Hardware Abstraction Layer

1 Hardware Abstraction Layer

1.1 Introduction

Hardware Abstraction Layer (HAL) provides function API-based service to the higher-level layers (ex: Application Framework, customer application, Et cetera) that allows them to perform hardware-oriented operations independent of actual hardware details. This document provides a detailed description of the Hardware Abstraction Layer, its architecture, components, and usage model.

1.1.1 Architectural Principles and Assumptions

1.1.1.1 Motivation for developing the Hardware Abstraction Layer

Legacy motor control application note source code projects, i.e. published prior to 2015, use very little hardware abstraction. They have multiple parts of their code sections accessing hardware/peripherals using inconsistent interfaces. This presented following challenges:

1. Porting code from one hardware to another requires an extensive search-and-replace operation
2. Each combination of PIM and hardware requires a dedicated release of the code project – this is very resource-intensive to maintain
3. Algorithm-related code is mixed with hardware access – it is very hard to make algorithm improvements separate from its hardware dependency.

The development of Hardware Abstraction Layer is intended to solve the above challenges.

1.1.1.2 Design goals

The Hardware Abstraction Layer has been designed with following top-level design targets:

1. Hardware Abstraction Layer should allow customers to generate their board-specific Hardware Abstraction Layer files with minimal effort,
2. Require minimum execution time overhead,
3. Use modular architecture and
4. Utilize MCC for generating the peripheral device drivers (when available).

1.1.1.3 Compatibility with 8-bit and 32-bit device families

The current *implementation* of Hardware Abstraction Layer is designed to work with 16-bit device families, specifically the dsPIC33E family. The architecture of device peripherals such as ADC, DMA, PWM, QEI and System clock tend to vary across different 16-bit device families. Hence, other 16-bit device families are compatible with this implementation of Hardware Abstraction Layer to varying extents depending on features and peripheral architecture of the specific device under question.

This *implementation* Hardware Abstraction Layer was not designed with an intention to work with 8-bit and 32-bit device families. Also, while the static function driver approach used in this implementation of Hardware Abstraction Layer works very well with 8-bit device families, it does make it inefficient with the 32-bit device families where dynamic driver approach would be more appropriate. Nevertheless the modular *architecture* of Hardware Abstraction Layer, in theory, will allow it to replace the current set of peripheral drivers with plibs while working with 32-bit device families.

To summarize, with respect to 8-bit and 32-bit device families:

- *Actual implementation* of Hardware Abstraction Layer – incompatible
- *Interfaces* of Hardware Abstraction Layer – currently incompatible. However, adding a thin 'adapter' layer with wrapper functions can help fix this to a large extent.
- *Architecture* of Hardware Abstraction Layer – no perceivable incompatibilities.

1.1.1.4 Support for motor control algorithms

The current *implementation* of Hardware Abstraction Layer is designed to support Field Oriented Control algorithms in general and sensorless dual-shunt algorithms (ex: AN1078, AN1292, etc.) in particular.

Nevertheless, a preliminary analysis shows that the interfaces included with this implementation of the Hardware Abstraction Layer and its architecture, in general, supports the implementation of following:

1. Single-shunt current reconstruction (AN1299)
2. Sensorless BLDC control with Back-EMF Filtering (six-step commutation; AN1160)
3. Sinusoidal control of PMSM i.e. non-FOC (AN1017)
4. Sensored BLDC control (AN957)

In addition to these, current implementation of Hardware Abstraction Layer also supports any other newer control algorithms (ex: Direct Torque Control, etc.) that use PWM and ADC modules in a similar manner as the above listed algorithms.

1.1.1.5 Support for General Purpose applications

The core element of Hardware Abstraction Layer, its Peripheral Drivers, was developed starting from a general purpose set of peripheral drivers from a different 16-bit device family. Hence, Peripheral Drivers within the Hardware Abstraction Layer are completely compatible with most of the typical General Purpose applications.

Apart from this, the other two components of the Hardware Abstraction Layer, Board Support Package and Hardware Access Functions, are designed specifically to cater to the needs of a typical motor control application. Nevertheless, the basic architecture of these two components does not prevent addition of general purpose interfaces. Depending on the specifics of the application under question, additional interfaces may need to be added to the Hardware Abstraction Layer before it can support General Purpose applications.

1.1.1.6 Optimizations and Assumptions specific to Motor Control usage

The implementation of the Hardware Abstraction Layer uses following features that are specifically designed for Motor Control use cases:

1. In order to reduce instruction cycle overhead (Design goals #2) while maintaining modularity (Design goals #3), Hardware Abstraction Layer extensively uses static inline functions instead of the regular function calls. Regular function calls incur function call overhead and are used only with peripheral

driver initialization functions that are typically not called very often, especially within a timing-sensitive control loop.

2. Instead of using call-back functions from Interrupt Service Routines (ISR), Hardware Abstraction Layer uses preprocessor macro definitions for ISR function headers that can be used in the application. This is, again, targeted to reduce instruction cycle overhead due to function calls.
3. With Design goals #2 in mind, some of the basic Hardware Abstraction Layer operations like SFR bit-set, SFR bit-clear, SFR read and SFR write operations could have been implemented either using static inline functions or using preprocessor macros. At this decision point, an assumption was made to use the static inline functions rather than preprocessor macros since the compiler has more "visibility" into static inline functions compared to a typical preprocessor macro that gets processed before the compiler has a chance to look at it. This choice of using static inline functions enables the compiler to make better optimizations in the application compared to the preprocessor macros.

1.2 Components of Hardware Abstraction Layer

Figure 1 shows a representative block diagram of the Hardware Abstraction Layer in relation to the MC Application Framework and the customer application.

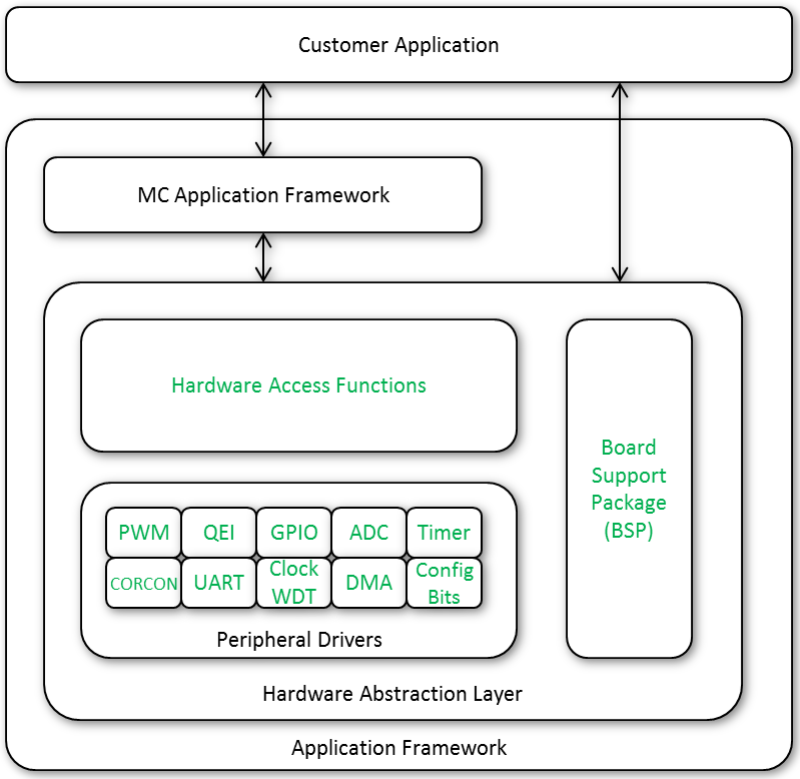

Figure 1: Block diagram of Hardware Abstraction Layer

1.2.1 Peripheral Drivers

This module provides simple functions that access the device peripherals directly. Peripheral driver functions have function names with a prefix correlated to the specific device peripheral that they service. For example, peripheral driver functions for ADC1 module have function names with a prefix of `ADC1_`. A few additional examples are listed below:

```
ADC1_Initialize()
DMA_SoftwareTriggerEnable(DMA_CHANNEL channel)
OSCILLATOR_Initialize(void)
PWM2_DutycycleSet(uint16_t dutyCycle)
QEI1_PositionCount16bitRead(void)
TMR1_Start(void)
```

Currently, following system functions and device peripherals are supported by the Hardware Abstraction Layer :

1. System functions
    a. Oscillator configuration
    b. Device configuration bits
    c. CORCON handling

   d. GPIO configuration with PPS handling
   e. Interrupt management for all HAL-supported peripherals
   f. Watchdog timer
2. ADC1 module
3. PWM module
4. Timer1 module
5. QEI1 module
6. DMA module
7. UART1 module

NOTE:

- Peripheral driver functions are currently hand-written. These hand-written peripheral driver functions maintain a similar look and feel as the MCC-generated functions for devices that are supported by MCC.
- In future, Hardware Abstraction Layer will include support for additional peripherals and also additional APIs for the currently-supported peripherals.
- In future, support for the current set of peripheral modules will be extended to more instantiations of these modules (ex: Timer1 -> Timer2, Timer3, etc.) where it is appropriate.

1.2.2 Hardware Access Functions

This module provides functions to interface low-level peripheral drivers in the Hardware Abstraction Layer with the higher-level application or the Application Framework. This module is a Facade pattern; it hosts simple wrapper functions that translate into peripheral driver operations as well as more complicated functions that utilize peripheral drivers to accomplish device/hardware specific operations. Hardware Access Functions have function names with a prefix `HAL_`. A few examples are listed below:

```
HAL_PwmUpperTransistorsOverrideDisable_Motor1(void)
HAL_PwmUpperTransistorsOverrideLow_Motor1(void)
HAL_PwmSetDutyCyclesIdentical_Motor1(uint16_t dc)
HAL_PwmSetDutyCyclesMotor1(const uint16_t *pdc)
```

This module also provides a set of macro #define names that are intended to provide abstraction from actual device-specific Interrupt Service Routine (ISR) and Trap function names. A few examples are listed below:

```
#define HAL_MATHERROR_TRAP_FUNCTION      _MathError
#define HAL_DMAC_TRAP_FUNCTION           _DMACError
#define HAL_ADC1_ISR                     _AD1Interrupt
#define HAL_DMA0_ISR                     _DMA0Interrupt
```

These ISR and Trap function names are intended to be used in the end-application with appropriate compiler attributes.

1.2.3 Board Support Package

The primary objective of the Board Support Package (BSP) module is to provide one access point for users to modify the hardware mapping details without requiring an extensive search and replace operation. In order to achieve this objective, the BSP is divided into two inter-dependent interfaces:

Application Interface and Peripheral Interface. This type of an approach localizes the *behavioral / functional mapping to the Application Interface* while keeping the *actual hardware mapping within the Peripheral Interface*.

1.2.3.1 Application Interface

The BSP provides a generic Application Interface to allow an abstracted access from the application to specific hardware features without dependency on the actual hardware names or their connections. The *actual names* of generic application macros defined here *should not be changed* and can be used in an application without any dependency on the actual hardware being used. While porting the application to a different hardware, the *mapping definition* can be updated to get the desired hardware behavior while keeping the application code unchanged.

1.2.3.1.1 GPIO Interfaces

General purpose IOs such as LEDs, switches and test points are interfaced in the application using generic macro names defined here. These macro names are defined to map into the appropriate Peripheral Interface macros based on the required application hardware behavior. Following code snippet shows the Application Interface mapping for the dsPICDEM MCLV-2 development board.

```
#define BSP_LED_GP1                          BSP_LATCH_MCLV2_LED_D2
#define BSP_LED_GP2                          BSP_LATCH_MCLV2_LED_D17
#define BSP_TESTPOINT_GP1                    BSP_LATCH_PIM_TESTPOINT_RD8
#define BSP_TESTPOINT_GP2                    BSP_LATCH_MCLV2_TESTPOINT_CANTX
#define BSP_TESTPOINT_GP3                    BSP_LATCH_MCLV2_TESTPOINT_CANRX
#define BSP_TESTPOINT_GP4                    BSP_PORT_PIM_TESTPOINT_HOME
#define BSP_TESTPOINT_GP5                    BSP_LATCH_PIM_TESTPOINT_PGC
#define BSP_TESTPOINT_GP6                    BSP_LATCH_PIM_TESTPOINT_PGD
#define BSP_BUTTON_GP1                       BSP_PORT_MCLV2_BUTTON_S2
#define BSP_BUTTON_GP2                       BSP_PORT_MCLV2_BUTTON_S3

#define BSP_MOTOR1_ADCCHANNEL_POT            ANALOG_CHANNEL_AN13
#define BSP_MOTOR1_ADCCHANNEL_VDC            ANALOG_CHANNEL_AN10
#define BSP_MOTOR1_ADCCHANNEL_IM1            ANALOG_CHANNEL_AN1
#define BSP_MOTOR1_ADCCHANNEL_IM2            ANALOG_CHANNEL_AN0
#define BSP_MOTOR1_ADCCHANNEL_ISUM           ANALOG_CHANNEL_AN2

#define BSP_MOTOR1_PHASEA_CURRENTSENSE()     ADC_SH_CHANNEL2()
#define BSP_MOTOR1_PHASEB_CURRENTSENSE()     ADC_SH_CHANNEL1()
#define BSP_MOTOR1_PHASEC_CURRENTSENSE()     0
#define BSP_MOTOR1_SUMPHASE_CURRENTSENSE()   ADC_SH_CHANNEL3()
#define BSP_MOTOR1_ADC_OUTPUT_POT()          ADC_SH_CHANNEL0()
#define BSP_MOTOR1_ADC_OUTPUT_VBUS()         ADC_SH_CHANNEL0()
```

In the code snippet above, the application interface macro `BSP_LED_GP1` can be used directly in the application while the Peripheral Interface macro `BSP_LATCH_MCLV2_LED_D2` is defined in the Peripheral Interface section of the BSP to provide the required peripheral access method.

In addition to this, BSP also provides static inline functions for implementing simple IO operations (ex: setting a HIGH / LOW state on an LED or a test point, reading the state of a button). A few of these functions are listed in the code snippet below.

```
inline static void BSP_LedGp1Activate() { BSP_LED_GP1 = 1; }
inline static void BSP_LedGp1Deactivate() { BSP_LED_GP1 = 0; }
inline static bool BSP_ButtonIsPressedGp1() { return BSP_BUTTON_GP1; }
```


1.2.3.1.2 Analog Interfaces

The Application Interface supports two different types of Analog Interfaces:

Analog multiplexer interface – this is essentially a mapping from Analog signals on the motor control board to their corresponding Analog channels on the device. This type of interface is useful for devices that have ADC peripherals with analog input multiplexers that allow a single sample-and-hold channel to scan multiple device analog input channels, one at a time. The analog multiplexer interface is intended to be used with an ADC peripheral driver function such as `_ChannelSelectSet()`, in order to select the required channel on the ADC input multiplexer. Next, after the sampling and conversion is complete, a suitable ADC peripheral driver function such as `ADC1_ConversionResultChannel0Get()` can be used to get the analog value for the selected analog signal.

For the example shown in Figure 2, we use this type of analog interface to sequentially sample the Potentiometer (V_{POT}) and DC bus voltage sense (V_{DC}) inputs as depicted in Figure 2.

```
#define BSP_MOTOR1_ADCCHANNEL_POT           ANALOG_CHANNEL_AN13
#define BSP_MOTOR1_ADCCHANNEL_VDC           ANALOG_CHANNEL_AN10
```

Here, macros `BSP_MOTOR1_ADCCHANNEL_POT` and `BSP_MOTOR1_ADCCHANNEL_VDC` can be directly used in the application framework while `ANALOG_CHANNEL_AN13` and `ANALOG_CHANNEL_AN10` are aliases defined by the BSP helper as numbers `10` and `13` respectively.

See 1.3.3.3.3 ADC channel switching for more details on this use case.

Static analog mapping interface – this interface provides direct mapping from Analog signals on the motor control board to their corresponding ADC channel buffers. This type of analog interface is useful in cases where an analog signal on the motor control board is connected to a device analog pin that is always sampled and converted by a dedicated sample-and-hold channel into one of the ADC buffers.

Figure 2 shows a typical case where this type of analog interface is useful. In this case, the motor phase current sense inputs are simultaneously sampled using sample-and-hold channels CH1, CH2 and CH3. The Application Interface defines macros to provide a direct mapping between board signal names and the appropriate peripheral access methods as shown in the snippet below.

```
#define BSP_MOTOR1_PHASEA_CURRENTSENSE()    ADC_SH_CHANNEL2()
#define BSP_MOTOR1_PHASEB_CURRENTSENSE()    ADC_SH_CHANNEL1()
#define BSP_MOTOR1_PHASEC_CURRENTSENSE()    0
```

Here, the peripheral access methods `ADC_SH_CHANNEL2()` and `ADC_SH_CHANNEL1()` are defined by the BSP as aliases of peripheral driver functions `ADC1_ConversionResultChannel2Get()` and `ADC1_ConversionResultChannel1Get()` respectively.

See 1.3.3.3.5 Phase currents for more details on this use case.

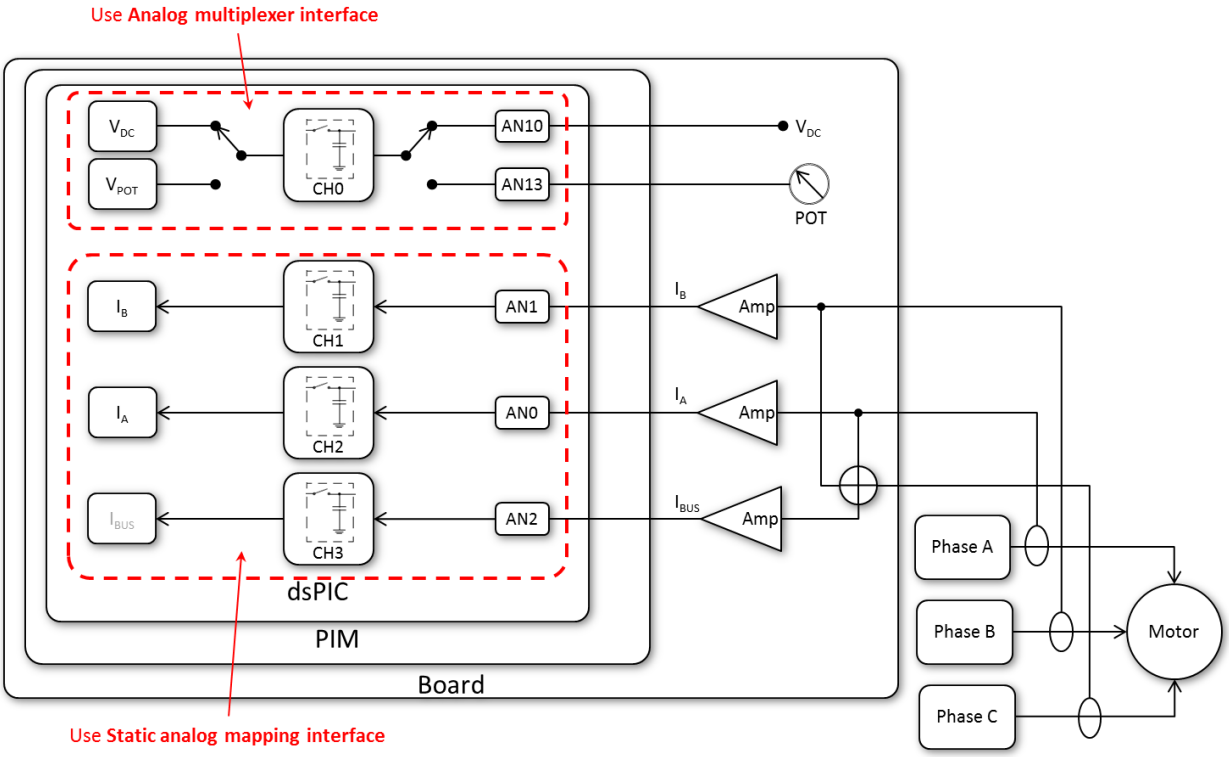

Figure 2: Analog channel interfaces in a typical application

1.2.3.2 Peripheral Interface

BSP uses the Peripheral Interface to map hardware specific macro names to their corresponding peripheral driver/access methods. The macros defined in this interface serve only as a means to define the mapping and should not be directly used in the application/framework. While porting an application to a different hardware, the macro names defined in this interface should be updated to match the signal names on the new hardware.

1.2.3.2.1 GPIO Interfaces

Peripheral Interface provides a mapping between hardware specific GPIO macro names and their peripheral access methods, which in this case are specific bits of the PORT and LATCH registers. Following code snippet shows the Peripheral Interface mapping for the dsPICDEM MCLV-2 development board.

```
#define BSP_PORT_MCLV2_BUTTON_S2            !PORTGbits.RG7
#define BSP_PORT_MCLV2_BUTTON_S3            !PORTGbits.RG6
#define BSP_LATCH_MCLV2_LED_D2              LATDbits.LATD6
#define BSP_LATCH_MCLV2_LED_D17             LATDbits.LATD5
#define BSP_LATCH_PIM_TESTPOINT_RD8         LATDbits.LATD8
#define BSP_PORT_PIM_TESTPOINT_RD8          PORTDbits.RD8
#define BSP_LATCH_MCLV2_TESTPOINT_CANTX     LATCbits.LATC8
#define BSP_LATCH_MCLV2_TESTPOINT_CANRX     LATCbits.LATC9
#define BSP_LATCH_PIM_TESTPOINT_HOME        LATCbits.LATC10
#define BSP_PORT_PIM_TESTPOINT_HOME         PORTCbits.RC10
#define BSP_PORT_PIM_TESTPOINT_RD8          PORTDbits.RD8
#define BSP_LATCH_PIM_TESTPOINT_PGC         LATBbits.LATB6
#define BSP_LATCH_PIM_TESTPOINT_PGD         LATBbits.LATB5
```


These Peripheral Interface macro names (ex: `BSP_LATCH_MCLV2_LED_D2`) serve only as a means to define the mapping and should not be directly used in the application/framework.

1.2.3.2.2 PWM channel to Motor phase mapping

The Peripheral Interface also provides the means to define the mapping between PWM channels and the motor phases. This mapping is not used by the Application Interface; however, it is used in some of the Hardware Access Functions to automatically adjust for cross-connected phases while using these macros as index for accessing a 1x3 array of duty cycle values.

Microchip development boards typically have PWM channels mapped linearly to their corresponding motor phases i.e. PWM1 -> Phase A, PWM2 -> Phase B and PWM3 -> Phase C. The following code snippet shows the linear mapping for a MCLV-2 development board.

```
#define BSP_MOTOR1_PHASEA_PWMCHANNEL     PWM_CHANNEL1
#define BSP_MOTOR1_PHASEB_PWMCHANNEL     PWM_CHANNEL2
#define BSP_MOTOR1_PHASEC_PWMCHANNEL     PWM_CHANNEL3
```

For a custom board with a PWM channel to motor phase mapping as shown in Figure 3, the application interface mapping will be as shown below.

```
#define BSP_MOTOR1_PHASEA_PWMCHANNEL     PWM_CHANNEL2
#define BSP_MOTOR1_PHASEB_PWMCHANNEL     PWM_CHANNEL1
#define BSP_MOTOR1_PHASEC_PWMCHANNEL     PWM_CHANNEL3
```

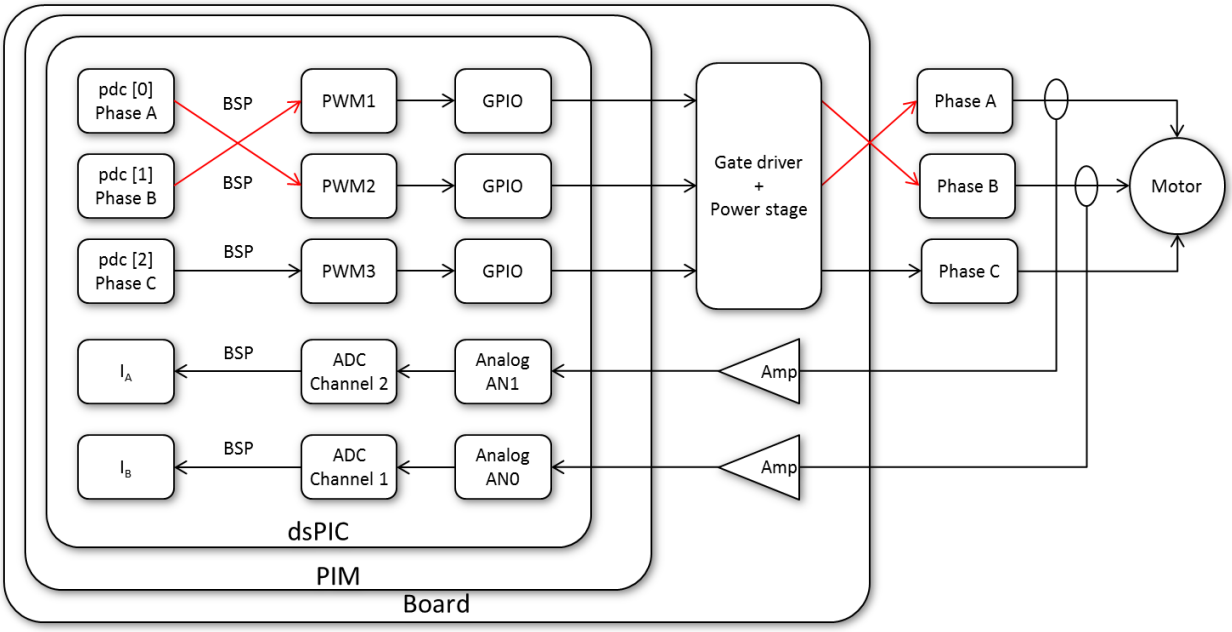

Figure 3: Custom board with crisscrossed PWM channel to motor phase mapping

1.2.3.3 Summary of BSP interfaces

Figure 4 shows a top-level interface diagram of the BSP.

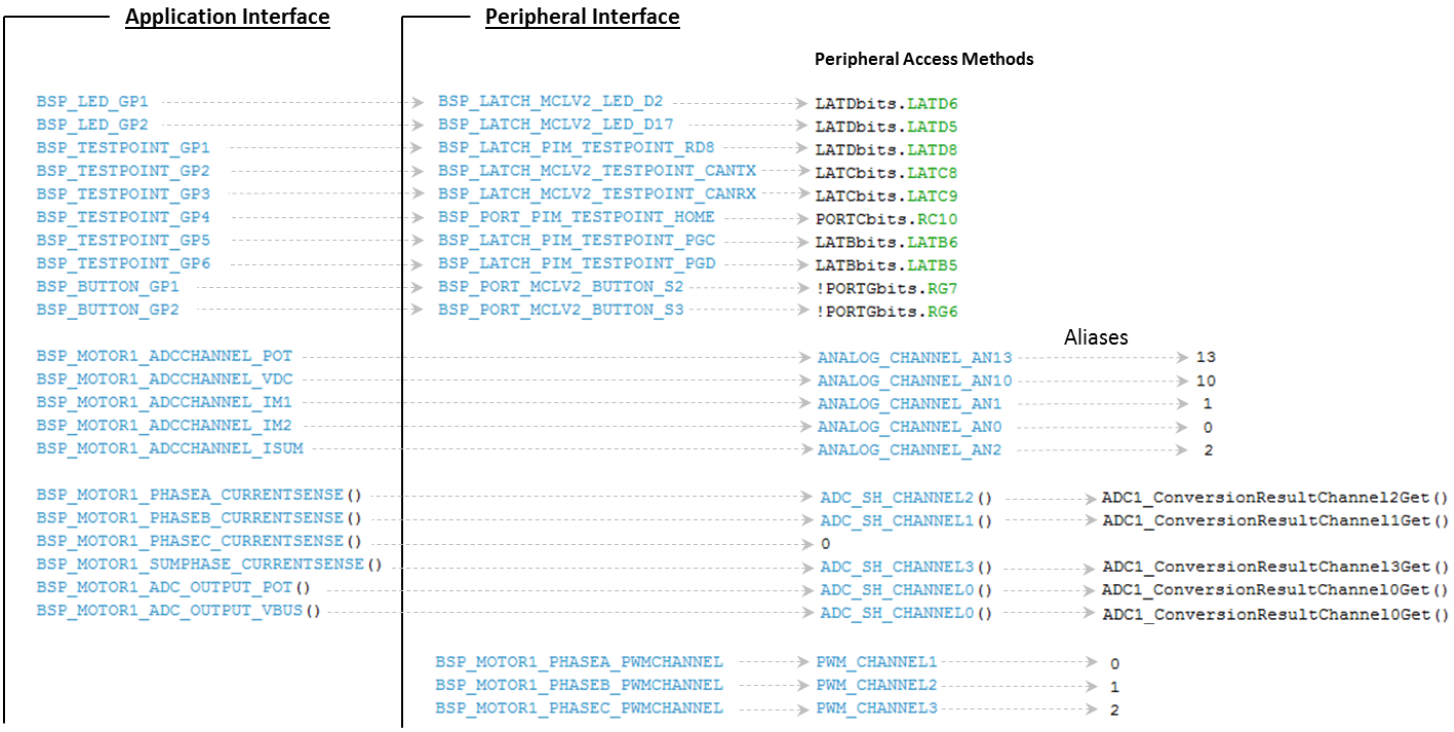

Figure 4: Summary of BSP interfaces

1.3 Usage Model

This section describes the usage model for the Hardware Abstraction Layer.

- Microchip motor control development boards and their compatible PIMs will be supported out-of-the-box with pre-defined Hardware Abstraction Layer files
- Each Microchip board-and-PIM combination will have *one set* of Hardware Abstraction Layer files
- End-users will have to create Hardware Abstraction Layer files for their end-application/development boards. The following section describes steps required to create Hardware Abstraction Layer files for any given non-Microchip board.

1.3.1 Creating Hardware Abstraction Layer files for a new board

Following are the recommended steps to create Hardware Abstraction Layer files for a new board:

1. Board Support Package
    a. Start with a Microchip-developed BSP file ***bsp.h***
    b. Update the Peripheral Interface section of the BSP to re-define the hardware mapping. Rename the existing macros to match the signal label on hardware. If required, define new macros to provide access to hardware IOs that are not already covered
    c. Update the Application Interface section of the BSP to re-define the behavioral mapping of the hardware. If required, define new macros to provide access to hardware IOs that are not already covered; however, *do not rename the macros that are already defined*.
2. Pin manager
    a. Start with a Microchip-developed pin manager file ***pin_manager.c***

   b. Update the `PIN_MANAGER_Initialize()` to ensure that all GPIOs required by the application/framework are appropriately setup and their remap configuration correctly assigned
3. Peripheral drivers
   a. Start with a set of Microchip-developed peripheral driver files
   b. Update the driver initialization functions `XYZ_Initialize()` to initialize the peripherals as required by the application/framework
4. System driver
   a. Start with the Microchip-developed system driver files ***mcc.c*** and ***mcc.h***
   b. Update the device configuration bits as needed by the application/framework
   c. Update the `OSCILLATOR_Initialize()` as required

1.3.2 Using Hardware Abstraction Layer in an application

In order to use Hardware Abstraction Layer functions within an application, follow these steps:

1. Add the Hardware Abstraction Layer files into the MPLAB X project
2. Include ***mcc.h*** file in all application source/header files that reference Hardware Abstraction Layer functions
3. At the device initialization phase in main(), call `SYSTEM_Initialize()` function to initialize device peripherals like oscillator, ADC, PWM, etc. In addition to this, individual peripheral driver initialization functions can be called from the application at a later point of time.
4. When application needs to access a GPIO, use the macro names defined by the BSP Application Interface in ***bsp.h*** file
5. When the application needs to access device peripheral features, use the Hardware Access Functions defined in ***hardware_access_functions.h*** file and/or peripheral driver functions defined in the corresponding device driver files along with the BSP defined in ***bsp.h*** file.
6. Define ISR and Trap functions using the ISR and Trap helper macros included in the ***hardware_access_functions.h*** file. Use `void __attribute__((interrupt))` for this purpose and include either `auto_psv` or `no_auto_psv` attribute in the function definition as required by the application.

Additional details regarding the peripheral driver functions and hardware access functions are documented in the API documentation (separate document).

1.3.3 Hardware Abstraction Layer usage excerpts from MC Application Framework

As example, this section provides snapshots of Hardware Abstraction Layer usage within the MC Application Framework.

NOTE: MC Application Framework source code that is quoted in this section may be out of date. Please use motorBench™ Development Tool to obtain the latest version of MC Application Framework.

1.3.3.1 System functions

In the application code snippet shown below, the `MCAPP_SystemInit()` is calling into the Hardware Abstraction Layer function `SYSTEM_Initialize()` in order to initialize the device peripherals.

```
void MCAPP_SystemInit(MCAPP_SYSTEM_DATA *psys)
{
    psys->debugCounters.reset = ++MCAPP_resetCounter;

    SYSTEM_Initialize();
    MCAPP_ConfigurationPwmUpdate();
    MCAPP_InterruptPriorityConfigure();
    ADC1_ModuleEnable();

    MCAPP_DiagnosticsInit();
}
```

1.3.3.2 Pin manager

The pin manager function of Hardware Abstraction Layer, `PIN_MANAGER_Initialize()`, is indirectly accessed by the application framework using the `SYSTEM_Initialize()` function. The Hardware Abstraction Layer pin manager performs the following tasks:

1. Initialize LATx and TRISx registers as needed by the application
2. Configure the PPS settings to assign remap IOs to the appropriate device peripherals

1.3.3.3 ADC module

1.3.3.3.1 Potentiometer

The following application code snippet reads the potentiometer input from ADC buffer0 and calculates the velocity command based on a scaling factor.

```
uint16_t unipolarADCResult = ADC1_ConversionResultChannel0Get() + 0x8000;
pmotor->velocityControl.velocityCmd = MCAPP_DetermineVelocityCommand(pmotor,
unipolarADCResult);
```

1.3.3.3.2 V_{BUS} sense

The following application code snippet reads the V_{BUS} sense input from ADC buffer0 and calculates scaled value of V_{BUS}.

```
uint16_t unipolarADCResult = ADC1_ConversionResultChannel0Get() + 0x8000;
pmotor->psys->vDC = unipolarADCResult >> 1;
```

1.3.3.3.3 ADC channel switching

The following application code snippet switches ADC sample-hold channel #0 to read the analog input connected to the potentiometer (as defined in the BSP).

```
ADC1_ChannelSelect(BSP_MOTOR1_ADCCHANNEL_POT);
```

The above code will set the ADC1 input channel #0 select register (AD1CHS0) to a number defined by the macro `BSP_MOTOR1_ADCCHANNEL_POT`.

1.3.3.3.4 ADC interrupt service routine

The following application code snippet implements the ADC interrupt service routine using the ADC peripheral driver.

```
ADC1_ISR_FUNCTION_HEADER(void)
```



```
{
    BSP_TestpointGp1Activate();
#ifdef MCAPP_TEST_PROFILING
    motor.testing.timestampReference = TMR1_Counter16BitGet();
#endif
    ADC1_FlagInterruptClear();
    MCAPP_SystemStateMachine_StepIsr(&motor);
    MCAPP_UiStepIsr(&motor.ui);
    MCAPP_MonitorStepIsr(&motor);
    MCAPP_WatchdogManageIsr(&watchdog);

    /* Test and diagnostics code are always the lowest-priority routine
within
     * this ISR; diagnostics code should always be last.
     */
    MCAPP_TestHarnessStepIsr(&systemData.testing);
    capture_timestamp(&motor.testing, 6);
    MCAPP_DiagnosticsStepIsr();
    capture_timestamp(&motor.testing, 7);
    BSP_TestpointGp1Deactivate();
}
```

Here, function header for the ADC1 interrupt service routine and the ADC1 interrupt flag access macro are defined by Hardware Abstraction Layer within the ***hardware_access_functions.h*** file.

1.3.3.3.5 Phase currents

The following application code snippet reads motor phase currents using the Analog Interfaces.

```
void MCAPP_FocReadADC(MCAPP_MOTOR_DATA *pmotor)
{
    pmotor->iabc.a = BSP_MOTOR1_PHASEA_CURRENTSENSE();
    pmotor->iabc.b = BSP_MOTOR1_PHASEB_CURRENTSENSE();
```

1.3.3.4 PWM module

1.3.3.4.1 PWM fault handling

The following application code snippet implements the PWM fault handling routine using the PWM peripheral driver function.

```
inline static bool MCAPP_OvercurrentHWDetect(void)
{
    return PWM1_FaultStatusGet();
}
```

1.3.3.5 QEI module

1.3.3.5.1 Mode initialization

The following application code snippet initializes QEI1 to operate in modulo-count mode.

```
void MCAPP_InitQEI(MCAPP_QEI_ESTIMATOR_T *pqei)
{
    QEI1_Initialize();
    QEI1_ModuloMode16bitSet(ENCODER_COUNTS_PER_REV);
    QEI1_ModuleEnable();
```



```
MCAPP_TrackingLoopInit(&pqei->trackingLoop);
```

1.3.3.5.2 Position write

The following application code snippet presets the QEI position value in order to correct the offset in QEI angle.

```
void MCAPP_AlignQEI(MCAPP_QEI_ESTIMATOR_T *pqei)
{
    /* Reset QEI position count to the motor zero angle */
    QEI1_PositionCountWrite(MCAPP_QEI_ALIGN_COUNT);
}
```

1.3.3.6 Hardware Access Functions

The following application code snippet limits the three-phase duty cycle values to `MIN_DUTY` and writes them to the appropriate PWM channels of motor #1 using Hardware Abstraction Layer.

```
inline void MCAPP_MotorControllerOnActiveStates(MCAPP_MOTOR_DATA *pmotor)
{
    MCAPP_FocStepIsrForwardPath(pmotor);

    {
        uint16_t pwmDutyCycle[3];

        pwmDutyCycle[0] = UTIL_LimitMinimumU16(pmotor->pwmDutycycle.dutycycle1,
MIN_DUTY);
        pwmDutyCycle[1] = UTIL_LimitMinimumU16(pmotor->pwmDutycycle.dutycycle2,
MIN_DUTY);
        pwmDutyCycle[2] = UTIL_LimitMinimumU16(pmotor->pwmDutycycle.dutycycle3,
MIN_DUTY);
        HAL_PwmSetDutyCycles_Motor1(pwmDutyCycle);
    }
}
```

The hardware access function `HAL_PwmSetDutyCyclesMotor1()` takes the three phase duty cycle values in `pwm_dutycycle` and writes them to the PWM channels as defined in the BSP.

1.4 Revision History

Version	Date	Author	Description
1	24/Feb/2017	Srikar Deshmukh - C14317	First revision of the document