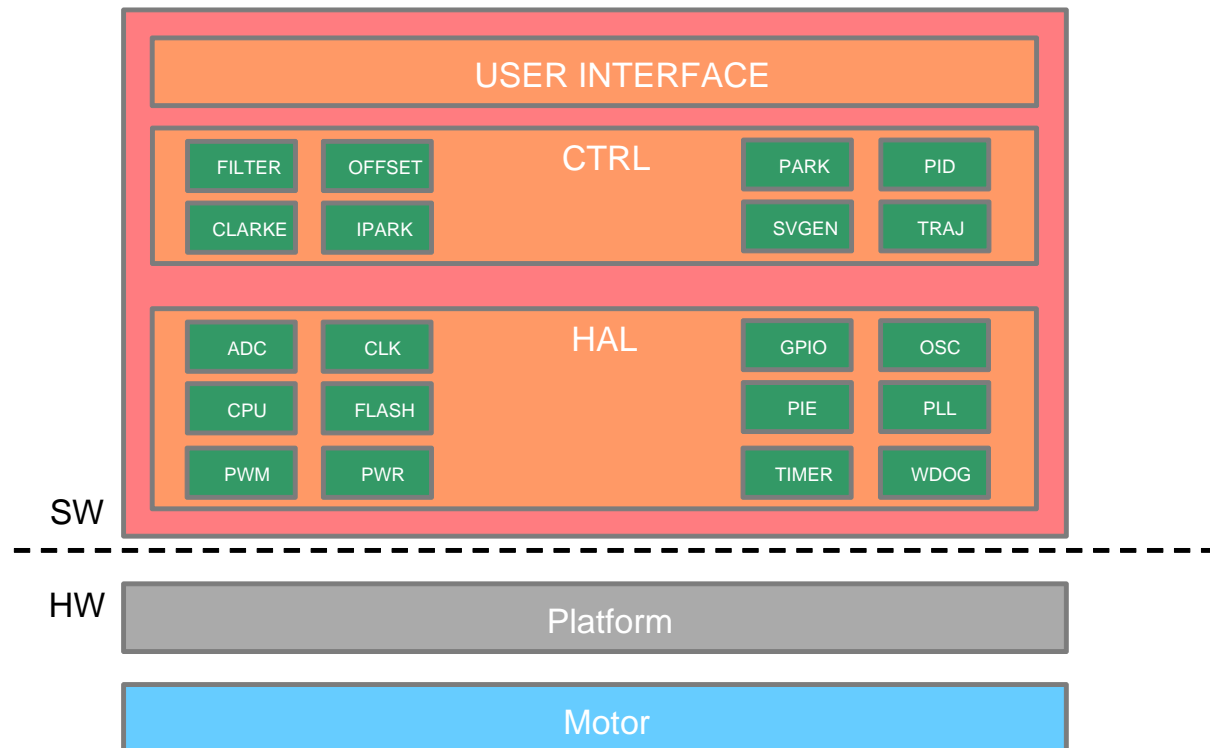
Module of MotorWare™

1. Introduction

The purpose of the HAL module is to allow a unified software base that can be ported across multiple MCUs (i.e. f2802x, f2805x and f2806x) and hardware platforms (i.e. boostxldr8301_revB, drv8301kit_revD, drv8312kit_revD and hvkit_rev1p1).

2. Software Architecture

The following block diagram shows how the HAL object fits within the software architecture followed by MotorWare:



Certain guidelines are followed by the current software architecture, which allows a better use of a HAL module:

- HAL module is the only module that can interface with the hardware. In this context, hardware includes MCU peripherals, and board functionality.
- No peripheral configuration should be done at a user's interface level, since this doesn't allow portability of the code.

- HAL functions which are time critical, should be inlined for faster execution.
- Use of macros must be avoided, and use of inline functions should be used instead.

3. Object Definition

There are several object definitions of the HAL module, depending on the processor and board combination. The object definition of the HAL module can be found within the MotorWare product tree as shown in the following picture:

	Name
\\sw\modules\hal\boards\boostxldr8301_revB\f28x\f2802x\src	hal_obj.h
\\sw\modules\hal\boards\drv8301kit_revD\f28x\f2802x\src	hal_obj.h
\\sw\modules\hal\boards\drv8301kit_revD\f28x\f2805x\src	hal_obj.h
\\sw\modules\hal\boards\drv8301kit_revD\f28x\f2806x\src	hal_obj.h
\\sw\modules\hal\boards\drv8312kit_revD\f28x\f2802x\src	hal_obj.h
\\sw\modules\hal\boards\drv8312kit_revD\f28x\f2805x\src	hal_obj.h
\\sw\modules\hal\boards\drv8312kit_revD\f28x\f2806x\src	hal_obj.h
\\sw\modules\hal\boards\hvkit_rev1p1\f28x\f2802x\src	hal_obj.h
\\sw\modules\hal\boards\hvkit_rev1p1\f28x\f2805x\src	hal_obj.h
\\sw\modules\hal\boards\hvkit_rev1p1\f28x\f2806x\src	hal_obj.h

The following object is defined for the f2806x and combination:

```

/*! \brief   Defines the hardware abstraction layer (HAL) data
/*! \details The HAL object contains all handles to peripherals. When accessing a
/*!          peripheral on a processor, use a HAL function along with the HAL handle
/*!          for that processor to access its peripherals.
/*!
typedef struct _HAL_Obj_
{
    ADC_Handle  adcHandle;    //!< the ADC handle

    CLK_Handle  clkHandle;    //!< the clock handle

    CPU_Handle  cpuHandle;    //!< the CPU handle

    FLASH_Handle flashHandle; //!< the flash handle

    GPIO_Handle gpioHandle;   //!< the GPIO handle

    OFFSET_Handle offsetHandle_I[3]; //!< the handles for the current offset estimators
    OFFSET_Obj  offset_I[3];    //!< the current offset objects

    OFFSET_Handle offsetHandle_V[3]; //!< the handles for the voltage offset estimators

```

```

OFFSET_Obj  offset_V[3];    //!< the voltage offset objects

OSC_Handle  oscHandle;      //!< the oscillator handle

PIE_Handle  pieHandle;      //!< the PIE handle

PLL_Handle  pllHandle;      //!< the PLL handle

PWM_Handle  pwmHandle[3];   //!< the PWM handles

PWMDAC_Handle pwmDacHandle[3]; //!< the PWMDAC handles

PWR_Handle  pwrHandle;      //!< the power handle

TIMER_Handle timerHandle[3]; //!< the timer handles

WDOG_Handle wdogHandle;     //!< the watchdog handle

HAL_AdcData_t adcBias;      //!< the ADC bias

_iq         current_sf;     //!< the current scale factor, amps_pu/cnt

_iq         voltage_sf;     //!< the voltage scale factor, volts_pu/cnt

uint_least8_t numCurrentSensors; //!< the number of current sensors
uint_least8_t numVoltageSensors; //!< the number of voltage sensors

#ifdef QEP
  QEP_Handle  qepHandle[2];  //!< the QEP handle
#endif

} HAL_Obj;

```

As can be seen, the HAL object definition is a combination of number of modules per peripheral, and modules related to a particular platform. For example, there is only one ADC module that we have to configure through the HAL module, so we only include one ADC module:

```
ADC_Handle  adcHandle;    //!< the ADC handle
```

And there are three PWM pairs that we have to configure, so we add three of these:

```
PWM_Handle  pwmHandle[3]; //!< the PWM handles
```

Also, related to the platform, the following HAL parameter will have the number of current sensors:

```
uint_least8_t numCurrentSensors; //!< the number of current sensors
```

4. APIs and Functions

The source files that contain all the API definitions and functions are in hal.c and hal.h. As mentioned earlier in this document, those functions which are time critical (i.e. executed inside an interrupt) should be inlined, therefore some of the HAL functions are implemented in hal.h as inlined functions.

First of all, the location of all hal.c and hal.h depends on the device and board combination as shown in the following picture:

	Name
\\sw\modules\hal\boards\boostxldr8301_revB\f28x\f2802x\src	hal.h
\\sw\modules\hal\boards\drv8301kit_revD\f28x\f2802x\src	hal.h
\\sw\modules\hal\boards\drv8301kit_revD\f28x\f2805x\src	hal.h
\\sw\modules\hal\boards\drv8301kit_revD\f28x\f2806x\src	hal.h
\\sw\modules\hal\boards\drv8312kit_revD\f28x\f2802x\src	hal.h
\\sw\modules\hal\boards\drv8312kit_revD\f28x\f2805x\src	hal.h
\\sw\modules\hal\boards\drv8312kit_revD\f28x\f2806x\src	hal.h
\\sw\modules\hal\boards\hvkit_rev1p1\f28x\f2802x\src	hal.h
\\sw\modules\hal\boards\hvkit_rev1p1\f28x\f2805x\src	hal.h
\\sw\modules\hal\boards\hvkit_rev1p1\f28x\f2806x\src	hal.h

	Name
\\sw\modules\hal\boards\boostxldr8301_revB\f28x\f2802x\src	hal.c
\\sw\modules\hal\boards\drv8301kit_revD\f28x\f2802x\src	hal.c
\\sw\modules\hal\boards\drv8301kit_revD\f28x\f2805x\src	hal.c
\\sw\modules\hal\boards\drv8301kit_revD\f28x\f2806x\src	hal.c
\\sw\modules\hal\boards\drv8312kit_revD\f28x\f2802x\src	hal.c
\\sw\modules\hal\boards\drv8312kit_revD\f28x\f2805x\src	hal.c
\\sw\modules\hal\boards\drv8312kit_revD\f28x\f2806x\src	hal.c
\\sw\modules\hal\boards\hvkit_rev1p1\f28x\f2802x\src	hal.c
\\sw\modules\hal\boards\hvkit_rev1p1\f28x\f2805x\src	hal.c
\\sw\modules\hal\boards\hvkit_rev1p1\f28x\f2806x\src	hal.c

The APIs are classified into different categories: Board management, Calibration management and Peripheral management.

4.1.Board Management

A few examples of board management HAL APIs are the ones that do not configure a peripheral directly, or manage variables related to a specific board, such as number of current sensors, number of quadrature encode modules (QEP), etc. Some examples of these APIs are:

- HAL_getBias() and HAL_setBias

- HAL_getCurrentScaleFactor() and HAL_setCurrentScaleFactor()
- HAL_getNumCurrentSensors() and HAL_setNumCurrentSensors()
- HAL_getNumVoltageSensors() and HAL_setNumVoltageSensors()
- HAL_getOffsetBeta_lp_pu() and HAL_setOffsetBeta_lp_pu()
- HAL_getOffsetValue() and HAL_setOffsetValue()
- HAL_getVoltageScaleFactor() and HAL_setVoltageScaleFactor()
- HAL_setupFaults
- HAL_readAdcData
- HAL_setOffsetInitCond()
- HAL_setParams()
- HAL_updateAdcBias()
- HAL_writeDacData()
- HAL_writePwmData()
- HAL_setTrigger()

4.2. Calibration Management

A few examples of calibration management HAL APIs are the ones that run certain code to adjust a calibration offset based on a device temperature sensor, or based on an ADC calibration subroutine. Some examples of these APIs are:

- HAL_cal()
- HAL_AdcCalChanSelect()
- HAL_AdcCalConversion()
- HAL_AdcOffsetSelfCal()
- HAL_getOscTrimValue()
- HAL_OscTempComp()
- HAL_osc1Comp()
- HAL_osc2Comp()

4.3. Peripheral Management

These APIs are the ones that configure a device peripheral, or a set of peripherals. Some of these functions are translated into a single function call to a peripheral configuration API, although they are still needed to allow the portability of the software architecture to work. An example of the simplest peripheral configuration is:

```
// enable global interrupts
HAL_enableGlobalInts(halHandle);
```

Which is implemented as follows:

```
void HAL_enableGlobalInts(HAL_Handle handle)
{
    HAL_Obj *obj = (HAL_Obj *)handle;
```

```
CPU_enableGlobalInts(obj->cpuHandle);

return;
} // end of HAL_enableGlobalInts() function
```

As can be seen, from the top level, a HAL function is called. In the HAL module itself, a CPU bit is changed. This HAL function might seem unnecessary, but it is actually needed to allow the top level to be fully portable to other MCUs or boards, which might modify another bit or bits to enable global interrupts.

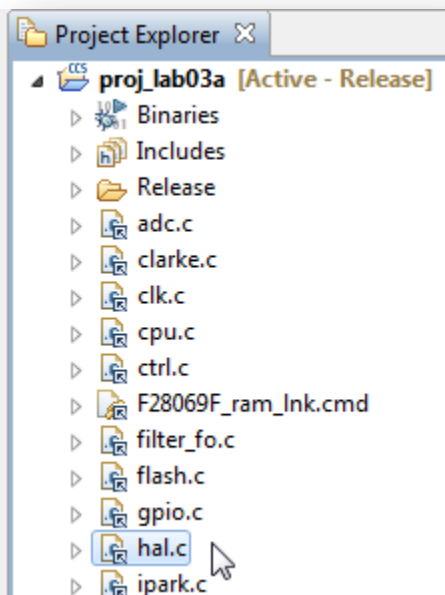
Some of the Peripheral management functions are:

- HAL_acqAdcInt()
- HAL_acqPwmInt()
- HAL_disableGlobalInts()
- HAL_disableWdog()
- HAL_disablePwm()
- HAL_enableAdcInts()
- HAL_enableDebugInt()
- HAL_enableGlobalInts()
- HAL_enablePwm()
- HAL_enablePwmInt()
- HAL_getAdcSocSampleDelay()
- HAL_initIntVectorTable()
- HAL_readTimerCnt()
- HAL_reloadTimer()
- HAL_startTimer()
- HAL_stopTimer()
- HAL_setTimerPeriod()
- HAL_getTimerPeriod()
- HAL_setAdcSocSampleDelay()
- HAL_setGpioHigh()
- HAL_toggleGpio()
- HAL_setGpioLow()
- HAL_setupAdcs()
- HAL_setupTimers()
- HAL_setupClks()
- HAL_setupFlash()
- HAL_setupGpios()
- HAL_setupPeripheralClks()

- HAL_setupPie()
- HAL_setupPII()
- HAL_setupPwms()
- HAL_setupPwmDacs()
- HAL_readPwmCmpA()
- HAL_readPwmCmpB()
- HAL_readPwmPeriod()

5. Usage in MotorWare

The following picture shows where the HAL object is added within a MotorWare project:



5.1.HAL Global Variables

The following global variables are declared in a MotorWare project related to the HAL module in proj_labxx.c:

```
// the globals

HAL_Handle halHandle;

HAL_PwmData_t gPwmData = {_IQ(0.0), _IQ(0.0), _IQ(0.0)};

HAL_AdcData_t gAdcData;
```

And the global HAL object declared in hal.c:

```
// the globals
```

```
HAL_Obj hal;
```

5.2.HAL Configuration

In a MotorWare project, there are several HAL functions that are called only once, related to the configuration of the Hardware. As an example, consider proj_lab03a of MotorWare Release 13. All of these functions deal with configuration of either a peripheral, or a driver IC (i.e. DRV8301).

```
// initialize the hardware abstraction layer
halHandle = HAL_init(&hal,sizeof(hal));

...

// set the hardware abstraction layer parameters
HAL_setParams(halHandle,&gUserParams);

...

// setup faults
HAL_setupFaults(halHandle);

// initialize the interrupt vector table
HAL_initIntVectorTable(halHandle);

// enable the ADC interrupts
HAL_enableAdcInts(halHandle);

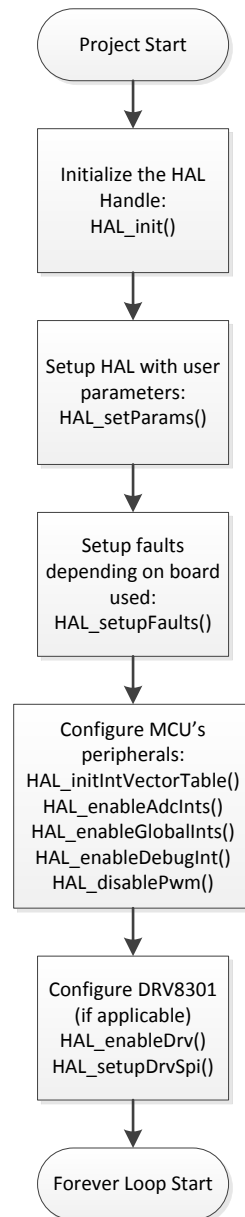
// enable global interrupts
HAL_enableGlobalInts(halHandle);

// enable debug interrupts
HAL_enableDebugInt(halHandle);


// disable the PWM
HAL_disablePwm(halHandle);

#ifdef DRV8301_SPI
// turn on the DRV8301 if present
HAL_enableDrv(halHandle);
// initialize the DRV8301 interface
HAL_setupDrvSpi(halHandle,&gDrvSpi8301Vars);
#endif
```

The following state machine summarizes the HAL configuration function calls:



5.3.HAL Background Functions

Considering the same example, proj_lab03a of MotorWare 13, there are a few functions that are executed in the background loop (commonly called the forever loop, outside of any interrupts). These functions are PWM enable and disable, ADC bias updates and DRV8301 status updates:

```
for(;;)
{
...
    if(CTRL_isError(ctrlHandle))
    {
...
        // disable the PWM
```

```

        HAL_disablePwm(halHandle);
    }
    else
    {
...
        if(ctrlState == CTRL_State_OffLine)
        {
            // enable the PWM
            HAL_enablePwm(halHandle);
        }
        else if(ctrlState == CTRL_State_OnLine)
        {
            // update the ADC bias values
            HAL_updateAdcBias(halHandle);

            // Return the bias value for currents
            gMotorVars.I_bias.value[0] = HAL_getBias(halHandle,HAL_SensorType_Current,0);
            gMotorVars.I_bias.value[1] = HAL_getBias(halHandle,HAL_SensorType_Current,1);
            gMotorVars.I_bias.value[2] = HAL_getBias(halHandle,HAL_SensorType_Current,2);

            // Return the bias value for voltages
            gMotorVars.V_bias.value[0] = HAL_getBias(halHandle,HAL_SensorType_Voltage,0);
            gMotorVars.V_bias.value[1] = HAL_getBias(halHandle,HAL_SensorType_Voltage,1);
            gMotorVars.V_bias.value[2] = HAL_getBias(halHandle,HAL_SensorType_Voltage,2);

            // enable the PWM
            HAL_enablePwm(halHandle);
        }
        else if(ctrlState == CTRL_State_Idle)
        {
            // disable the PWM
            HAL_disablePwm(halHandle);
...
        }
...
    }
}
...
#ifdef DRV8301_SPI
    HAL_writeDrvData(halHandle,&gDrvSpi8301Vars);

    HAL_readDrvData(halHandle,&gDrvSpi8301Vars);
#endif

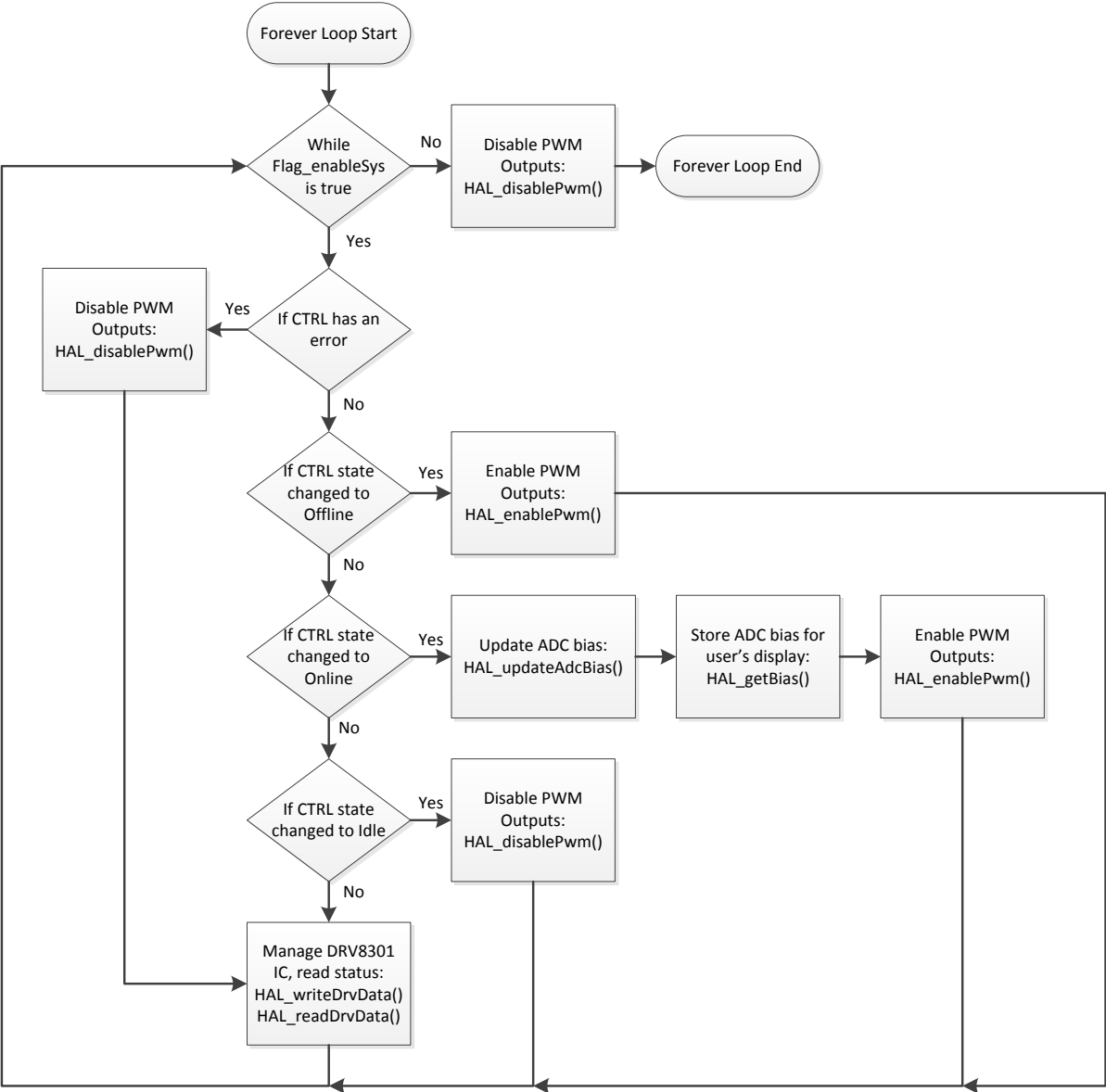
} // end of while(gFlag_enableSys) loop

// disable the PWM

```

```
HAL_disablePwm(halHandle);
...
} // end of for(;;) loop
```

The following state machine summarizes the HAL function calls in the forever loop (outside of the interrupt):



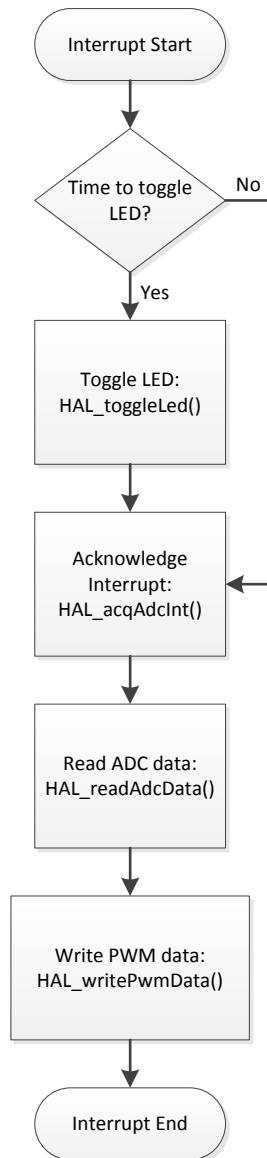
5.4.HAL Functions in Interrupt

The following functions are called in our proj_lab03a example of MotorWare 13.

```
interrupt void mainISR(void)
{
...
}
```

```
    HAL_toggleLed(halHandle,(GPIO_Number_e)HAL_Gpio_LED2);  
...  
    // acknowledge the ADC interrupt  
    HAL_acqAdcInt(halHandle,ADC_IntNumber_1);  
  
    // convert the ADC data  
    HAL_readAdcData(halHandle,&gAdcData);  
...  
    // write the PWM compare values  
    HAL_writePwmData(halHandle,&gPwmData);  
...  
    return;  
} // end of mainISR() function
```

The following state machine summarizes the HAL function calls in the interrupt:

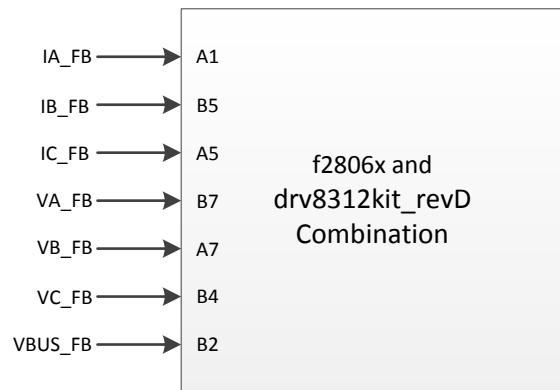


6. Examples of Modifying the HAL Module

This section shows a few examples on how to change the HAL module. The examples shown here are tested with MotorWare release 13.

6.1.Changing ADC Channels

The HAL module configures the ADC channels according to the board of interest. For example, the f2806x and the drv8312kit_revD board combination uses the following ADC connections:



Which is configured as shown in the following code snippet, taken from hal.c, function HAL_setupAdcs() located in the following location: sw\modules\hal\boards\drv8312kit_revD\f28x\f2806x\src

```
// configure the SOC's for drv8312kit_revD
// EXT IA-FB
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_0,ADC_SocChanNumber_A1);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_0,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_0,ADC_SocSampleDelay_9_cycles);

// EXT IA-FB
// Duplicate conversion due to ADC Initial Conversion bug (SPRZ342)
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_1,ADC_SocChanNumber_A1);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_1,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_1,ADC_SocSampleDelay_9_cycles);

// EXT IB-FB
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_2,ADC_SocChanNumber_B5);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_2,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_2,ADC_SocSampleDelay_9_cycles);

// EXT IC-FB
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_3,ADC_SocChanNumber_A5);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_3,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_3,ADC_SocSampleDelay_9_cycles);

// ADC-Vhb1
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_4,ADC_SocChanNumber_B7);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_4,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_4,ADC_SocSampleDelay_9_cycles);

// ADC-Vhb2
```

```

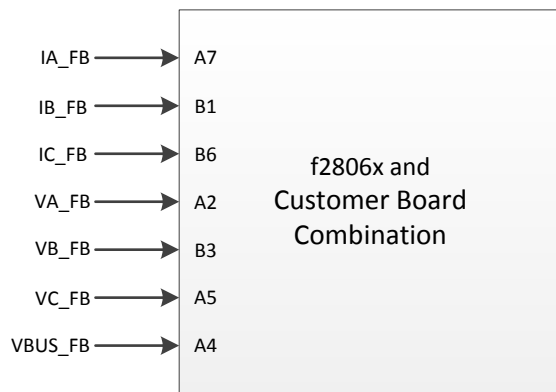
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_5,ADC_SocChanNumber_A7);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_5,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_5,ADC_SocSampleDelay_9_cycles);

// ADC-Vhb3
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_6,ADC_SocChanNumber_B4);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_6,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_6,ADC_SocSampleDelay_9_cycles);

// VDCBUS
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_7,ADC_SocChanNumber_B2);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_7,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_7,ADC_SocSampleDelay_9_cycles);

```

Now let's change the connections, as if we had a customer's board, which has a different ADC configuration as in the following diagram:



The only piece of code that's affected is in the HAL module. The configuration of the ADC, done in function HAL_setupAdcs() would be like this (changes are highlighted):

```

// configure the SOCs for customer board
// EXT IA-FB
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_0,ADC_SocChanNumber_A7);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_0,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_0,ADC_SocSampleDelay_9_cycles);

// EXT IA-FB
// Duplicate conversion due to ADC Initial Conversion bug (SPRZ342)
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_1,ADC_SocChanNumber_A7);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_1,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_1,ADC_SocSampleDelay_9_cycles);

```

```

// EXT IB-FB
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_2,ADC_SocChanNumber_B1);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_2,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_2,ADC_SocSampleDelay_9_cycles);

// EXT IC-FB
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_3,ADC_SocChanNumber_B6);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_3,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_3,ADC_SocSampleDelay_9_cycles);

// ADC-Vhb1
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_4,ADC_SocChanNumber_A2);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_4,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_4,ADC_SocSampleDelay_9_cycles);

// ADC-Vhb2
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_5,ADC_SocChanNumber_B3);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_5,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_5,ADC_SocSampleDelay_9_cycles);

// ADC-Vhb3
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_6,ADC_SocChanNumber_A5);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_6,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_6,ADC_SocSampleDelay_9_cycles);

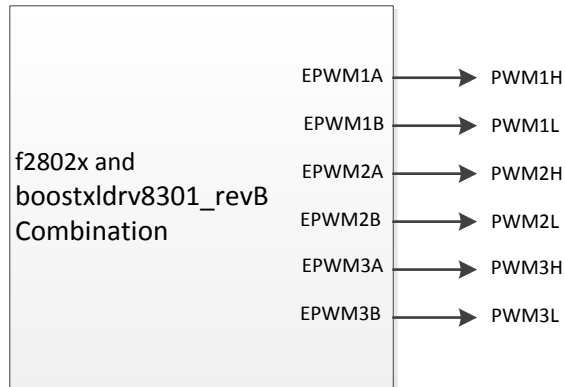
// VDCBUS
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_7,ADC_SocChanNumber_A4);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_7,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_7,ADC_SocSampleDelay_9_cycles);

```

Keep in mind that the first two channels need to be repeated, so that the ADC initial conversion bug (SPRZ342) is handled.

6.2.Changing PWM Channels

Similar to the previous example, the PWM outputs can also be changed to accommodate another board compared to the TI motor control kits. For example, if we consider the booster pack + Launchpad and f2802x combination, this is how the PWM outputs are connected:



There are two functions that configure the PWM connections. One is in HAL_setupGpios() to setup the GPIOs as PWM outputs:

```
// PWM1H
GPIO_setMode(obj->gpioHandle,GPIO_Number_0,GPIO_0_Mode_EPWM1A);

// PWM1L
GPIO_setMode(obj->gpioHandle,GPIO_Number_1,GPIO_1_Mode_EPWM1B);

// PWM2H
GPIO_setMode(obj->gpioHandle,GPIO_Number_2,GPIO_2_Mode_EPWM2A);

// PWM2L
GPIO_setMode(obj->gpioHandle,GPIO_Number_3,GPIO_3_Mode_EPWM2B);

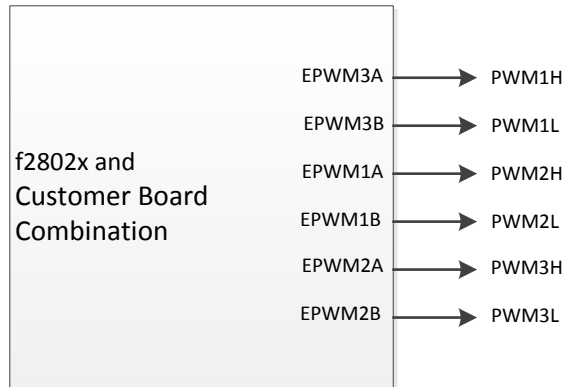
// PWM3H
GPIO_setMode(obj->gpioHandle,GPIO_Number_4,GPIO_4_Mode_EPWM3A);

// PWM3L
GPIO_setMode(obj->gpioHandle,GPIO_Number_5,GPIO_5_Mode_EPWM3B);
```

And the second function that handles this configuration is HAL_init(), which assigns the address of the PWM modules used:

```
// initialize PWM handle
obj->pwmHandle[0] = PWM_init((void *)PWM_ePWM1_BASE_ADDR,sizeof(PWM_Obj));
obj->pwmHandle[1] = PWM_init((void *)PWM_ePWM2_BASE_ADDR,sizeof(PWM_Obj));
obj->pwmHandle[2] = PWM_init((void *)PWM_ePWM3_BASE_ADDR,sizeof(PWM_Obj));
```

Now, consider the following customer's board connection instead:



The following code snippet shows how to enable these GPIOs as outputs in HAL_setupGpios(). In fact, the configuration is the same, although the comments are changed reflecting new order of connections.

```
// PWM2H
GPIO_setMode(obj->gpioHandle,GPIO_Number_0,GPIO_0_Mode_EPWM1A);

// PWM2L
GPIO_setMode(obj->gpioHandle,GPIO_Number_1,GPIO_1_Mode_EPWM1B);

// PWM3H
GPIO_setMode(obj->gpioHandle,GPIO_Number_2,GPIO_2_Mode_EPWM2A);

// PWM3L
GPIO_setMode(obj->gpioHandle,GPIO_Number_3,GPIO_3_Mode_EPWM2B);

// PWM1H
GPIO_setMode(obj->gpioHandle,GPIO_Number_4,GPIO_4_Mode_EPWM3A);

// PWM1L
GPIO_setMode(obj->gpioHandle,GPIO_Number_5,GPIO_5_Mode_EPWM3B);
```

A second function that handles this configuration is HAL_init(). The assigned addresses are changed according to the PWM output connections:

```
// initialize PWM handle
obj->pwmHandle[0] = PWM_init((void *)PWM_ePWM3_BASE_ADDR,sizeof(PWM_Obj));
obj->pwmHandle[1] = PWM_init((void *)PWM_ePWM1_BASE_ADDR,sizeof(PWM_Obj));
obj->pwmHandle[2] = PWM_init((void *)PWM_ePWM2_BASE_ADDR,sizeof(PWM_Obj));
```

This way, every time we reference “obj->pwmHandle[PWM_Number_1]” we will be referencing outputs PWM1H and PWM1L of the actual inverter, regardless of what EPWM it is assigned to. Similarly when we reference “obj->pwmHandle[PWM_Number_2]” we will be referencing outputs PWM2H and PWM2L, and when referencing “obj->pwmHandle[PWM_Number_1]” we will be referencing outputs PWM3H and PWM3L. So in summary:

- pwmHandle[PWM_Number_1] → f2802x: EPWM3A/EPWM3B → Inverter: PWM1H/PWM1L
- pwmHandle[PWM_Number_2] → f2802x: EPWM1A/EPWM1B → Inverter: PWM2H/PWM2L
- pwmHandle[PWM_Number_3] → f2802x: EPWM2A/EPWM2B → Inverter: PWM3H/PWM3L

Finally, the start of conversion of the ADC samples needs to be changed in HAL_setupAdcs() function, since now the conversions will be triggered by Inverter outputs PWM1H/PWM1L, which are now assigned to f2802x: EPWM3A/EPWM3B. To change this, the following code snippet shows a highlighter text, which changes from a value of “ADC_SocTrigSrc_EPWM1_ADCSOCA” to a value of “ADC_SocTrigSrc_EPWM3_ADCSOCA”:

```
//configure the SOCs for customer board
// sample the first sample twice due to errata sprz342f
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_0,ADC_SocChanNumber_B1);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_0,ADC_SocTrigSrc_EPWM3_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_0,ADC_SocSampleDelay_7_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_1,ADC_SocChanNumber_B1);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_1,ADC_SocTrigSrc_EPWM3_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_1,ADC_SocSampleDelay_7_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_2,ADC_SocChanNumber_B3);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_2,ADC_SocTrigSrc_EPWM3_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_2,ADC_SocSampleDelay_7_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_3,ADC_SocChanNumber_B7);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_3,ADC_SocTrigSrc_EPWM3_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_3,ADC_SocSampleDelay_7_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_4,ADC_SocChanNumber_A3);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_4,ADC_SocTrigSrc_EPWM3_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_4,ADC_SocSampleDelay_7_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_5,ADC_SocChanNumber_A1);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_5,ADC_SocTrigSrc_EPWM3_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_5,ADC_SocSampleDelay_7_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_6,ADC_SocChanNumber_A0);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_6,ADC_SocTrigSrc_EPWM3_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_6,ADC_SocSampleDelay_7_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_7,ADC_SocChanNumber_A7);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_7,ADC_SocTrigSrc_EPWM3_ADCSOCA);
```

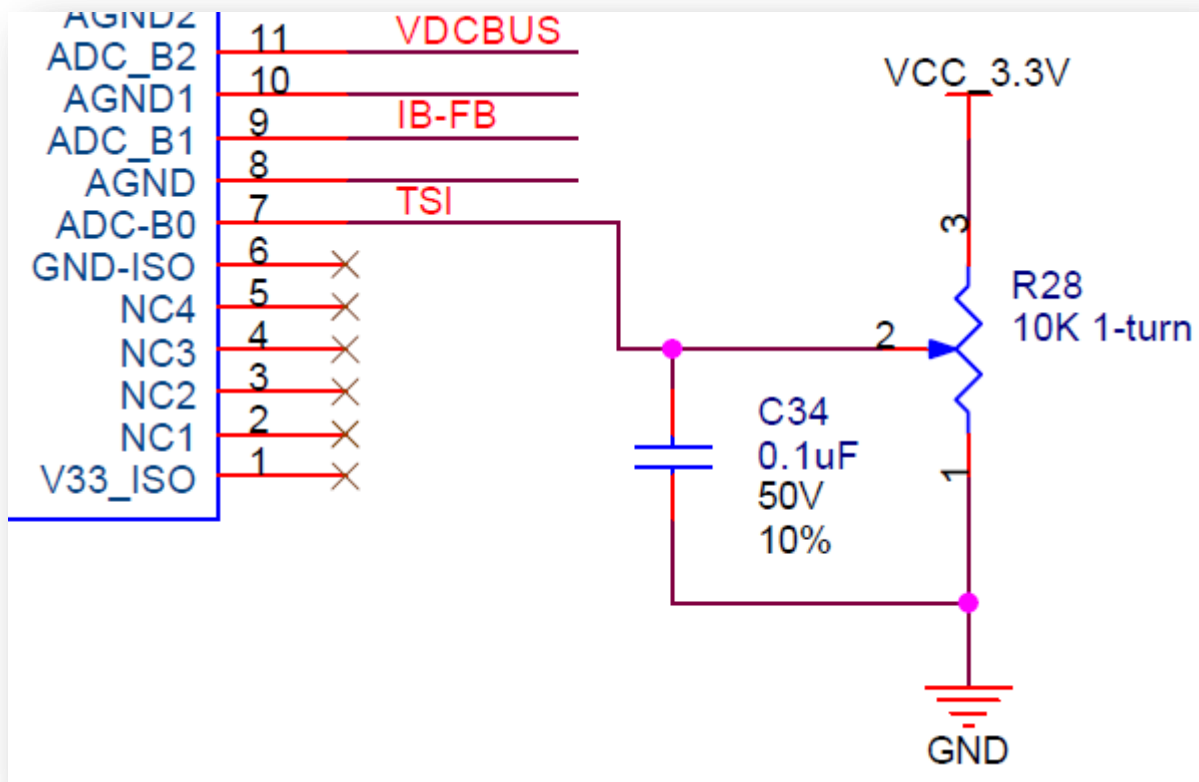
```
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_7,ADC_SocSampleDelay_7_cycles);
```

6.3.Adding a Potentiometer Read

As an example, let us change the HAL module so that we can add a potentiometer read. For this example, we will assume we have an f2806x and a drv8301kit_revD combination. Let us also assume that we are running proj_lab03a of MotorWare 13. So the files that will change in this case are:

- sw\modules\hal\boards\drv8301kit_revD\f28x\f2806x\src\hal.h
- sw\modules\hal\boards\drv8301kit_revD\f28x\f2806x\src\hal.c
- sw\solutions\instaspin_foc\src\proj_lab03a.c

First of all, we need to configure a start of conversion channel. We will assign the start of conversion 8 to this potentiometer we want to read. The potentiometer we would like to read is connected to ADC-B0 according to drv8301kit_revD schematics as shown in the following picture:

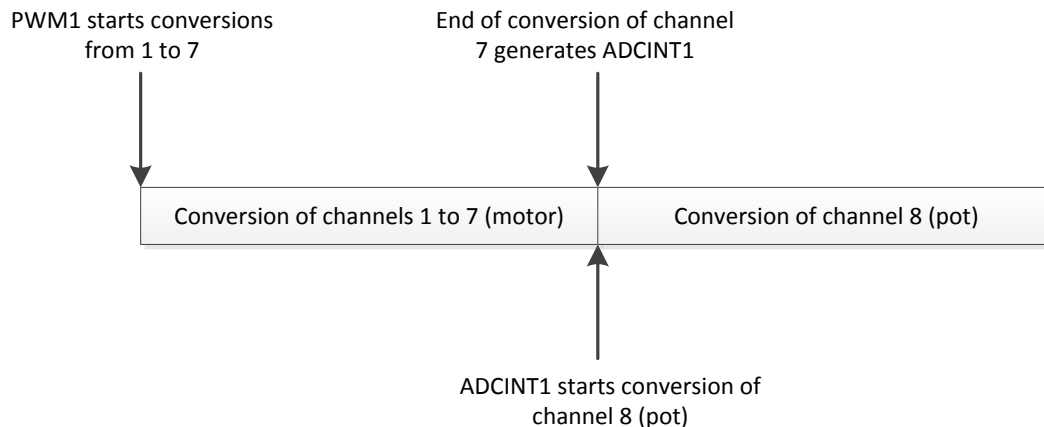


This is done in function HAL_setupAdcs() of hal.c file as follows:

```
// Potentiometer
// Configure it so that ADCINT1 will trigger a potentiometer conversion
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_8,ADC_SocChanNumber_B0);
```

```
ADC_setupSocTrigSrc(obj->adcHandle, ADC_SocNumber_8, ADC_Int1TriggersSOC);
ADC_setSocSampleDelay(obj->adcHandle, ADC_SocNumber_8, ADC_SocSampleDelay_9_cycles);
```

It is very important to use ADCINT1 as the trigger source, since we don't want the potentiometer conversion to delay motor control related conversions. Since ADCINT1 is triggered by the end of conversion of the other 7 channels (dedicated to motor control), the potentiometer will always be triggered once the time critical channels are converted. Looking at this in a time diagram would look something like this:



The second change is in hal.h file. A new function is created to read the potentiometer. This function simply reads the ADC result register and scales the value to an IQ24 value. This function is listed here:

```
//! \brief Reads the Potentiometer
//! \param[in] handle The hardware abstraction layer (HAL) handle
//! \return The potentiometer value from _IQ(-1.0) to _IQ(1.0)
static inline _iq HAL_readPotentiometerData(HAL_Handle handle)
{
    HAL_Obj *obj = (HAL_Obj *)handle;

    _iq value;

    // convert potentiometer from IQ12 to IQ24.
    value = _IQ12toIQ((_iq)ADC_readResult(obj->adcHandle, ADC_ResultNumber_8));

    return(value);
} // end of HAL_readPotentiometerData() function
```

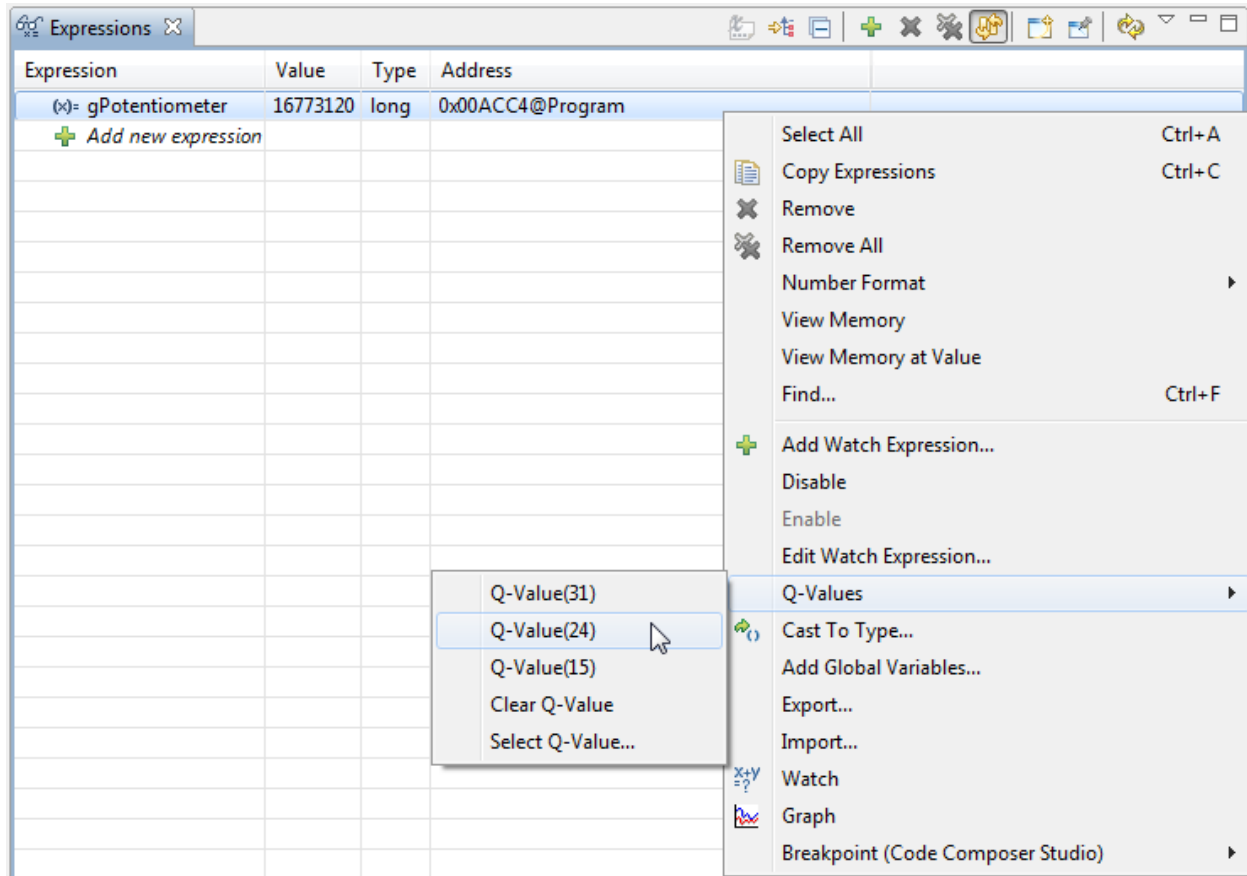
The third change is in the project itself. Since we are working on proj_lab03a, we only need to declare a global variable to store the converted value:

```
_iq gPotentiometer = _IQ(0.0);
```

And we call the new function we created in a background loop, outside of the interrupt to avoid any CPU bandwidth hit.

```
gPotentiometer = HAL_readPotentiometerData(halHandle);
```

User can confirm that the value is being updated by adding the new global variable to the watch window. The value type needs to be changed to IQ24 by right clicking the variable name as shown in the following screen shot:



If we move the potentiometer all the way CCW, we see a value of 0.0 being displayed:

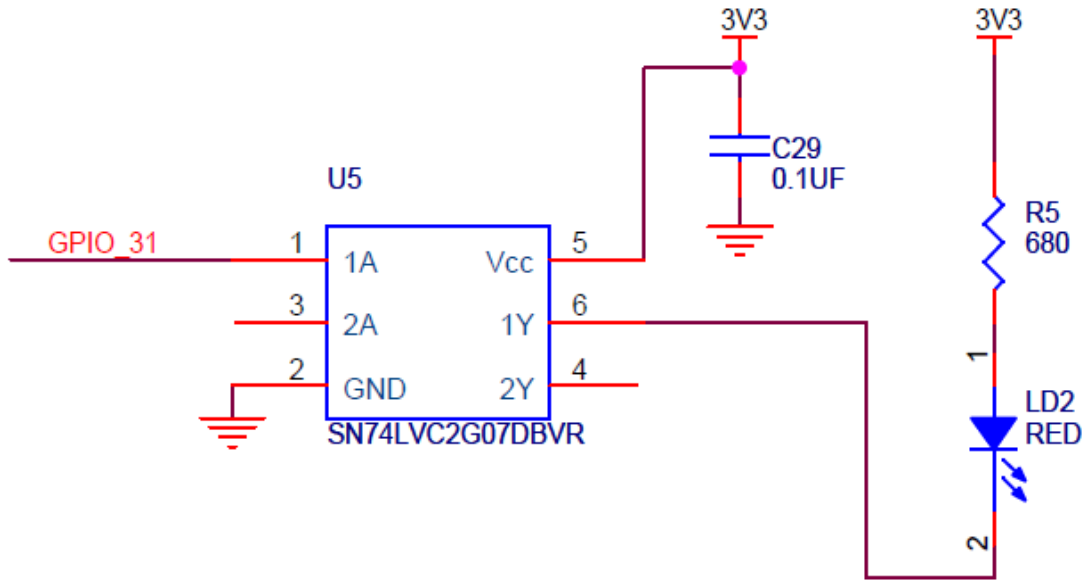
```
(x)- gPotentiometer 0.0 (Q-Value(24))
```

And if we move it all the way to CW:

```
(x)- gPotentiometer 0.9997558594 (Q-Value(24))
```

6.4.Adding a User PWM Output

This is another example, where a simple PWM output is added to the HAL module. Let us use the f2806x with the drv8301kit_revD combination. We will use GPIO 31 as a PWM output, which drives the LED labeled as "LD2" in the control card.



Our goal is to have a PWM with fixed frequency, and a global variable for duty cycle. The following files are going to be affected based on proj_lab03a MotorWare release 13:

- sw\modules\hal\boards\drv8301kit_revD\f28x\f2806x\src\hal.c
- sw\modules\hal\boards\drv8301kit_revD\f28x\f2806x\src\hal.h
- sw\modules\hal\boards\drv8301kit_revD\f28x\f2806x\src\hal_obj.h
- sw\solutions\instaspin_foc\src\proj_lab03a.c

6.4.1. Adding a PWM Handle to the HAL Object

The first step is to add a handle of a PWM user in the HAL object.

Important Note: Additional members to the HAL object must be added to the end of the object definition. This is because an instantiation of the HAL object was built for ROM, and the ROM code expects the same object definition. Additional members of the HAL layer added at the bottom of the HAL object won't conflict with the ROMed code.

Having said that, this code example shows the new member added to the HAL object:

```

//! \brief   Defines the hardware abstraction layer (HAL) data
//! \details The HAL object contains all handles to peripherals. When accessing a
//!           peripheral on a processor, use a HAL function along with the HAL handle
//!           for that processor to access its peripherals.
//!
typedef struct _HAL_Obj_

```

```

{
    ...

    PWM_Handle  pwmUserHandle;  //!< the PWM user handle

} HAL_Obj;

```

6.4.2. Initializing the New Handle

The following code is needed to initialize the new handle, that is, to have a handle point to the peripheral address, so every time we reference this handle, the peripheral itself is referenced.

```

HAL_Handle HAL_init(void *pMemory,const size_t numBytes)
{
    ...
    // initialize PWM user handle
    obj->pwmUserHandle = PWM_init((void *)PWM_ePWM8_BASE_ADDR,sizeof(PWM_Obj));

    return(handle);
} // end of HAL_init() function

```

6.4.3. Create a Setup Function for the new PWM

The third step is to create a function to setup the PWM. The function prototype in hal.h file looks like this:

```

//! \brief   Sets up the PWM (Pulse Width Modulator) for user control
//! \param[in] handle      The hardware abstraction layer (HAL) handle
//! \param[in] systemFreq_MHz  The system frequency, MHz
//! \param[in] pwmPeriod_usec  The PWM period, usec
extern void HAL_setupPwmUser(HAL_Handle handle,
                             const uint_least16_t systemFreq_MHz,
                             const float_t pwmPeriod_usec);

```

Notice that this function takes the CPU frequency in MHz and the desired PWM period in microseconds. This is really a user's choice, but in this example, it is convenient to specify the PWM period depending on the CPU frequency. The function itself, in hal.c, is listed here:

```

void HAL_setupPwmUser(HAL_Handle handle,
                     const uint_least16_t systemFreq_MHz,
                     const float_t pwmPeriod_usec)
{
    HAL_Obj  *obj = (HAL_Obj *)handle;
    uint16_t period_cycles = (uint16_t)((float_t)systemFreq_MHz*pwmPeriod_usec);
}

```



```

// turns off the output of the EPWM peripheral
PWM_setOneShotTrip(obj->pwmUserHandle);

// setup the Time-Base Control Register (TBCTL)
PWM_setCounterMode(obj->pwmUserHandle,PWM_CounterMode_Up);
PWM_disableCounterLoad(obj->pwmUserHandle);
PWM_setPeriodLoad(obj->pwmUserHandle,PWM_PeriodLoad_Immediate);
PWM_setSyncMode(obj->pwmUserHandle,PWM_SyncMode_Disable);
PWM_setHighSpeedClkDiv(obj->pwmUserHandle,PWM_HspClkDiv_by_1);
PWM_setClkDiv(obj->pwmUserHandle,PWM_ClkDiv_by_1);
PWM_setPhaseDir(obj->pwmUserHandle,PWM_PhaseDir_CountUp);
PWM_setRunMode(obj->pwmUserHandle,PWM_RunMode_FreeRun);

// setup the Timer-Based Phase Register (TBPHS)
PWM_setPhase(obj->pwmUserHandle,0);

// setup the Time-Base Counter Register (TBCTR)
PWM_setCount(obj->pwmUserHandle,0);

// setup the Time-Base Period Register (TBPRD)
// set to zero initially
PWM_setPeriod(obj->pwmUserHandle,0);

// setup the Counter-Compare Control Register (CMPCTL)
PWM_setLoadMode_CmpA(obj->pwmUserHandle,PWM_LoadMode_Zero);
PWM_setShadowMode_CmpA(obj->pwmUserHandle,PWM_ShadowMode_Shadow);

// setup the Action-Qualifier Output A Register (AQCTLA)
PWM_setActionQual_CntUp_CmpA_PwmA(obj->pwmUserHandle,PWM_ActionQual_Clear);
PWM_setActionQual_Period_PwmA(obj->pwmUserHandle,PWM_ActionQual_Set);

// setup the Dead-Band Generator Control Register (DBCTL)
PWM_setDeadBandOutputMode(obj->pwmUserHandle,PWM_DeadBandOutputMode_Bypass);

// setup the PWM-Chopper Control Register (PCCTL)
PWM_disableChopping(obj->pwmUserHandle);

// setup the Trip Zone Select Register (TZSEL)
PWM_disableTripZones(obj->pwmUserHandle);

```

```

// setup the Event Trigger Selection Register (ETSEL)
PWM_disableInt(obj->pwmUserHandle);
PWM_disableSocAPulse(obj->pwmUserHandle);

// setup the Event Trigger Prescale Register (ETPS)
PWM_setIntPeriod(obj->pwmUserHandle,PWM_IntPeriod_FirstEvent);
PWM_setSocAPeriod(obj->pwmUserHandle,PWM_SocPeriod_FirstEvent);

// setup the Event Trigger Clear Register (ETCLR)
PWM_clearIntFlag(obj->pwmUserHandle);
PWM_clearSocAFlag(obj->pwmUserHandle);

// since the PWM is configured as an up counter, the period register is set to
// the desired PWM period
PWM_setPeriod(obj->pwmUserHandle,period_cycles);

// turns on the output of the EPWM peripheral
PWM_clearOneShotTrip(obj->pwmUserHandle);

return;
} // end of HAL_setupPwmUser() function

```

The PWM is configured as an up counter only (commonly known as edge aligned), with positive duty cycle (the higher the duty cycle, the wider the high side is), with only one output (the A output, so no dead band is needed).

6.4.4. Create a Write Function

We need to create a HAL function that will write the CMPA register of the PWM with the correct scaling. The following example function was created, which takes a duty cycle as a parameter, from _IQ(0.0) (representing 0.0%) up to _IQ(1.0) (representing 100.0% duty cycle).

```

//! \brief   Writes PWM data to the PWM comparator for user control
//! \param[in] handle   The hardware abstraction layer (HAL) handle
//! \param[in] dutyCycle The duty cycle to be written, from _IQ(0.0) to _IQ(1.0)
static inline void HAL_writePwmDataUser(HAL_Handle handle,_iq dutyCycle)
{
    HAL_Obj *obj = (HAL_Obj *)handle;
    PWM_Obj *pwm;
    _iq period;
    _iq value;
    uint16_t value_sat;

```

```

pwm = (PWM_Obj *)obj->pwmUserHandle;
period = (_iq)pwm->TBPRD;
value = _IQmpy(dutyCycle, period);
value_sat = (uint16_t)_IQsat(value, period, _IQ(0.0));

// write the PWM data
PWM_write_CmpA(obj->pwmUserHandle,value_sat);

return;
} // end of HAL_writePwmDataUser() function

```

6.4.5. Configuring GPIO as PWM Output

The following code configures GPIO 31 as a PWM output. This code is in hal.c, in HAL_setupGpios() function.

```

// ControlCARD LED2
GPIO_setMode(obj->gpioHandle,GPIO_Number_31,GPIO_31_Mode_EPWM8A);

```

6.4.6. Enable PWM Clock

This is done in HAL_setupPeripheralClks() function of hal.c file:

```

CLK_enablePwmClock(obj->clkHandle,PWM_Number_8);

```

6.4.7. Calling the Setup Function

We need to call the setup function we created, so that the PWM is configured. This is done when the HAL object is setup, in the HAL_setParams() function in hal.c file:

```

// setup the PWM User
HAL_setupPwmUser(handle,90,(float_t)100.0);

```

As can be seen, the CPU is running at 90 MHz, and our desired PWM period is 100 microseconds, or 10 kHz frequency.

6.4.8. Changing Duty Cycle in the Project

Finally, we create a global variable that we use to change the duty cycle. We add this global variable to proj_lab03a.c, and initialize it with some duty cycle:

```

_iq gLEDduty = _IQ(0.9);

```

Then, we call the function we created inside the project. This can be done anywhere in the code, as long as it is periodically called, so the duty cycle is updated.

```

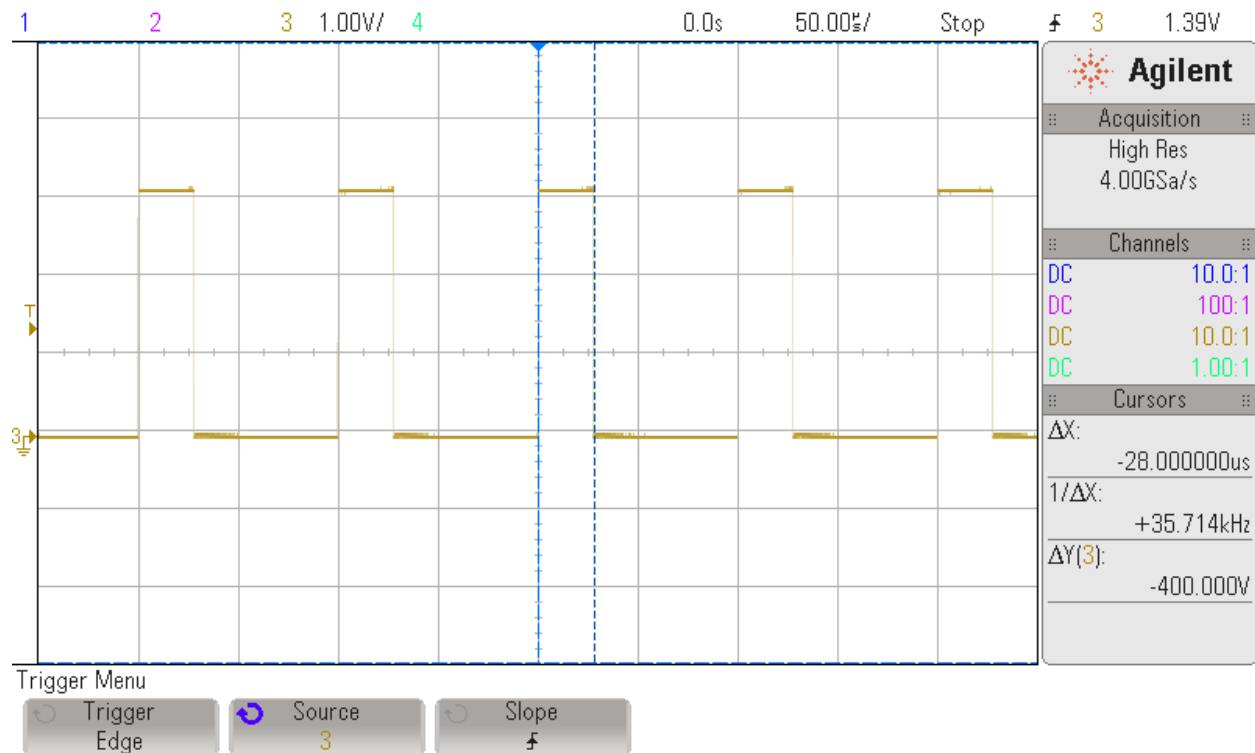
HAL_writePwmDataUser(halHandle,gLEDduty);

```

To verify, plot the PWM output in the scope. For example, in our test, we changed the duty cycle to 28%, so:

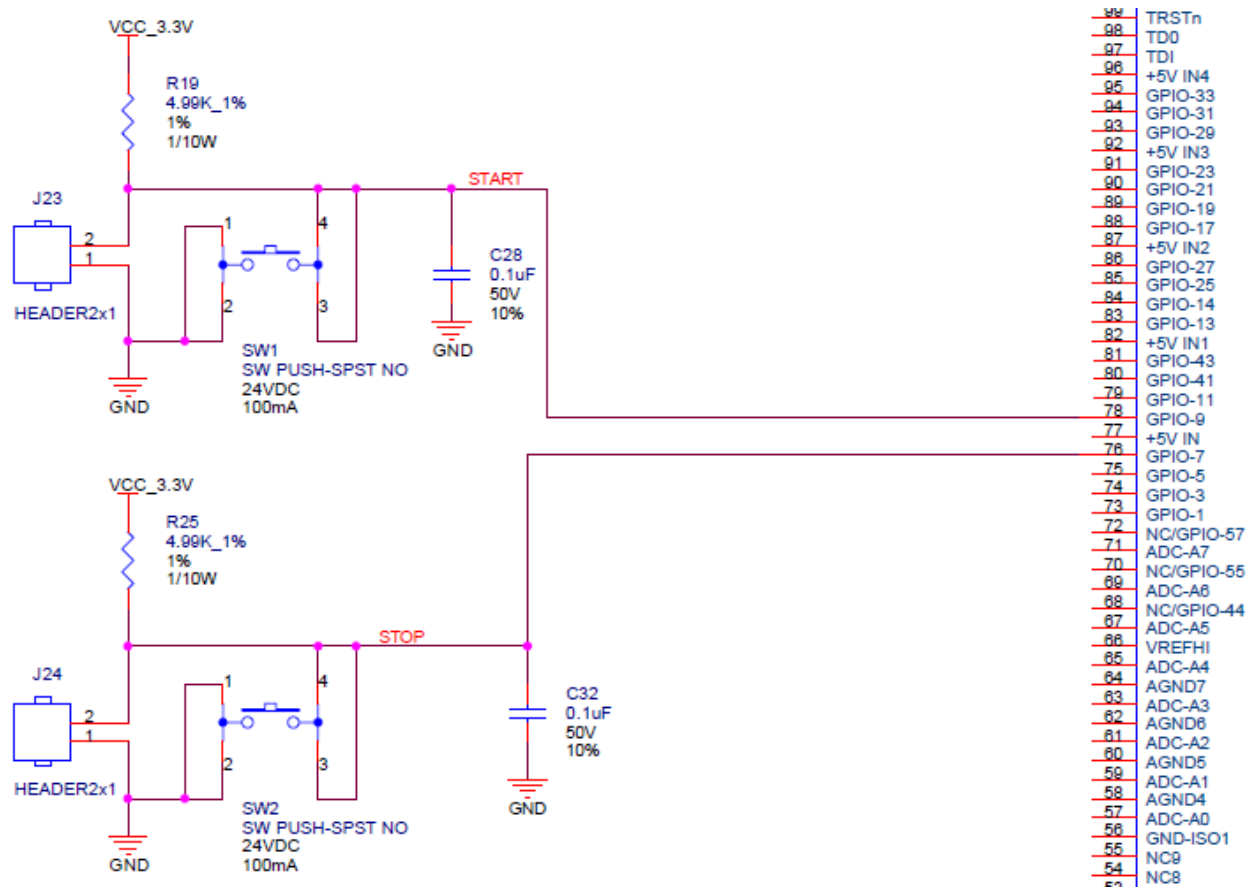
(*)= gLEDduty 0.2799999714 (Q-Value(24))

And this is the PWM waveform, where it can be seen that the period is what we set it to, 100 microseconds, and the 28% of that is 28 microseconds as shown in the scope below:



6.5. Reading Push Buttons

It is often useful to read push buttons to allow a motor to run, stop, or simply to change the state of a global variable when pushing a button. Using a combination of f2806x and drv8301kit_revD, and proj_lab03a of MotorWare 13, let's add a few lines of code to read the two push buttons available in this board. First of all, we need to see where those push buttons are connected to. The schematics showing where these two switches are connected:



As can be seen, SW2 is connected to GPIO-7 and SW1 is connected to GPIO-9. We need to configure the GPIOs in HAL_setupGpios() function of hal.c file to allow those pins to be inputs:

```
// Push Button SW2
GPIO_setMode(obj->gpioHandle,GPIO_Number_7,GPIO_7_Mode_GeneralPurpose);

// Push Button SW1
GPIO_setMode(obj->gpioHandle,GPIO_Number_9,GPIO_9_Mode_GeneralPurpose);
```

We also need to define a function in hal.h that reads a GPIO as follows:

```
///  
//! \brief Reads the specified GPIO pin  
//! \details Takes in the enumeration GPIO_Number_e and reads that GPIO  
//! \param[in] handle The hardware abstraction layer (HAL) handle  
//! \param[in] gpioNumber The GPIO number  
static inline bool HAL_readGpio(HAL_Handle handle,const GPIO_Number_e gpioNumber)  
{  
    HAL_Obj *obj = (HAL_Obj *)handle;  
  
    // read GPIO
```

```
return(GPIO_read(obj->gpioHandle,gpioNumber));  
} // end of HAL_readGpio() function
```

Then, we add two HAL_ definitions that link a switch with a specific GPIO. This is done so that in the top level source file (in this case proj_lab03a.c) there is no reference to specific GPIO numbers. The reason for this is to allow the top level code to be fully portable to other platforms, where the push buttons might be connected to different GPIOs.

```
//! \brief Defines the GPIO pin number for drv8301kit_revD Switch 1  
//!  
#define HAL_GPIO_SW1 GPIO_Number_9  
  
//! \brief Defines the GPIO pin number for drv8301kit_revD Switch 2  
//!  
#define HAL_GPIO_SW2 GPIO_Number_7
```

From the project main file, we need to add two global variables as follows, where we will store the state of the push buttons:

```
bool gSw1;  
  
bool gSw2;
```

The final step is to call the function we created, and assign the returned value in our new global variables:

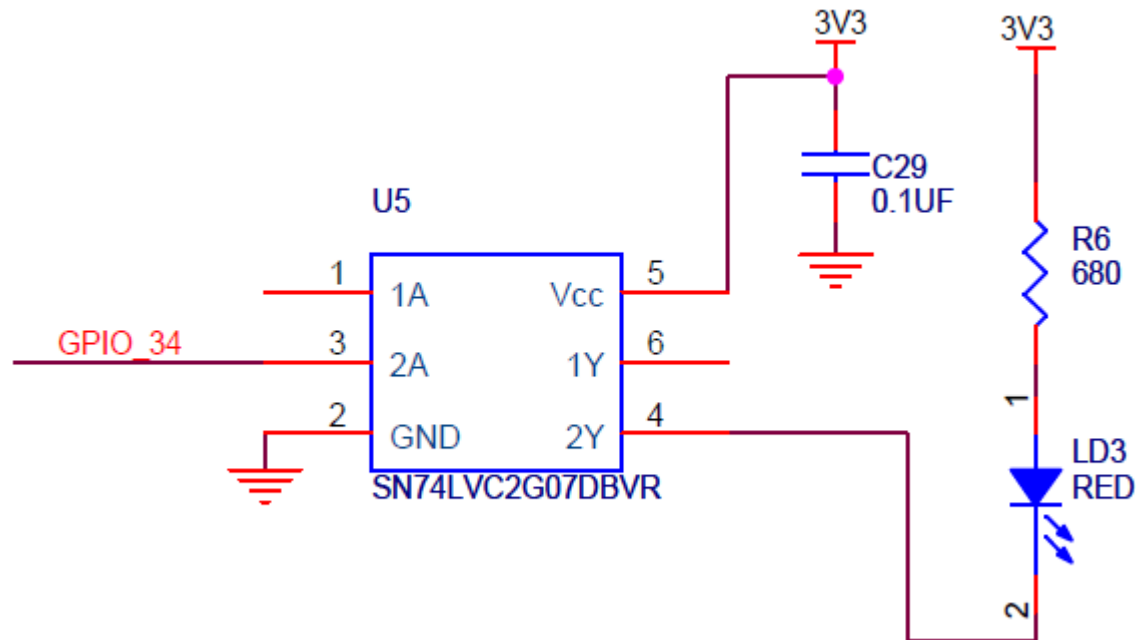
```
gSw1 = HAL_readGpio(halHandle, HAL_GPIO_SW1);  
  
gSw2 = HAL_readGpio(halHandle, HAL_GPIO_SW2);
```

If we add these global variables to the watch window, we will see that when we press the buttons, the state of the variables will change:

(x)= gSw1	0
(x)= gSw2	0

6.6.Adding a Timer Interrupt

In this example, we will enable Timer 0 interrupt, and generate an LED toggle every second. We will use the f2806x and drv8301kit_revD combination, and proj_lab03a of MotorWare 13. The following LED is used for the example, the one labeled “LD3” in the f2806x control card.



For this example, the following files will be updated:

- sw\drivers\pie\src\32b\f28x\f2806x\pie.c
- sw\drivers\pie\src\32b\f28x\f2806x\pie.h
- sw\modules\hal\boards\drv8301kit_revD\f28x\f2806x\src\hal.c
- sw\modules\hal\boards\drv8301kit_revD\f28x\f2806x\src\hal.h
- sw\solutions\instaspin_foc\src\proj_lab03a.c

6.6.1. Configuring the GPIO as an Output

The following code from HAL_setupGpios() function in hal.c shows how GPIO 34 is configured as an output pin:

```
// ControlCARD LED3
GPIO_setMode(obj->gpioHandle,GPIO_Number_34,GPIO_34_Mode_GeneralPurpose);
GPIO_setLow(obj->gpioHandle,GPIO_Number_34);
GPIO_setDirection(obj->gpioHandle,GPIO_Number_34,GPIO_Direction_Output);
```

6.6.2. Configuring Timer 0 to Have a 1 Second Period

The following function is taken from hal.c, function HAL_setupTimers(), showing that Timer 0 is already configured to have a 1 second period, since it takes the system frequency, and multiplies it times 1E6. So if the processor is running at 90 MHz, then the timer period counts would be $90 * 1,000,000 - 1 = 89,999,999$. This represents a period of $(89,999,999 + 1) / 90 \text{ MHz} = 1 \text{ second}$.

```
void HAL_setupTimers(HAL_Handle handle,const uint_least16_t systemFreq_MHz)
{
    HAL_Obj *obj = (HAL_Obj *)handle;
    uint32_t timerPeriod_cnts = ((uint32_t)systemFreq_MHz * 1000000) - 1;
```

```

// use timer 0 for frequency diagnostics
TIMER_setDecimationFactor(obj->timerHandle[0],0);
TIMER_setEmulationMode(obj->timerHandle[0],TIMER_EmulationMode_RunFree);
TIMER_setPeriod(obj->timerHandle[0],timerPeriod_cnts);
TIMER_setPreScaler(obj->timerHandle[0],0);

// use timer 1 for CPU usage diagnostics
TIMER_setDecimationFactor(obj->timerHandle[1],0);
TIMER_setEmulationMode(obj->timerHandle[1],TIMER_EmulationMode_RunFree);
TIMER_setPeriod(obj->timerHandle[1],timerPeriod_cnts);
TIMER_setPreScaler(obj->timerHandle[1],0);

return;
} // end of HAL_setupTimers() function

```

6.6.3. Create a PIE Function that Enables Timer 0 Interrupts

The following function and prototypes were created in pie.c and pie.h to enable Timer 0 interrupts.

Function prototype in pie.h:

```

/*! \brief Enables the Timer 0 interrupt
 *! \param[in] pieHandle The peripheral interrupt expansion (PIE) handle
extern void PIE_enableTimer0Int(PIE_Handle pieHandle);

```

Function implementation in pie.c, enabling interrupts in group 1, bit 7:

```

void PIE_enableTimer0Int(PIE_Handle pieHandle)
{
    PIE_Obj *pie = (PIE_Obj *)pieHandle;
    uint16_t index = 0;
    uint16_t setValue = (1 << 6);

    // set the value
    pie->PIEIER_PIEIFR[index].IER |= setValue;

    return;
} // end of PIE_enableTimer0Int() function

```

6.6.4. Create a HAL Function that Enables Timer 0 Interrupts

Besides the interrupt enable in the PIE module, we need to create an interrupt enable in the HAL module, the calls the PIE module interrupt enable function, and sets other flags such as CPU interrupt flags, timer register flags, etc.

Function prototype in hal.h:

```
//! \brief Enables the Timer 0 interrupt
//! \param[in] handle The hardware abstraction layer (HAL) handle
extern void HAL_enableTimer0Int(HAL_Handle handle);
```

Function implementation in hal.c:

```
void HAL_enableTimer0Int(HAL_Handle handle)
{
    HAL_Obj *obj = (HAL_Obj *)handle;

    PIE_enableTimer0Int(obj->pieHandle);

    // enable the interrupt
    TIMER_enableInt(obj->timerHandle[0]);

    // enable the cpu interrupt for TINT0
    CPU_enableInt(obj->cpuHandle,CPU_IntNumber_1);

    return;
} // end of HAL_enablePwmInt() function
```

The new function HAL_enableTimer0Int() needs to be called in the main project source file, so in proj_lab03a.c we add this before the forever loop:

```
// enable the Timer 0 interrupts
HAL_enableTimer0Int(halHandle);
```

6.6.5. Create a HAL Function that Acknowledges Timer 0 Interrupts

We also need to create a function in the HAL module that clears the appropriate flags so that the Timer 0 interrupt can happen again. This function will be implemented as an inline function in hal.h as follows:

```
//! \brief Acknowledges an interrupt from Timer 0 so that another Timer 0 interrupt can
//! happen again.
//! \param[in] handle The hardware abstraction layer (HAL) handle
static inline void HAL_acqTimer0Int(HAL_Handle handle)
{
    HAL_Obj *obj = (HAL_Obj *)handle;
```

```

// clear the Timer 0 interrupt flag
TIMER_clearFlag(obj->timerHandle[0]);

// Acknowledge interrupt from PIE group 1
PIE_clearInt(obj->pieHandle,PIE_GroupNumber_1);

return;
} // end of HAL_acqTimer0Int() function

```

6.6.6. Initialize Interrupt Vector Table

In hal.h, we have a function that initializes the interrupt vector table with mainISR(), which is our main interrupt that is triggered at the end of conversion of the ADC. For Timer 0, we need to create a function name and assign it to the vector table. We add two things to hal.h, one is an external reference to our new interrupt name, and we assign that interrupt name to the corresponding area in the vector table:

```

extern interrupt void timer0ISR(void);

//! \brief   Initializes the interrupt vector table
//! \details Points the TINT0 to timer0ISR and ADCINT1 to mainISR
//! \param[in] handle The hardware abstraction layer (HAL) handle
static inline void HAL_initIntVectorTable(HAL_Handle handle)
{
    HAL_Obj *obj = (HAL_Obj *)handle;
    PIE_Obj *pie = (PIE_Obj *)obj->pieHandle;

    ENABLE_PROTECTED_REGISTER_WRITE_MODE;

    pie->TINT0 = &timer0ISR;

    pie->ADCINT1 = &mainISR;

    DISABLE_PROTECTED_REGISTER_WRITE_MODE;

    return;
} // end of HAL_initIntVectorTable() function

```

6.6.7. Implement the New Timer 0 Interrupt to Toggle an LED

Finally, we implement the interrupt service routine to toggle an LED. First of all we call the acknowledge function to allow interrupts to happen again, and then we toggle the LED. In proj_lab03a.c we have:

```

interrupt void timer0ISR(void)
{
    // acknowledge the Timer 0 interrupt
    HAL_acqTimer0Int(halHandle);

    // toggle status LED
    HAL_toggleLed(halHandle,HAL_GPIO_LED3);

    return;
} // end of timer0ISR() function

```

Where the following definition is created in hal.h to allow portability to other platforms:

```

//! \brief Defines the GPIO pin number for control card LD3
//!
#define HAL_GPIO_LED3 GPIO_Number_34

```

When running the code, the LED labeled as “LD3” in the control card will toggle every second.


6.7. Adding SCI/UART functionality to a Motorware project

In this example, we will add UART functionality (known as the SCI module in C2000) to an existing Motorware project. We will explore both polling and interrupt based communication with a terminal program through the on-board USB/UART interface. For the below information, a LaunchPadXL-F28069M was used with proj_lab11, but the tutorial should be applicable to any C2000 InstaSPIN-capable device.

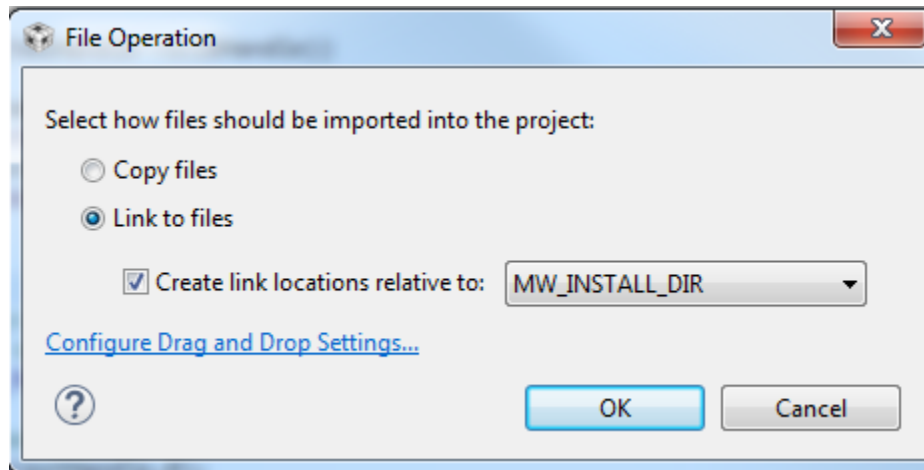
6.7.1. Add SCI source files to project

First, add the SCI/UART source files to the project you are working. Right-click on the project name in the CCS project explorer window, and select “Add Files.” Next, navigate to the following folder and select:

- C:\ti\motorware\motorware_1_01_00_18\sw\drivers\sci\src\32b\f28x\f2806x\sci.c

 sci.c	9/26/2016 2:10 PM	C File	15 KB
 sci.h	9/26/2016 2:10 PM	H File	28 KB

Select “Link to Files” and select the link location relative to “MW_INSTALL_DIR”



6.7.2. Edit the HAL_Obj to add the SCI_Handle

Next, add the SCI handles that will be utilized by the hardware abstraction layer (HAL) object. This is done in “hal_obj.h” header file. Please note, you need to add the “SCI_Handle” lines after the rest of the HAL_Obj definitions, as shown below:

```

178 DRV8301_Handle drv8301Handle;    //!< the drv8301 interface handle
179 DRV8301_Obj    drv8301;          //!< the drv8301 interface object
180
181 #ifdef QEP
182 QEP_Handle     qepHandle[2];      //!< the QEP handles
183 #endif
184
185 SCI_Handle     sciAHandle;         //!< the SCI handle
186 SCI_Handle     sciBHandle;         //!< the SCI handle
187
188 } HAL_Obj;
189

```

6.7.3. Initialize the SCI handles in the HAL initialization function

Now that the SCI_Handle objects have been added to the HAL object, we need to instantiate them in the HAL_init() function call. In the “hal.c” source file, find the HAL_init() function call, and add the following lines of code:

```

// initialize the SCI handles
obj->sciAHandle = SCI_init((void *)SCIA_BASE_ADDR, sizeof(SCI_Obj));
obj->sciBHandle = SCI_init((void *)SCIB_BASE_ADDR, sizeof(SCI_Obj));

```

6.7.4. Prototype the SCI setup functions

Next, it is necessary to prototype the HAL setup SCI function; we will be keeping the same format as the other HAL device driver functions. In the “hal.h” header file, please add the following lines of code:

```

/*! \brief      Sets up the sciA peripheral
/*! \param[in] handle  The hardware abstraction layer (HAL) handle
extern void HAL_setupSciA(HAL_Handle handle);

/*! \brief      Sets up the sciB peripheral
/*! \param[in] handle  The hardware abstraction layer (HAL) handle
extern void HAL_setupSciB(HAL_Handle handle);

```

6.7.5. Define the SCI setup functions

Here we give definition to the function we prototyped in the last step. Shown is the definition for the SCI-B module; that said, it is possible to copy the definition to define the SCI-A module as well. Please add these lines of code to “hal.c” source file:

```

void HAL_setupSciB(HAL_Handle handle)
{
    HAL_Obj *obj = (HAL_Obj *)handle;

    SCI_reset(obj->sciBHandle);
    SCI_enableTx(obj->sciBHandle);
    SCI_enableRx(obj->sciBHandle);

    SCI_disableParity(obj->sciBHandle);
    SCI_setNumStopBits(obj->sciBHandle,SCI_NumStopBits_One);
    SCI_setCharLength(obj->sciBHandle,SCI_CharLength_8_Bits);
    // set baud rate to 115200
    SCI_setBaudRate(obj->sciBHandle,(SCI_BaudRate_e)(0x0061));
    SCI_setPriority(obj->sciBHandle,SCI_Priority_FreeRun);

    SCI_enable(obj->sciBHandle);

    return;
    // end of HAL_setupSciB() function
}

```

Please note, the various SCI module parameters are to be initialized according to the system needs, and the above is just a simple reference. The Baud rate can be calculated as:

$$BRR = \frac{LSPCLK}{f_{UART} * 8} - 1$$

Where BRR is the value stored into the Baud rate register. For a Baud of 115200, BRR is calculated as:

$$BRR = \frac{90MHz}{115200 * 8} - 1 \cong 97_{10} = 61_{16}$$

Please note that LSPCLK = SYSCLKOUT/1 in the above example. For more information, please see the “Serial Communications Interface (SCI)” section for the device’s accompanying Technical Reference Manual (TRM)

6.7.6. Call the SCI setup function for the HAL set parameters function

Next, it is necessary to call the HAL_setupSciB() function in the correct HAL setup function. In the “hal.c” source file, navigate to the HAL_setParams() function and add the following lines:

```
812
813 // setup the spiB
814 HAL_setupSpiB(handle);
815
816
817 // setup the sciA
818 HAL_setupSciA(handle);
819
820
821 // setup the sciB
822 HAL_setupSciB(handle);
823
824
825 // setup the PWM DACs
826 HAL_setupPwmDacs(handle);
827
```

6.7.7. Enable the appropriate SCI peripheral clocks in the HAL setup clocks function

In order to use the SCI modules, the corresponding clocks must be enabled. In the “hal.c” source file, file the HAL_setupPeripheralClks() function, and ensure that these two lines of code are present:

```
1357 CLK_disableClaClock(obj->clkHandle);
1358
1359 CLK_enableSciaClock(obj->clkHandle);
1360 CLK_enableScibClock(obj->clkHandle);
1361
1362 CLK_enableSpiaClock(obj->clkHandle);
1363 CLK_enableSpibClock(obj->clkHandle);
```

The function might have CLK_disableSciaClock() by default; if that is the case, be sure to remove this function call in favor of the one highlighted above.

6.7.8. Enable the appropriate SCI peripheral clocks in the HAL setup clocks function

Depending on the chosen device and the chosen SCI module, the appropriate GPIO settings need to be made. For the LaunchPadXL-F28069M, SCI-B module is pre-set to GPIO15 and GPIO58 for SCIRX and SCITX respectively. In the “hal.c” source file, find the HAL_setupGpios() function and confirm the settings are appropriate for the system needs. See example initialization below, which is the default for the device mentioned above:

```
1134 // UARTB RX
1135 GPIO_setMode(obj->gpioHandle,GPIO_Number_15,GPIO_15_Mode_SCIRXDB);
1294 // UARTB TX
1295 GPIO_setMode(obj->gpioHandle,GPIO_Number_58,GPIO_58_Mode_SCITXDB);
```

Please consult the device TRM and/or EVM/LaunchPad user’s guide for more information on GPIO pinouts and the accompanying mux settings

6.7.9. Add code to background loop to echo a character to a Terminal application

The SCI module should now be ready to receive and transmit. Since interrupts have not been enabled, a simple polling routine is shown below to receive and transmit back to a terminal application:

```
if(SCI_rxDatReady(halHandle->sciBHandle))
{
    while(SCI_rxDatReady(halHandle->sciBHandle) == 0);
    dataRx = SCI_getDataNonBlocking(halHandle->sciBHandle, &success);
    success = SCI_putDataNonBlocking(halHandle->sciBHandle, dataRx);
}
```

Please note: SCI-B has been routed to USB/UART by selecting the appropriate serial connectivity settings for JP6 & JP7 on the LaunchPadXL-F28069M. Please see the accompanying TRM or user's guide for the device being utilized if different.

6.7.10. Adding the SCI interrupt enable code – prototyping the SCI interrupt function

From the previous example of character echo using a polling based system, we will move on to an interrupt driven example. In the "hal.h" header file, please add a function prototype for enabling the SCI interrupts.

```
//! \brief          Enables the SCI interrupts
//! \details Enables the SCI interrupts in the PIE and CPU. Enables the
//!                interrupt to be sent from the SCI peripheral.
//! \param[in]      handle - the hardware abstraction (HAL) handle
extern void HAL_enableSciInts(HAL_Handle handle);
```

6.7.11. Add code to define the SCI interrupt function

Next, in "hal.c," enable the PIE, SCI, and CPU interrupts necessary for an interrupt based receive system for the SCI-B module by adding the following code. The process is similar for initializing a receive interrupt for SCI-A module, as well as initializing SCI transmit interrupts if desired.

```
void HAL_enableSciInts(HAL_Handle handle)
{
    HAL_Obj *obj = (HAL_Obj *)handle;

    // enable the PIE interrupts associated with the SCI interrupts
    // enable SCIB RX interrupt in PIE
    PIE_enableInt(obj->pieHandle,PIE_GroupNumber_9,PIE_InterruptSource_SCIBRX);

    // enable SCI RX interrupts
    // enable SCIB RX interrupt
    SCI_enableRxInt(obj->sciBHandle);

    // enable the cpu interrupt for SCI interrupts
    CPU_enableInt(obj->cpuHandle,CPU_IntNumber_9);
} // end of HAL_enableSciInts() function
```

****PLEASE NOTE:** The CPU interrupt level for the SCI receive and transmit interrupts is level 9. By default, the ADC interrupt (which drives the mainISR function and comprises the backbone of the motor control system) is CPU interrupt level 10. It is recommended to change the ADC interrupt to the highest priority possible. The process is shown below:

In hal.c, change the following function:

```
478 void HAL_enableAdcInts(HAL_Handle handle)
479 {
480     HAL_Obj *obj = (HAL_Obj *)handle;
481
482
483     // enable the PIE interrupts associated with the ADC interrupts
484     PIE_enableAdcInt(obj->pieHandle, ADC_IntNumber_IHP);
485
486
487     // enable the ADC interrupts
488     ADC_enableInt(obj->adcHandle, ADC_IntNumber_1);
489
490
491     // enable the cpu interrupt for ADC interrupts
492     CPU_enableInt(obj->cpuHandle, CPU_IntNumber_1);
493
494     return;
495 } // end of HAL_enableAdcInts() function
```

In hal.h, in the HAL_initIntVectorTable() function, change the following:

```
539     pie->ADCINT1_HP = &mainISR;
```

The previous lines will change the ADC interrupt to the highest priority for the PIE chart, making sure the SCI interrupt will not supersede the necessary motor control function.

6.7.12. Prototype the SCI interrupt service (ISR) routine

Add the following lines of code to “main.h:”

```
///! \brief The SCI-B receive interrupt service (ISR) routine
///!
interrupt void sciBRxISR(void);
```

6.7.13. Add the SCI interrupt service (ISR) routine vector to the PIE table

The Peripheral Interrupt Enable (PIE) table provides the correct address to an interrupt, allowing the correct code to be called when an interrupt is serviced. In the “hal.h” header file, please add the SCI interrupt service routine address we prototyped in the previous step:


```

530 static inline void HAL_initIntVectorTable(HAL_Handle handle)
531 {
532     HAL_Obj *obj = (HAL_Obj *)handle;
533     PIE_Obj *pie = (PIE_Obj *)obj->pieHandle;
534
535     ENABLE_PROTECTED_REGISTER_WRITE_MODE;
536
537     pie->ADCINT1_HP = &mainISR;
538     pie->SCIRXINTB = &sciBRxISR;
539
540     DISABLE_PROTECTED_REGISTER_WRITE_MODE;
541
542     return;
543 } // end of HAL_initIntVectorTable() function

```

In addition, it is necessary to provide a global statement so that the compiler knows that the ISR will be defined in another source file. In the “hal.h” globals section (towards the top of the file), add the following:

```

192 // *****
193 // the globals
194
195 extern interrupt void mainISR(void);
196 extern interrupt void sciBRxISR(void);
197

```

6.7.14. Call the SCI interrupt initialization function

In steps 10 and 11, we prototyped and defined a function to initialize the SCI interrupt. Call the function now in “proj_lab11.c”, or whichever Motorware lab you chose to work with. For the sake of continuity, it is recommended to call the function after the ADC interrupt initialization function has been called, as shown below:

```

425 // initialize the interrupt vector table
426 HAL_initIntVectorTable(halHandle);
427
428 // enable the ADC interrupts
429 HAL_enableAdcInts(halHandle);
430
431 // enable the SCI interrupts
432 HAL_enableSciInts(halHandle);
433
434 // enable global interrupts
435 HAL_enableGlobalInts(halHandle);

```

6.7.15. Define the SCI interrupt routine

Finally, we will define the “sciBRxISR” function that has been prototyped and passed to the PIE vector table. The example code below provides a simple echo back function and clears the interrupt in the PIE, allowing the interrupt to be triggered again.

```
//! \brief the ISR for SCI-B receive interrupt
interrupt void sciBRxISR(void)
{
    HAL_Obj *obj = (HAL_Obj *)halHandle;

    dataRx = SCI_getDataNonBlocking(halHandle->sciBHandle, &success);
    success = SCI_putDataNonBlocking(halHandle->sciBHandle, dataRx);

    // acknowledge interrupt from SCI group so that SCI interrupt
    // is not received twice
    PIE_clearInt(obj->pieHandle, PIE_GroupNumber_9);
} // end of sciBRxISR() function
```

7. References

- MotorWare: www.ti.com/motorware



8. Revision History

- March 2017 release in Motorware 18 package