MotorWare Software Architecture

February 2013

Contents

1 Acronyms

API – Application Programming Interface

ACIM – Alternating Current Induction Motor

ADC – Analog-to-Digital Converter

ePWM – Enhanced Pulse Width Modular

FIFO – First In, First Out

GUI – Graphical User Interface

HAL – Hardware Abstraction Layer

IDE – Integrated Development Environment

ISR – Interrupt Service Routine

MCU – Micro-Controller Unit

PIE – Peripheral Interrupt Expansion

PWM – Pulse Width Modulation

PMSM – Permanent Magnet Synchronous Motor

RAM – Random Access Memory

ROM – Read Only Memory

RTOS – Real Time Operating System

SOC – Start Of Conversion

1 Introduction

The purpose of a software architecture is to provide a consistent development, testing, and maintenance platform for a development team working on a common collection of software. By having a consistent architecture, it becomes easier to use and maintain the software as the number of files in the collection grows and as software developers come and go from the team. A well-defined software architecture should port across processor architectures easily and should be understood by software developers with different levels of programming skill. A software architecture can have the best features in the world when it comes to programming concepts (e.g. inheritance, abstraction, etc.), but if it is difficult to use by most of the team, what good is it really. The architecture becomes more of efficiency inhibitor rather than an efficiency enabler for the overall software development and maintenance process.

Today's software development teams have to adapt quickly to new processors and new applications. Software re-use is a must in today's world of software development. As a result, software that is designed for a particular processor architecture should be avoided in most cases. It is expensive - both from a development and a maintenance perspective. As a result, the software architecture must support and promote re-use. And once again, if a software architecture is difficult to use, it certainty won't be re–used.

This document describes the software architecture for the MotorWare package. The software found in this package contains the following features

- Well-defined Software Stack – driver, controller, estimator
- Modular software methodology – object oriented programming model
- Improved code execution - Queues for priority based code execution, inlined functions
- Software portability across processors
- Software re-use
- Embedded software documentation - Doxygen
- Coding standard
- Single source files

2 Software Architecture

This section describes the software architecture for the MotorWare package in terms of its software stack, its modular design (e.g. objects), the use of software handles, API definitions, software-reuse and the MotorWare package directory structure.

2.1 Software Stack

The MotorWare software stack is shown pictorially in Figure 1, which contains three main software components – the user interface (i.e. IFACE), the controller (i.e. CTRL) and the driver (i.e. DRV). Each of these components can be thought of as having a specific and unique set of tasks to do. Each of these components contains the various modules needed by the component to complete its set of tasks. This modular structure forms the basis for object oriented programming, which will discussed later in this document.

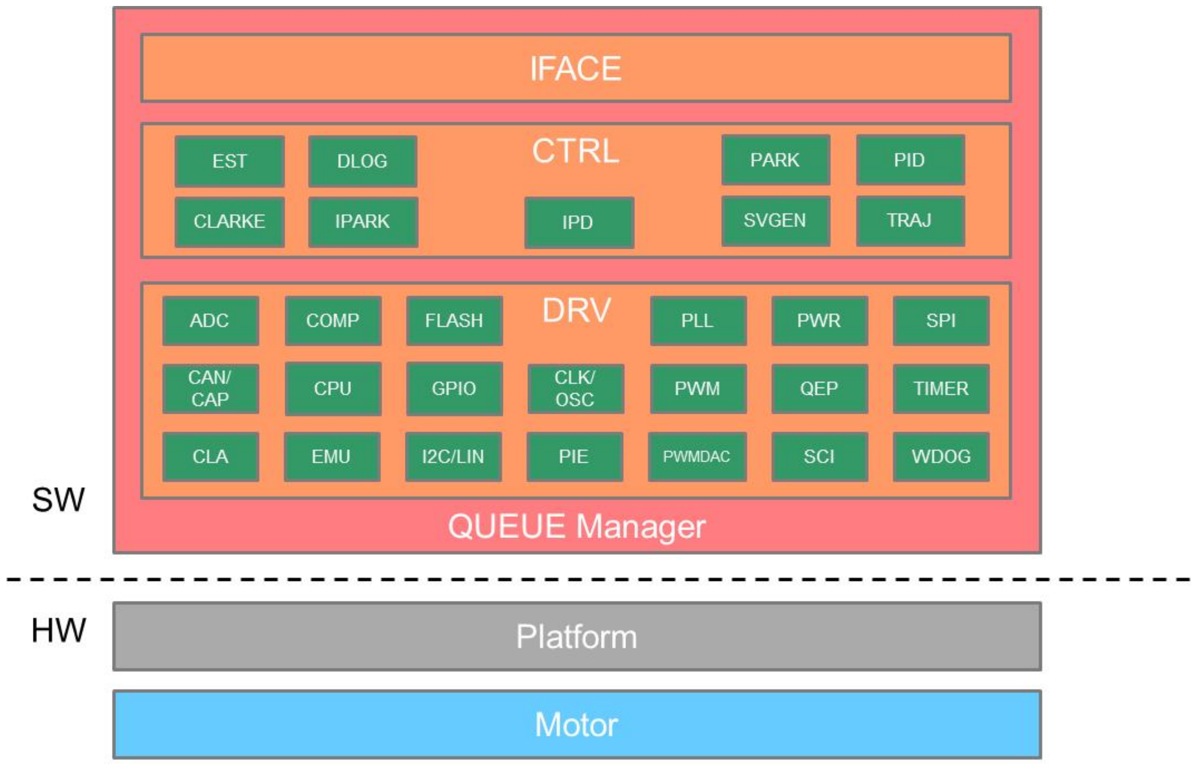


Figure 1 MotorWare Software Stack

The IFACE component is a mechanism for users to input parameters into the control system. A GUI running on a host machine that communicates via a serial port to input desired set points would be one example. The CTRL component contains all of the components of a conventional digital control system. The DRV component contains all of the software needed to configure and control the hardware components in the system. Some software architectures refer to this layer as the HAL. In the most general implementation, the software runs from within a queue manager that executes the control system in the background and gives priority to ISRs that give commands to

the ePWM modules. To determine these commands, data must be converted and the control system must be run at a frequency less than or equal to the ISR rate.
The hardware represented in the figure refers to the particular motor control kit and corresponding target processor that the software is running on as well as the motor that the software has been properly configured to control.

2.2 Objects

The fundamental building block in the software architecture is an object. An object is nothing more than a collection of parameters and associated methods (i.e. functions) needed to perform the tasks which define the object. For example, a filter object would contain the filter coefficients needed to implement the filtering operation as well as any intermediate data values needed by the filter to compute the desired output value. It would not, however, contain parameters needed to perform a second operation such as a Park transform, since this operation is independent of the filtering operation.

The filter object would also contain methods used to perform the filtering operation and methods used to access parameters in the object. These methods are commonly called accessor methods (i.e. get(), set(), etc.) because they access parameters in the object. A good object oriented programming practice is to only access an object's parameters thru accessor methods and not by directly manipulating the object. While on the surface, this may appear to be programming overhead, most compilers today abstract this layer away. However, it provides some important options to the object developer.

For example, consider that you are using a speed estimation object and need to set the speed to a desired value. If you are unfamiliar with the algorithms used in the speed estimation object, you may naively manipulate the object directly and just set the speed to the desired value. However, the object might contain a low pass filter as part of the algorithm used to compute the speed estimate. Without setting the values in the filter, the speed estimator may produce some initial speed estimates that are undesirable or even inaccurate. This issue is avoided by having an accessor function (e.g. SPEED_setSpeed()) which takes care of all the necessary steps needed to set the speed to the desired value and to ensure that the object operated correctly after a new speed value has been set. Since the object developer knows the module best, they should provide accessor methods to ensure the proper functionality of the object.

Writing software in an object oriented manner also produces more efficient, re-useable and maintainable code. A software developer can create an object without having to know the entire software application, objects can be re-used from application to application and their maintenance is easier because they are smaller, independent components that are usually well defined (which translate into well-defined test strategies).

2.3 Handles

In this software architecture, handles are used to reference objects. Functions should pass handles to objects and not the objects themselves (or the pointers to objects). Handles are nothing more than abstracted pointers to objects, without the extra typing and interpretation overhead of the pointer syntax. However, the use of handles makes the code much more readable and understandable by most programmers.
All handles are initialized by calling an object's init() method. By design, every object must have an init() method.

2.4 APIs

APIs are the interface for every object. They provide a mechanism to initialize the object, to quarry the object for parameter values and to execute the object. The following naming convention is used for all APIs related to an object

void OBJECT_foo(OBJECT_Handle handle,const void *pInputData_1, …,void *pOutputData_1, …)

where

OBJECT – the name of the object in capital letters
Benefits: Software developers can easily determine the owner object for methods
Requirements: must be the first text in the method name

foo – an arbitrary method name
Requirements: Every object must have a few common methods. Additional method names are at the discretion of the software developer

handle – the handle to the object
Requirements: the first input argument to every method

inputData – the input data to the method. This data is used by the object to produce output data
Requirements: data types include const, single data values are passed by value, arrays of data are passed by reference

outputData – the output data from the method. This data is generated by the object
Requirements: single data values and arrays of data are passed by reference

Note that the input and output data references are not part of the object itself but rather passed to the object's methods as arguments. The object operates on the input data using the object's parameters to produce the needed output data.

Each object, which is located in the modules directory of the MotorWare package, contains the following methods

1 OBJECT_getValue() – used to get a value from an object

2 OBJECT_init() – used to initialize an object

3 OBJECT_run() – used to run an object

4 OBJECT_setValue() – used to set a value in an object

As a result, every object has a built-in initialization method, accessor methods and a run method.

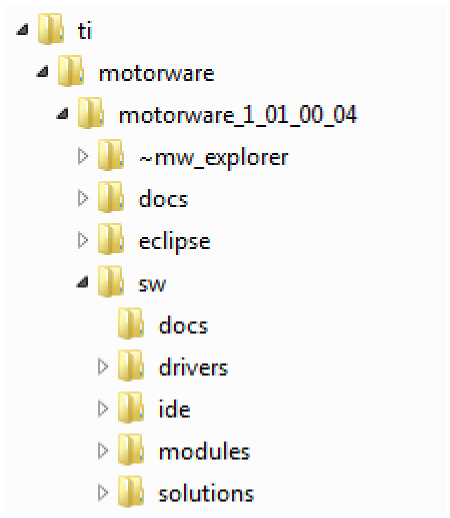
2.5 Software Re-Use

The concept of software re-use is demonstrated throughout the software architecture for the MotorWare package. For example, at the module level, a single PID module is used to implement the speed controller, the Id current controller and the Iq current controller. By using a single module for three different controllers, the code size in terms of the memory footprint can be smaller when compared to other motor control software architectures. At the project level, a single driver can be used for multiple projects for a given processor target/board combination. This software re-use principle translates to single source files, which minimizes the number of files in the software package and reduces the amount of software testing and maintenance.

To enable software re-use at the module level, they have to be re-entrant, which means that a module should properly manage data in its object definition and its functions. Each time a module is entered, it should ensure the integrity of the data values that are needed to execute the function to completion. For example, a module can be interrupted during execution by a higher priority task which plans to execute the function. The module should have separate memory regions for each instance so that when the first task is interrupted, the second task can execute to completion and then continue with the first task until it completes. The compiler and operating system will handle the data movement properly if the scope all of the data values are temporary with respect to the function. As a result, there are no static variables or global variables used in a function.

2.6 Directory Structure

The directory structure used for the MotorWare package can be represented pictorially as shown in Figure 2. The top level directory is called motorware. The main components within a given version of motorware are ~mw_explorer, docs, eclipse and sw.



```
ti
  motorware
    motorware_1_01_00_04
      ~mw_explorer
      docs
      eclipse
      sw
        docs
        drivers
        ide
        modules
        solutions
```

Figure 2 MotorWare Top Level Directory Structure

The ~mw_explorer and eclipse directories contains files for the MotorWare.exe resource explorer.

The docs directory contains top level MotorWare, hardware boards and GUI documentation for the software.

The sw directory contains the MotorWare software. The main components are docs, drivers, ide, modules and solutions.

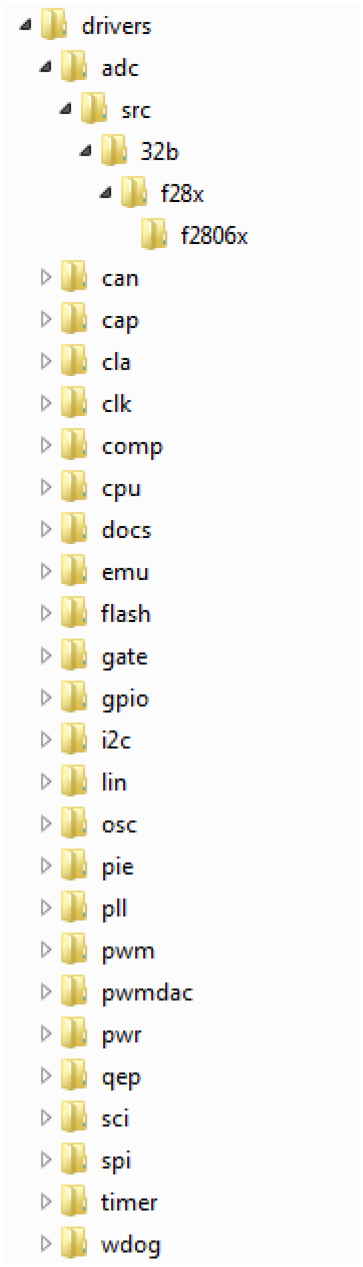
The sw/docs directory contains any top level documentation for the the software. The Doxygen documentation generated from the sowftware is located here.

The sw/drivers directory contains the driver software for the hardware components in a particular processor chip. This structure of this directory will be discussed in Section 1.1.1.

The sw/ide directory contains any IDE related files that are shared across projects within the MotorWare package.   For example, target configuration files are kept here. The modules directory contains the various algorithmic building blocks used to implement a motor control solution. The structure of this directory will be discussed in Section 2.1.1.

The sw/modules directory contains all of the modules for MotorWare, such as the FAST^{TM} estimator.   The structure of the directory will be discussed in Section 2.1.1.

The sw/solutions directory contains example projects that combine the drivers and modules to create a motor control implementation, such as InstaSPIN-FOC^{TM}. The structure of this directory will be discussed in Section 3.1.1.



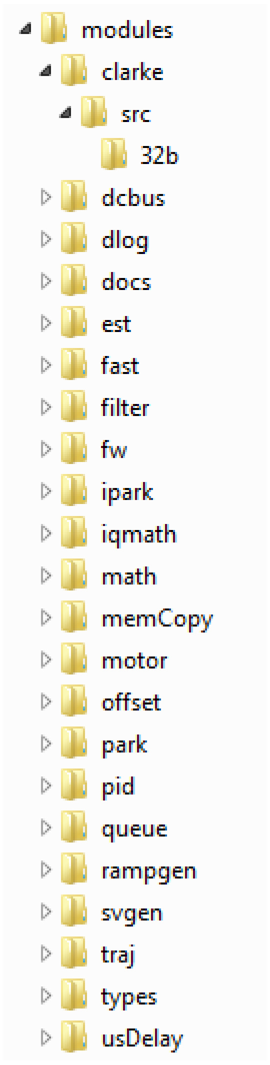
```
drivers
    adc
        src
            32b
                f28x
                    f2806x
    can
    cap
    cla
    clk
    comp
    cpu
    docs
    emu
    flash
    gate
    gpio
    i2c
    lin
    osc
    pie
    pll
    pwm
    pwmdac
    pwr
    qep
    sci
    spi
    timer
    wdog
```

Figure 3 MotorWare Drivers Directory Structure

2.6.1 Drivers

The sw/drivers directory structure is shown in Figure 3. The sw/drivers directory contains over 35 different drivers, which will not be listed here. See the MotorWare package for a complete list. For each driver, the main component is a src directory.

The src directory contains driver code for supported processors. The types of processors are sub-divided into 16 bit integer (i.e. 16b), 32 bit integer (i.e. 32b), 16 bit float (i.e. float16) and/or 32 bit float (i.e. float32). Within these four categories, the processors are further divided into families (i.e. f2806x, etc).



```
modules
  clarke
    src
      32b
  dcbus
  dlog
  docs
  est
  fast
  filter
  fw
  ipark
  iqmath
  math
  memCopy
  motor
  offset
  park
  pid
  queue
  rampgen
  svgen
  traj
  types
  usDelay
```

Figure 4 MotorWare Modules Directory Structure

2.6.2 Modules

The sw/modules directory structure is shown in Figure 4. The sw/modules directory contains over 20 different modules, which will not be listed here. See the MotorWare package for a complete list. For each module, the main components are a docs, lib and src directories.

The docs directory contains any documentation related to a particular driver.

The lib directory contains any library files that have been created using the module.

The src directory contains module code for supported processors. The types of processors are sub-divided down into 16 bit integer (i.e. 16b), 32 bit integer (i.e. 32b), 16 bit float (i.e. float16) and 32 bit float (i.e. float32) if needed. The portable versions for a given module reside here. Within these four categories, the processors are further divided into families (i.e. f2806x, etc) if needed. The architecture specific versions of a given module reside here.

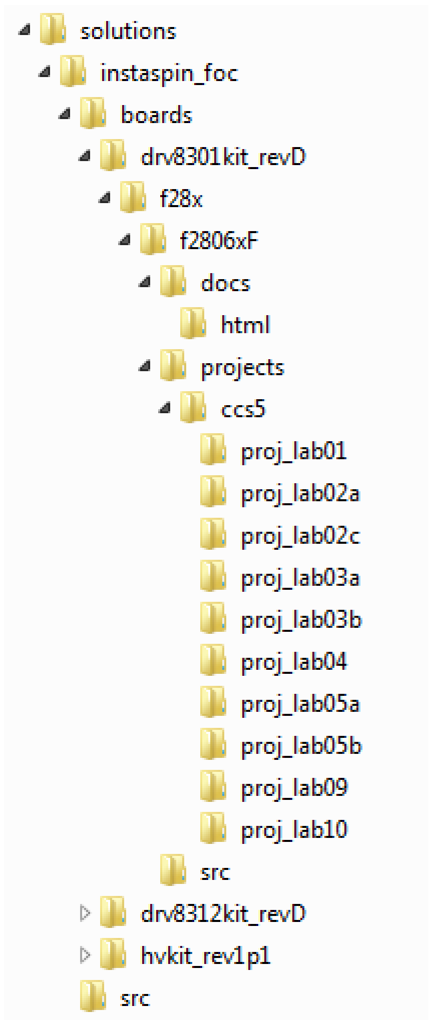
2.6.3 Solutions

The sw/solutions directory structure is shown in Figure 5. The sw/solutions directory currently only contains the InstaSPIN-FOC solution. For each solution, the main components are boards, docs and src directories.

The boards subdirectory contains the solution code for a particular board/target combination. For a given board, the targets directory (e.g. f28x) contains a docs subdirectory contains any documentation related to the tests, a project subdirectory for related project files and a src subdirectory contains the main test source files.

The docs directory contains any documentation related to a particular solution.

The src directory contains common code to test the solution.



```
solutions
  instaspin_foc
    boards
      drv8301kit_revD
        f28x
          f2806xF
            docs
              html
            projects
              ccs5
                proj_lab01
                proj_lab02a
                proj_lab02c
                proj_lab03a
                proj_lab03b
                proj_lab04
                proj_lab05a
                proj_lab05b
                proj_lab09
                proj_lab10
            src
      drv8312kit_revD
      hvkit_rev1p1
    src
```

Figure 5 MotorWare Solutions Directory Structure

3 Documentation

The software itself contains Doxygen comments located in the header file for each driver, module and solution.

3.1 Coding Standard

The coding standard used for MotorWare is defined in motorware_coding_standards.pdf [1].

4 Control Path Architecture

This section describes the control path architecture for the digital control system used in the MotorWare package. Different aspects of the design such as system timing, multi-rate hardware and software execution and threads are discussed. Several examples are provided along the way to help explain the issues and some solutions that are provided in the MotorWare package.

4.1 System Timing Configuration

Digital control is the design of control systems in a domain where signals have been uniformly sampled in the continuous time domain and have quantized amplitude values. The sampling process can be represented mathematically as

$$x[n] = x(n \cdot T)$$

where $x[n]$ is the discrete time signal representation, $x(t)$ is the continuous time signal representation, *T* is the periodic sampling rate of the control system and *n* is the sample number. A/D converters typically perform the sampling process, which is the case for the control systems implemented using TI processors and micro-controllers. This section will discuss how to configure periodic sampling in TI's F28x class of MCUs and how the software architecture is designed to give the most flexibility for user algorithms.

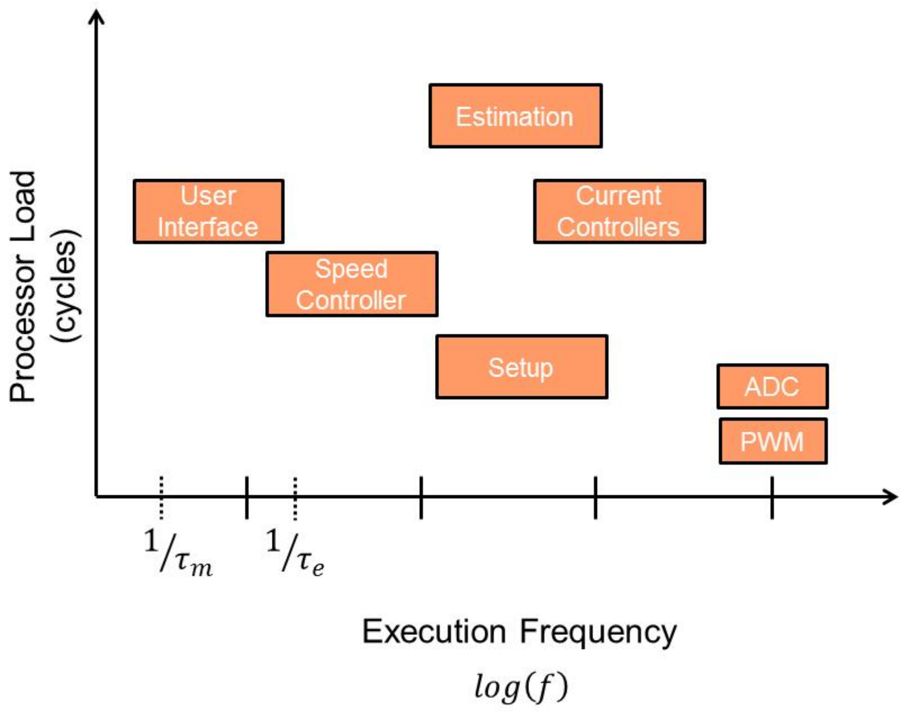


Figure 6 Process Load vs. Execution Frequency for Digital Control Algorithms

Figure 6 shows some typical algorithms that a MCU needs to execute in modern digital control systems. Each algorithm is represented in terms of its loading of the process as a function of algorithm execution frequency (i.e. how frequently the algorithm needs to be executed). For reference, the execution frequencies are given with respect to the mechanical time constant, τ_m, and the electrical time constant, τ_e, of a given motor. For example, the mechanical time constant of a motor is usually longer than the electrical time constant, thus the effective mechanical frequency ($^1/_{\tau_m}$) is less than the effective electrical frequency ($^1/_{\tau_e}$). The speed controller typically runs 5x to 10x the effective mechanical frequency of the motor to ensure an acceptable speed response. Likewise, the current controllers run 5x to 10x the effective electrical frequency of the motor. Some applications implement a user interface such as a GUI to provide desired set points for position or speed, to observe internal variables, etc. but this module usually runs at a fairly slow frequency.
In addition to the conventional speed and current controller, most modern control systems have some type of estimation, whether it is initial parameter estimation, real time parameter estimation or both. These algorithms can be quite sophisticated and can use most of the processor horsepower when viewed on a percentage algorithm usage basis. A/D sampling and data conversion along with PWM are typically the fastest frequency events in the system but they do not require much compute horsepower. However, the entire control system is usually synchronized to these events and they will provide the periodic clocking mechanism for executing the other algorithms. Finally, periodic setup code is needed to modify parameters during control system execution. This code typically does not burden the system with a large compute load but must still execute at a high enough rate to impact the algorithms that rely upon the updated parameters.

The next few sections describe how the hardware and software can be configured on TI's F28x class of MCUs to accommodate the various execution frequencies needed by most modern digital control systems. More information related to the hardware registers mentioned in these sections can be found here [2,3,4].

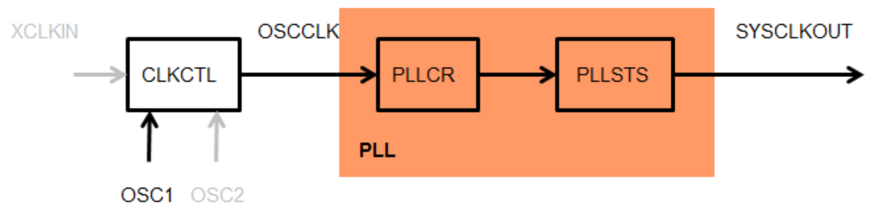


Figure 7 SYSCLKOUT Clock Tree

4.1.1 Hardware Timing Configuration

The reference oscillator used for clocking the MCU can either be an external oscillator or one of two internal oscillators, as shown in Figure 7. The Clock Control (CLKCTL) register can be used to select the one that is best for a given system. For most of the kits that TI provides, internal oscillator 1 (OSC1) is used as the reference clock. The output clock signal, denoted as OSCCLK, can then be decimated in a variety of ways using the Phase Lock Look Control (PLLCR) register and the Phase Lock Loop Status (PLLSTS) register. The resulting clock signal is the system clock, denoted as SYSCLKOUT. This clock becomes the reference clock to the ePWM module in the F28x MCU.

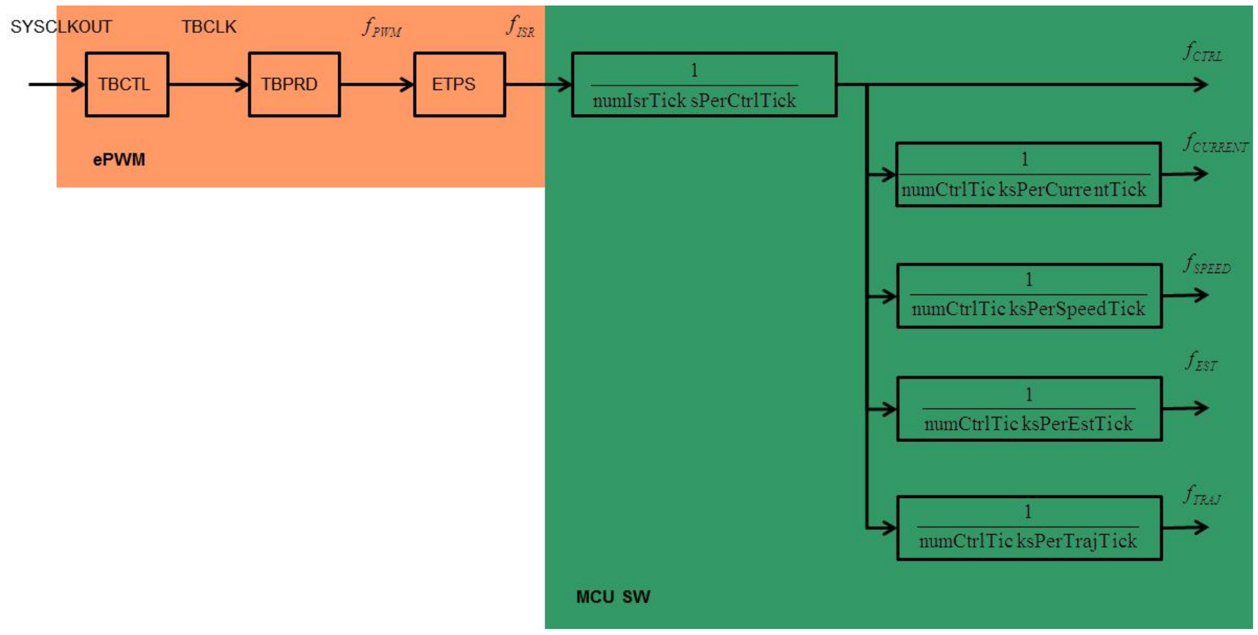

SYSCLKOUT
TBCLK
f_{PWM}
f_{ISR}
TBCTL
TBPRD
ETPS
ePWM
$\frac{1}{numIsrTicksPerCtrlTick}$
f_{CTRL}
$\frac{1}{numCtrlTicksPerCurrentTick}$
$f_{CURRENT}$
$\frac{1}{numCtrlTicksPerSpeedTick}$
f_{SPEED}
$\frac{1}{numCtrlTicksPerEstTick}$
f_{EST}
$\frac{1}{numCtrlTicksPerTrajTick}$
f_{TRAJ}
MCU SW

Figure 8 Hardware and Software Clock Tree

Figure 8 shows the system clock as the input clock to the ePWM module, whose timing reference is configured using the Time Base Control (TBCTL) register and the Time Base Period (TBPRD) register. These two registers configure the frequency by which the ePWM module operates.

The ePWM module can also generate an event trigger at a decimated rate from the PWM frequency. The Event Trigger Pre-Scale (ETPS) register is used to configure the decimation rate, which can be a value from one to three. This event trigger is used to trigger the A/D conversions in the system via the A/D Converter, Start of Conversion Control (ADCSOCxCTL) registers. After the last A/D conversion, an interrupt can be generated via the Interrupt Select (INTxSEL) register to denote that all of the A/D conversions are complete. This periodic interrupt triggers an ISR, which runs the digital motor control system software found in the MotorWare package.

NOTE: Currently in the MotorWare package, the ISR frequency, f_{ISR}, is always equal to the PWM frequency, f_{PWM}.

4.1.2 Software Execution Timing Configuration

The software in the MCU receives a periodic interrupt at a frequency denoted as f_{ISR} in Figure 8. This frequency can be decimated further using several factors in the software to provide various levels of execution granularity for different types of algorithms. These decimation factors are user configurable and can be found in the user.h file for a given project.

The first decimation factor is USER_NUM_ISR_TICKS_PER_CTRL_TICK. This factor decimates the interrupt frequency to create an execution frequency for the top-level controller. The controller execution frequency can be written in terms of the ISR frequency as follows

$$f_{CTRL} = \frac{1}{numIsrTicksPerCtrlTick} \cdot f_{ISR}$$

The next decimation factor is USER_NUM_CTRL_TICKS_PER_CURRENT_TICK. This factor decimates the top-level controller frequency to create an execution frequency for the current controllers. The current controller execution frequency can be written in terms of the ISR frequency as follows

$$f_{CURRENT} = \frac{1}{numCtrlTicksPerCurrentTick} \cdot \frac{1}{numIsrTicksPerCtrlTick} \cdot f_{ISR}$$

The next decimation factor is USER_NUM_CTRL_TICKS_PER_SPEED_TICK. This factor decimates the top-level controller frequency to create an execution frequency for the speed controller. The speed controller execution frequency can be written in terms of the ISR frequency as follows

$$f_{SPEED} = \frac{1}{numCtrlTicksPerSpeedTick} \cdot \frac{1}{numIsrTicksPerCtrlTick} \cdot f_{ISR}$$

The next decimation factor is USER_NUM_CTRL_TICKS_PER_EST_TICK. This factor decimates the top-level controller frequency to create an execution frequency for the estimator. The estimator execution frequency can be written in terms of the ISR frequency as follows

$$f_{EST} = \frac{1}{numCtrlTicksPerEstTick} \cdot \frac{1}{numIsrTicksPerCtrlTick} \cdot f_{ISR}$$

The last decimation factor is USER_NUM_CTRL_TICKS_PER_TRAJ_TICK. This factor decimates the top-level controller frequency to create an execution frequency for the trajectory generator. The trajectory generator execution frequency can be written in terms of the ISR frequency as follows

$$f_{TRAJ} = \frac{1}{numCtrlTicksPerTrajTick} \cdot \frac{1}{numIsrTicksPerCtrlTick} \cdot f_{ISR}$$

To implement the various frequencies in the control system, counters are coupled with the decimation factors in the software architecture. Access to the counter values is available using the following accessor functions:

```
CTRL_getCount_current()
CTRL_getCount_est()
CTRL_getCount_isr()
CTRL_getCount_speed()
CTRL_getCount_traj()
```

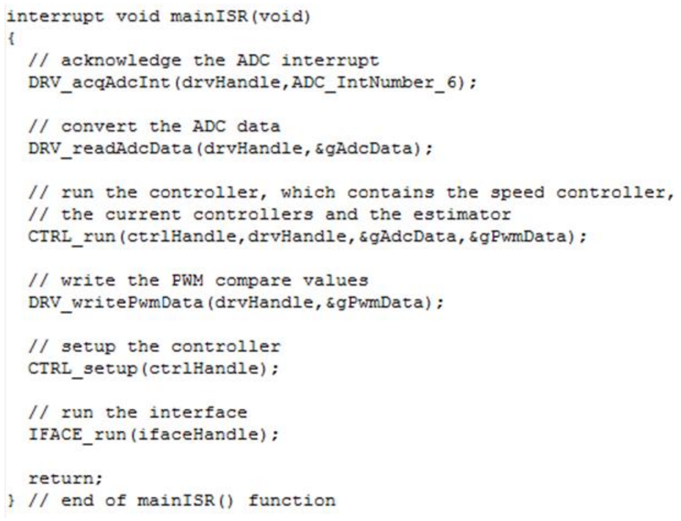
These functions can be used to read the counter values and easily determine if the software is stuck in a given configuration. If some part of the control system is not being given an opportunity to execute, the counter value will not be changing during real-time software execution, which often implies an invalid set of decimation factors.

4.2 Software Threads

Just as with the software execution frequency, the software architecture should scale with the number of software threads. A software thread is a set of software modules that can be managed independently in terms of their resource needs (e.g. data, memory, etc.). The threads can have different priority levels, which add to the software architecture complexity. This section presents how to partition software into different number of threads and discusses some of the tradeoffs between them.

4.2.1 Single Thread

A software architecture that is the easiest to visualize and to implement is one with just a single software thread. In this scenario, all of the software functions are run in the ISR itself. For clarity, an ISR is a function that executes each time an interrupt is set in the processor. The function is linked to the interrupt via an interrupt vector table, which is nothing more than a table of function addresses for the processor to reference when it receives and acknowledges a given interrupt. Basically, the processor receives an interrupt from a secondary source. Once the processor decides to acknowledge the interrupt, it looks in the interrupt vector table to determine which function to execute based upon the address in the table. For reference, this table is configured in MotorWare using the DRV_initIntVectorTable() function.



```
interrupt void mainISR(void)
{
  // acknowledge the ADC interrupt
  DRV_acqAdcInt(drvHandle,ADC_IntNumber_6);

  // convert the ADC data
  DRV_readAdcData(drvHandle,&gAdcData);

  // run the controller, which contains the speed controller,
  // the current controllers and the estimator
  CTRL_run(ctrlHandle,drvHandle,&gAdcData,&gPwmData);

  // write the PWM compare values
  DRV_writePwmData(drvHandle,&gPwmData);

  // setup the controller
  CTRL_setup(ctrlHandle);

  // run the interface
  IFACE_run(ifaceHandle);

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

Figure 9 Example ISR for Single Thread

The ISR might look like the mainISR() function shown in Figure 9. Since all the software in the mainISR() function must complete before the next interrupt, the timeline for the real-time software execution will look something like the one shown in Figure 10. After the last A/D conversion, an interrupt is generated and the mainISR() function executes. It is easy to conclude that the ISR frequency, and thus the PWM frequency, is limited by the execution time of the mainISR() function and the A/D conversion time. However, since not all of the software in the control system needs to be executed at the ISR rate, it could be run outside of the mainISR() function. The partitioning of the software based upon required execution frequency becomes the motivation for more sophisticated schemes of prioritizing software execution.

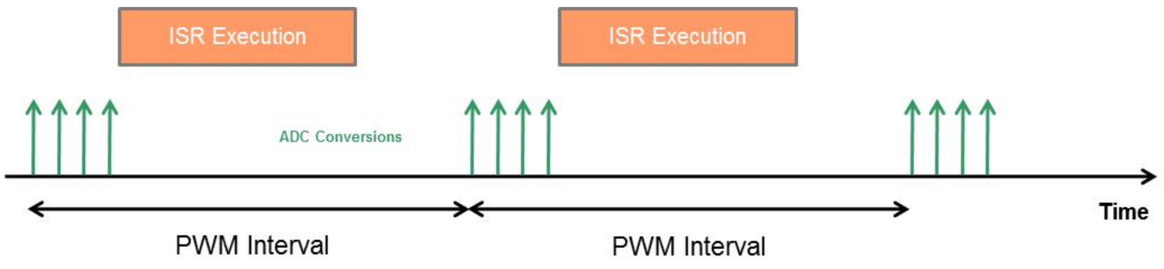
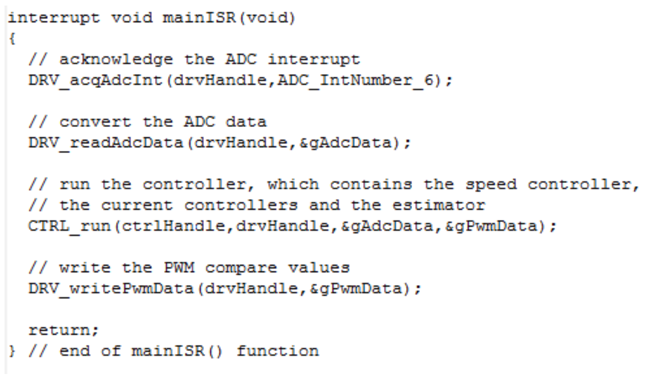
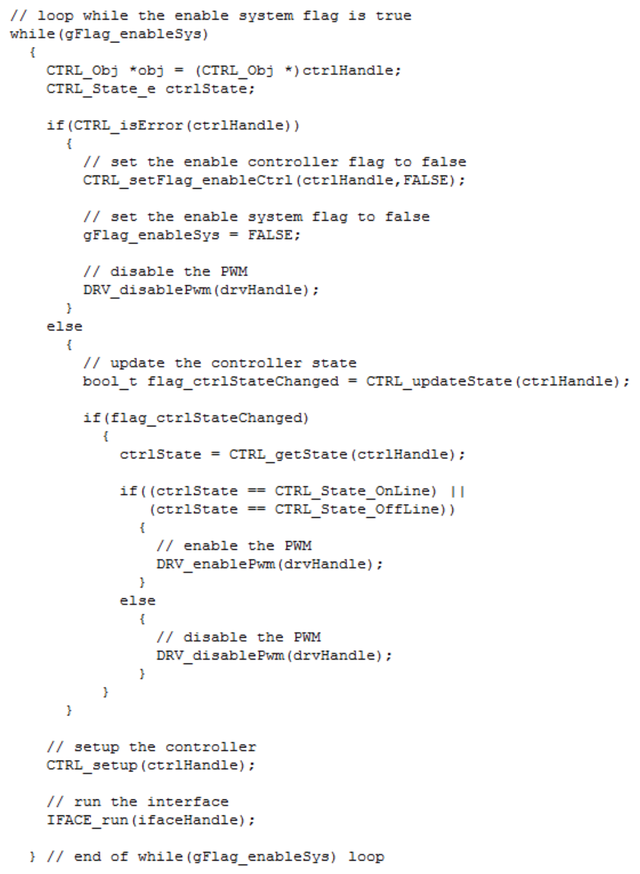


Figure 10 Real-Time Software Execution for a Single Thread (ISR Only)



```
interrupt void mainISR(void)
{
  // acknowledge the ADC interrupt
  DRV_acqAdcInt(drvHandle,ADC_IntNumber_6);

  // convert the ADC data
  DRV_readAdcData(drvHandle,&gAdcData);

  // run the controller, which contains the speed controller,
  // the current controllers and the estimator
  CTRL_run(ctrlHandle,drvHandle,&gAdcData,&gPwmData);

  // write the PWM compare values
  DRV_writePwmData(drvHandle,&gPwmData);

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

Figure 11 Example ISR for Two Threads



```
// loop while the enable system flag is true
while(gFlag_enableSys)
  {
    CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;
    CTRL_State_e ctrlState;

    if(CTRL_isError(ctrlHandle))
      {
        // set the enable controller flag to false
        CTRL_setFlag_enableCtrl(ctrlHandle,FALSE);

        // set the enable system flag to false
        gFlag_enableSys = FALSE;

        // disable the PWM
        DRV_disablePwm(drvHandle);
      }
    else
      {
        // update the controller state
        bool_t flag_ctrlStateChanged = CTRL_updateState(ctrlHandle);

        if(flag_ctrlStateChanged)
          {
            ctrlState = CTRL_getState(ctrlHandle);

            if((ctrlState == CTRL_State_OnLine) ||
               (ctrlState == CTRL_State_OffLine))
              {
                // enable the PWM
                DRV_enablePwm(drvHandle);
              }
            else
              {
                // disable the PWM
                DRV_disablePwm(drvHandle);
              }
          }
      }

    // setup the controller
    CTRL_setup(ctrlHandle);

    // run the interface
    IFACE_run(ifaceHandle);

  } // end of while(gFlag_enableSys) loop
```

Figure 12 Example Background Loop for Two Threads

4.2.2 Two Threads

The software architecture with the next level of complexity is one with two threads. One thread is commonly called the foreground task, which is high priority, and one thread is commonly called the background task, which is low priority. All of the software that needs to be executed at a high frequency remains in the ISR and all of the software that can be executed at a lower frequency is placed in a second function that runs in the background. Whenever an interrupt is received, the background function is interrupted by the ISR, the ISR runs to completion and then the background function continues to execute. It is not difficult to visualize that the background function must contain some sort of infinite loop to provide recurring execution functionality for the background software. As a result, the mainISR() function and the background loop should look something like those found in Figure 11 and Figure 12. Notice that the CTRL_setup() function and the IFACE_run() function have been moved to the background function to reduce the execution time of the mainISR() function.

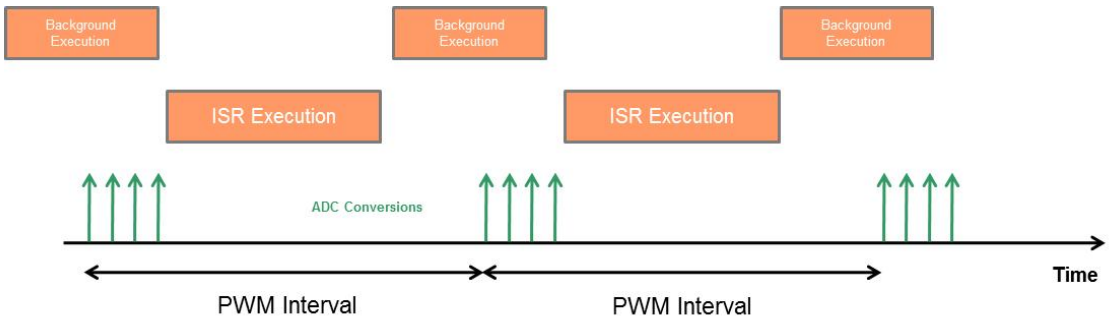
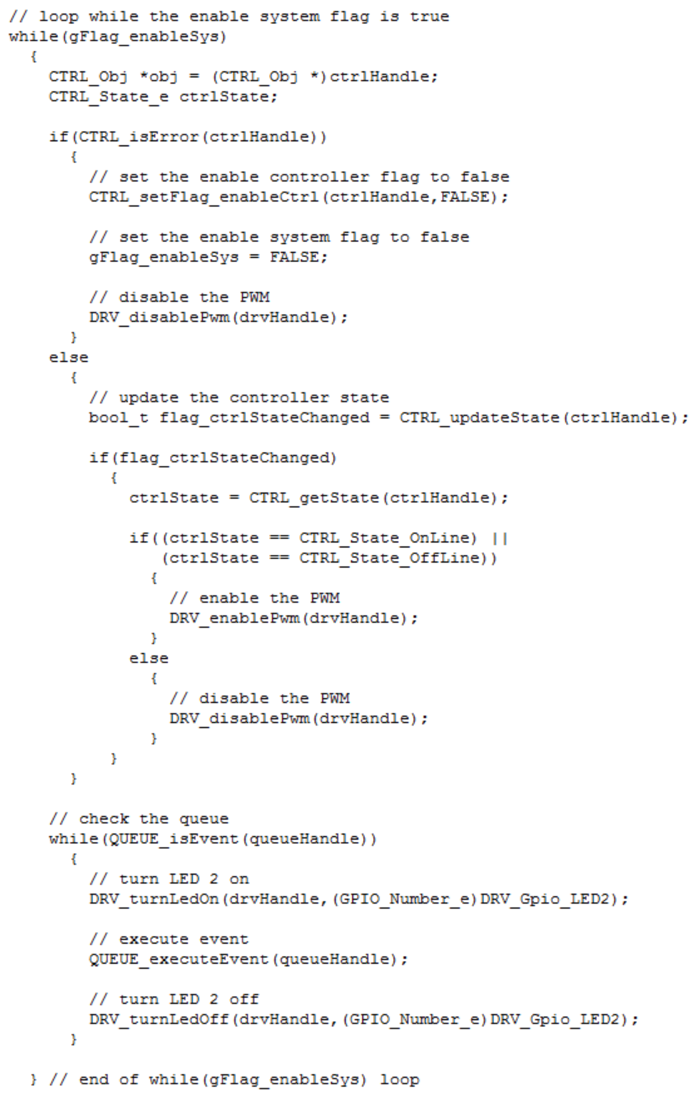


Figure 13 Real-Time Software Execution for Two Threads (ISR and Background)

The timeline for the real-time software execution for a background loop and an ISR function will look something like the one shown in Figure 13. It is clear to see that the background execution gets interrupted by the ISR at each interrupt instant, which occurs after the A/D conversions. While this architecture will allow for a higher interrupt frequency (and thus a higher PWM frequency) than a single thread architecture, the maximum achievable interrupt frequency is still quite limited because of the computational needs of the ISR function.

```
interrupt void mainISR(void)
{
  uint_least16_t count_isr;
  uint_least16_t numIsrTicksPerCtrlTick = CTRL_getNumIsrTicksPerCtrlTick(ctrlHandle);

  // acknowledge the ADC interrupt
  DRV_acqAdcInt(drvHandle,ADC_IntNumber_6);

  // increment the isr counter
  CTRL_incrCounter_isr(ctrlHandle);

  // get the ISR count
  count_isr = CTRL_getCount_isr(ctrlHandle);

  // if needed, run the code
  if(count_isr >= numIsrTicksPerCtrlTick)
    {
      EVENT_ArgList argList;

      // reset the isr count
      CTRL_resetCounter_isr_user(ctrlHandle);

      // convert the ADC data
      DRV_readAdcData(drvHandle,&gAdcData[gAdcDataBufferIndex]);

      // post the mainCode() to the queue
      argList.arg[0] = &gAdcData[gAdcDataBufferIndex];
      argList.arg[1] = NULL;
      argList.arg[2] = NULL;
      argList.arg[3] = NULL;
      QUEUE_postEventLast(queueHandle,(EVENT_Fxn)mainCode,&argList,1);

      // increment the data buffer index
      incrAdcDataBufferIndex();
    }

  return;
} // end of mainISR() function


void mainCode(DRV_AdcData_t *pAdcData)
{
  // run the controller, which contains the speed controller,
  // the current controllers and the estimator
  CTRL_run(ctrlHandle,drvHandle,pAdcData,&gPwmData);

  // write the PWM compare values
  DRV_writePwmData(drvHandle,&gPwmData);

  // write the DAC data
  DRV_writeDacData(drvHandle,&gDacData);

  // setup the controller
  CTRL_setup(ctrlHandle);

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

Figure 14 Example ISR and Queue Posting



```
// loop while the enable system flag is true
while(gFlag_enableSys)
  {
    CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;
    CTRL_State_e ctrlState;

    if(CTRL_isError(ctrlHandle))
      {
        // set the enable controller flag to false
        CTRL_setFlag_enableCtrl(ctrlHandle,FALSE);

        // set the enable system flag to false
        gFlag_enableSys = FALSE;

        // disable the PWM
        DRV_disablePwm(drvHandle);
      }
    else
      {
        // update the controller state
        bool_t flag_ctrlStateChanged = CTRL_updateState(ctrlHandle);

        if(flag_ctrlStateChanged)
          {
            ctrlState = CTRL_getState(ctrlHandle);

            if((ctrlState == CTRL_State_OnLine) ||
               (ctrlState == CTRL_State_OffLine))
              {
                // enable the PWM
                DRV_enablePwm(drvHandle);
              }
            else
              {
                // disable the PWM
                DRV_disablePwm(drvHandle);
              }
          }
      }

    // check the queue
    while(QUEUE_isEvent(queueHandle))
      {
        // turn LED 2 on
        DRV_turnLedOn(drvHandle,(GPIO_Number_e)DRV_Gpio_LED2);

        // execute event
        QUEUE_executeEvent(queueHandle);

        // turn LED 2 off
        DRV_turnLedOff(drvHandle,(GPIO_Number_e)DRV_Gpio_LED2);
      }

  } // end of while(gFlag_enableSys) loop
```

Figure 15 Example Background Loop with Queue Checking

A better software architecture for two levels of priority is to add queue processing to a background task and post a thread from the ISR into a queue. The thread would contain all of the cycle intensive software as shown in Figure 14. Figure 15 shows the resulting background loop for this software architecture. In this design, the ISR processor load requirements are quite small, which means much higher interrupt frequencies. The higher interrupt frequencies translates into higher PWM rates for demanding control applications where high current sampling rates are needed and minimizing current ripple is desired. In this architecture, the decimations factors discussed in Section 5.1.1 must be configured properly to ensure that the execution of the control system happens the way that the software is configured.

One way to verify the software execution is to toggle GPIO pins on a processor controlCARD and record the waveforms using a digital oscilloscope. For the sake of an example, the software timing parameters have been set as follows:

```
USER_PWM_FREQ_kHz = 40 kHz
USER_NUM_ISR_TICKS_PER_CTRL_TICK  = 4
USER_NUM_CTRL_TICKS_PER_CURRENT_TICK = 1
USER_NUM_CTRL_TICKS_PER_SPEED_TICK = 10
USER_NUM_CTRL_TICKS_PER_EST_TICK = 1
```

and the interrupt vector table values have been set as follows:

```
EPWM1_INT = &led2ISR()
ADCINT1 = &led3OffISR()
ADCINT2 = &led3OnISR()
ADCINT6 = &mainISR()
```

With this configuration the controller execution frequency becomes 10 kHz. The resulting current controller execution frequency is 10 kHz, the resulting estimator execution frequency becomes 10 kHz and the resulting speed controller execution frequency becomes 1 kHz.

The interrupt vector table configuration allows for some real-time monitoring of the software activity. For example, the EPWM1_INT interrupt calls the led2ISR() function, which will turn controlCARD LED2 on and off to denote the start of A/D conversions. LED2 is also turned on and off each time software is executed from the queue, as can be seen from the background software in Figure 15. The ADCINT1 interrupt calls the led3OffISR() function and the ADCINT6 interrupt calls the led3OnISR() function to toggle the LED on and off to denote the start of the last A/D conversion for a given PWM cycle. Finally, the ADCINT6 interrupt calls the mainISR() function in Figure 14.

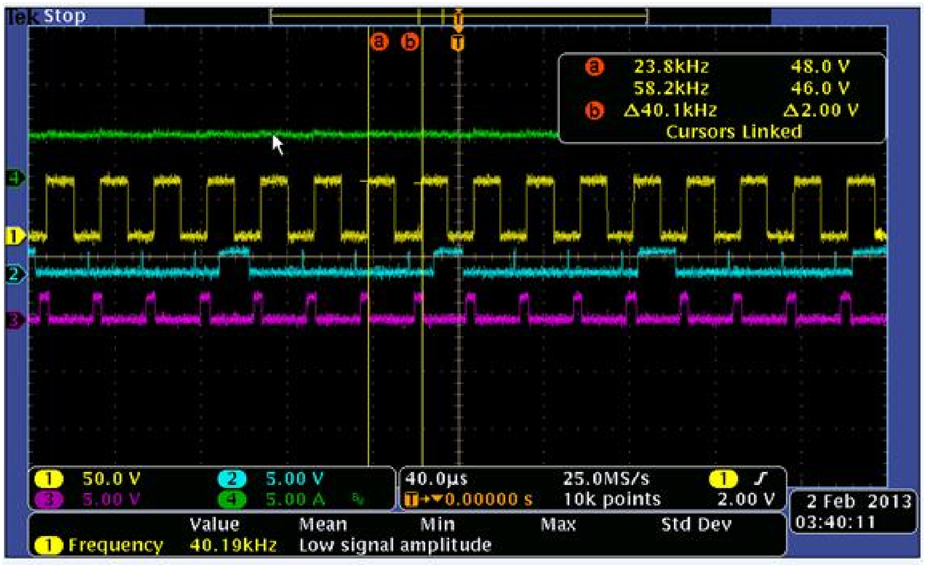


Figure 16 Oscilloscope Screen Capture for Multi-Rate Control Example

Figure 16 shows an oscilloscope screen capture which contains the Phase A current waveform (green), the Phase A PWM waveform (yellow), the LED2 waveform (blue) and the LED3 waveform (magenta) for a PM motor. Notice that the A/D conversion start happen during the dead-time of the PWM waveform and the mainISR() function executes soon after this start time. The duration of the mainISR() function is quite short (< 10 usec) so this function can be used for PWM frequencies up to 100 kHz without violating real-time timing constraints. Also notice that the controller runs every fourth PWM period, as prescribed by the USER_NUM_ISR_TICKS_PER_CTRL_TICK parameter. Since the real-time timing constraints are met and match the user configuration, the control system will function properly from a timing perspective.

4.2.3 Multiple Threads

The software architecture that offers the most execution flexibility is a RTOS. In this type of architecture, a more granular execution strategy can be configured using an execution priority level for each software thread. The nice part about using an RTOS is that it takes care of scheduling the thread execution. The only issue is that there has to be enough processor cycles available to accommodate the timing configuration for all of the software threads. It can be very easy to configure a given timing configuration in the software but it can be difficult to ensure that the proper timing is always met. Due to the added complexity of multiple thread software architectures, the various control path architectures will not be presented in this document.

5 Data Path Architecture

This section describes the data path architecture for the digital control system used in MotorWare. Different aspects of the design such as data integrity, data conversion and software thread data dependencies are discussed.

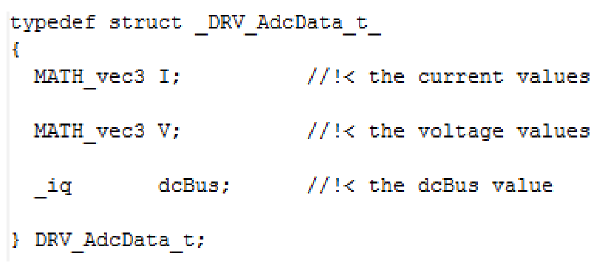
5.1 System Timing Configuration

The main concern for the data path architecture in a digital control system with different levels of hardware and software execution configurations is data integrity. The data for a given iteration of the controller and/or estimator must be the same for the entire execution cycle of the algorithm. Otherwise, the control system will be operating on a data set that is not consistent in terms of its voltage versus current relationship. For example, assume that the control system begins its computations with a given set of current and voltage samples. Part way thru the computations, a new set of samples arrives and the remaining controller computations use the new samples. If the computations for the first part of the control system involve voltage values, values from the first data set would be used. If the computations for the remaining portion of the control system involve current values, values from the second data set would be used. The result is that the control system used an inconsistent data set because the voltage and current samples do not conform to a mathematical model because they are taken at different time samples. In some instances, this issue is minor. In others, it is not.

There is a simple solution to this problem - data buffering. Data buffering is required as the complexity of the control system increases from a single thread implementation to a more complicated, multiple thread control system. The next few sections describe the data vectors and when it makes sense to buffer these vectors.

5.1.1 Global Data Structures

A driver function called DRV_readAdcData() transforms the raw bits from each A/D converter into useful input data for the digital control system. A global data vector named gAdcData is used to store the converted A/D data values after they have been transformed to signed, per unit values. The gAdcData variable is of type DRV_AdcData_t, which is a C structure of the form



```
typedef struct _DRV_AdcData_t_
{
  MATH_vec3 I;          //!< the current values

  MATH_vec3 V;          //!< the voltage values

  _iq       dcBus;      //!< the dcBus value

} DRV_AdcData_t;
```

A driver function called DRV_writePwmData() writes the PWM data to the PWM hardware. A global data vector named gPwmData is used to store the set of time durations computed by the digital control system. By using only one global variable, the most recent controller output value are always available, even if the controller did not execute during a given ISR or if the controller has not completed executing during a given PWM interval. The gPwmData variable is of type DRV_PwmData_t, which is a C structure of the form

```
typedef struct _DRV_PwmData_t_
{
  MATH_vec3  Tabc;      //!< the PWM time-durations for each motor phase

} DRV_PwmData_t;
```

5.2 Software Threads

Just as with control path architecture, the data path architecture should scale with the number of software threads. The threads can have different priority levels, which add to the software architecture complexity. This section presents how to scale the data path as the number of threads grows.

5.2.1 Single Thread

For the software architecture with just a single thread, the data path is quite simple. The DRV_readAdcData() function transforms the raw bits from each A/D converter into useful input data for the digital control system.   The controller takes these A/D data values as input, runs the speed and current controllers, runs the estimator and produces a PWM control signal, which is a set of time durations for the ePWM module. The DRV_writePwmData() function is then called to write the PWM data to the ePWM module.

5.2.2 Two Threads

The data path architecture for two threads can be the same as the architecture for one thread, as long as the execution latency does not compromise the integrity of the sampled data (e.g. the global data vector gets overwritten during controller execution). If latency becomes an issue, then the A/D data values must be buffered to ensure data integrity. Buffering is easily handled with an array of data vectors. In some of the example projects, A/D data values are stored in an array named gAdcData[NUM_ADC_DATA_BUFFERS]. An global variable named gAdcDataBufferIndex is used keep track of the current write index into the array and a function called incrAdcDataBufferIndex() is used to properly increment the index value. The desired controller frequency and the other processor load requirements will determine the size of the array (i.e. the value of NUM_ADC_DATA_BUFFERS).

Once queue processing is added to the software architecture, the A/D data values must be buffered. An array of A/D data values can be used to store the converted A/D data values to ensure data integrity.

5.2.3 Multiple Threads

For the multiple thread software architecture, an array of data should be used for the input and output of each thread. This configuration will ensure data integrity for each thread. The size of each data buffer will depend upon the execution frequency of each thread and the data dependencies between threads. Due to the complexity of multiple thread software architectures, the various data path architectures will not be presented in this document.

6 Memory Architecture

This section describes the memory allocation, initialization and partitioning for the MotorWare software.

6.1 Object Memory Allocation

The IFACE, CTRL and DRV object definitions are static memory allocations. That is, the objects are pre-defined in the software and are not dynamically allocated during runtime. The reason is that MCU memory is limited and no one is willing to waste code space on dynamic memory allocation functions. As a result, many MCU software libraries do not even contain the common memory functions such as malloc() and free().

6.2 Initialization

All of the projects in the MotorWare package initialize the software using a global variable name gUserParams. This variable is of type USER_Params_t, which is a C structure defined in the user.h file. Once this data structure is initialized, it is used to initialize all of the modules in the digital control system.

6.3 ROM Partitioning

The software partition for a typical digital motor control application using InstaSPIN-FOC is shown in Figure 17. The user software and public objects resides in unsecure memory, either flash and/or RAM. The InstaSPIN-FOC algorithms reside in secure memory, either flash, RAM or ROM, depending on the device.

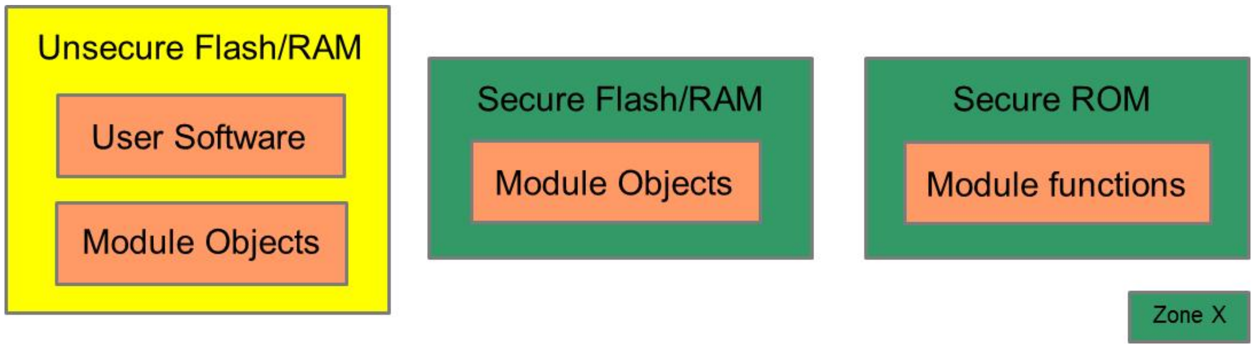


Figure 17 MotorWare ROM Partitioning

7 Flow Diagrams

This section describes the flow diagrams for the controller and the estimator.

7.1 Controller Flow Diagram

The flow diagram for the controller is shown in Figure 18. From the Start location, the controller immediately goes into the Idle state. Until the system and controller are enabled, the controller stays in the Idle state. Once these system and controller have been enabled, the controller checks to see if pre-determined motor parameters are supposed to be used or if the estimator is supposed to be executed. One the motor is identified, the controller goes into the Run state. During motor identification, various situations can cause the controller to transition into the Error state. By looking at the error condition, one can determine what caused the controller to reach the Error state. Once the controller is in the Run state, the user can disable the controller and return it to the Idle State.

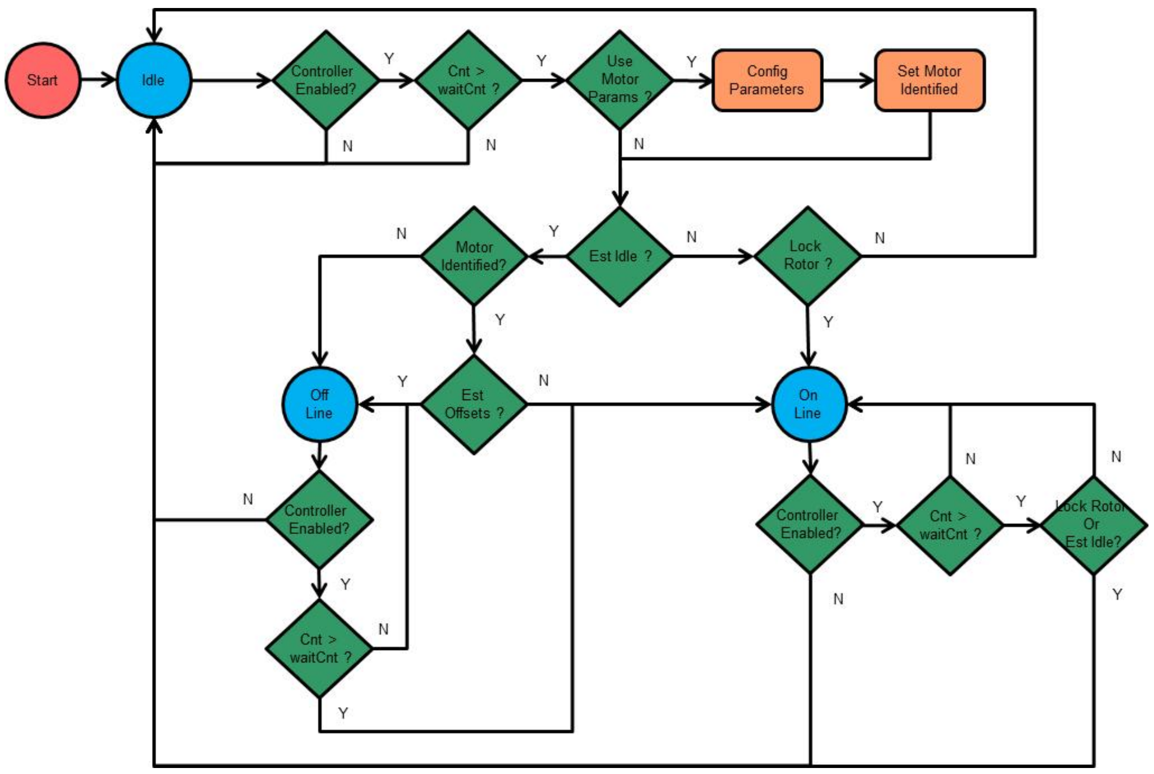


Figure 18 Controller State Transition Diagram

7.2 Estimator Diagram

The flow diagram for the estimator is shown in Figure 19. From the Start location, the estimator immediately goes into the Idle state. Until the system and controller are enabled, the estimator stays in this state. Once these systems have been enabled, the estimator performs ADC offset estimation. If the motor is not identified, the estimator performs high frequency resistance and inductance estimation followed by stator resistance estimation. If the motor has been previously identified, the previously identified stator resistance is finely tuned to account for any temperature effects due to previous estimator runs.

Once the stator resistance is known, the estimator transitions to the Done state if the motor has been identified. Otherwise, the estimator begins to spin the motor during the RampUp state. For ACIM, the rated flux is determined in the next state (FluxRated_OL). For PMSM, the rated Id current is determined in the IdRated state. The rated flux for either motor type is then determined using a closed loop routine in the FluxRated_CL1 state. For ACIM, a second closed loop routine is performed during the

FluxRated_CL2 state. This state is followed by a fine flux correction routine in the FluxFine state. The second closed loop routine is not used for PMSM and the fine flux correction is run directly after the first routine.

At this point the flow diagram, the motor stops turning for the ACIM and goes to the LockRotor state. At this point, the user should lock the rotor before re-enabling the controller. After the ACIM rotor shaft is locked and the controller has been enabled, the controller performs inductance estimation followed by rotor resistance estimation. For PMSM, the LockRotor state is bypassed altogether and inductance estimation is performed. Once the estimator has reached this point, the motor has been identified. The motor stops turning and the controller is disabled. Once the controller is re-enabled, the estimator returns to the Offset state and performs ADC offset estimation and resistance fine tuning before reaching the Done state.

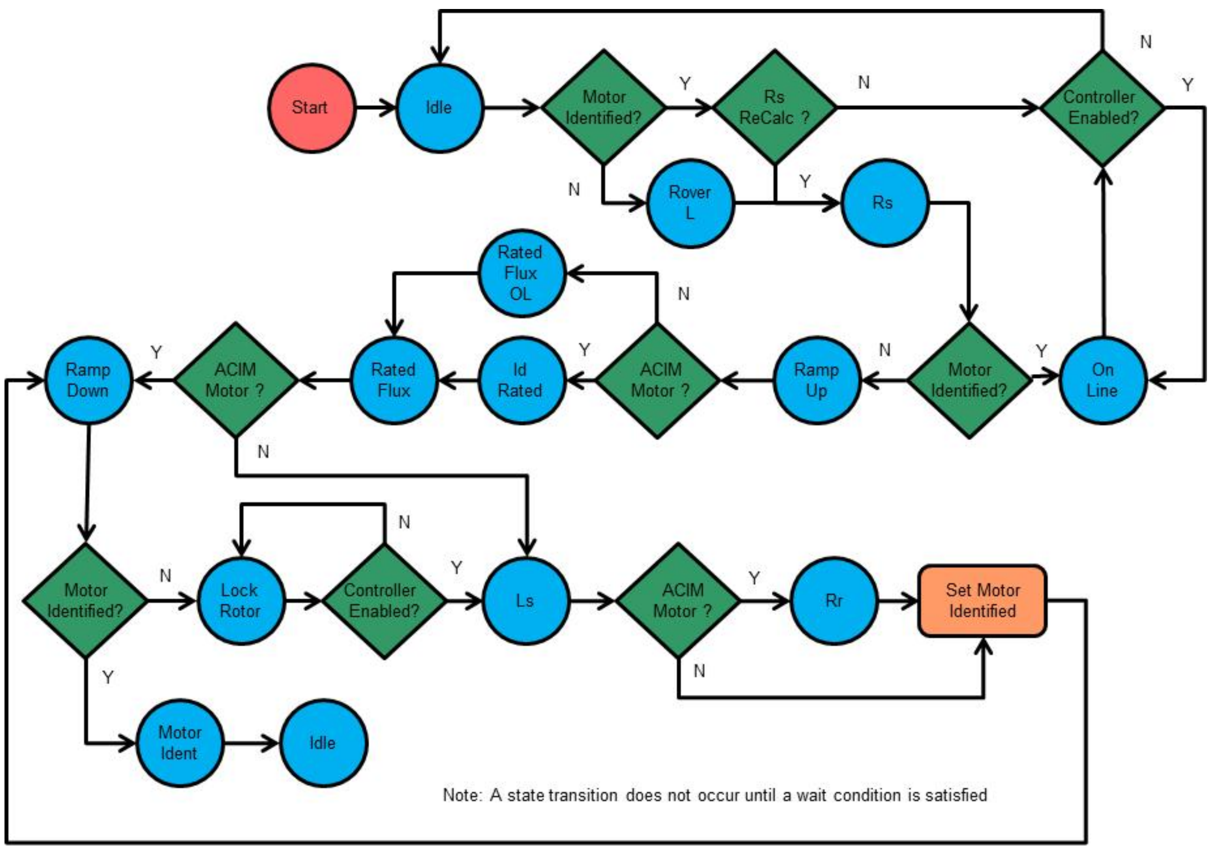


Figure 19 Estimator State Transition Diagram

8 Error Handling

This section describes the error handling mechanism used in the MotorWare software architecture. All of the error checking is centralized in one function and this function will report an error if one is found. As part of the module execution, this function is called from the function that updates the state of the module. If an error is identified, the module reports an error and changes its state to an error state, which will prevent the module from functioning.

8.1 Controller Errors

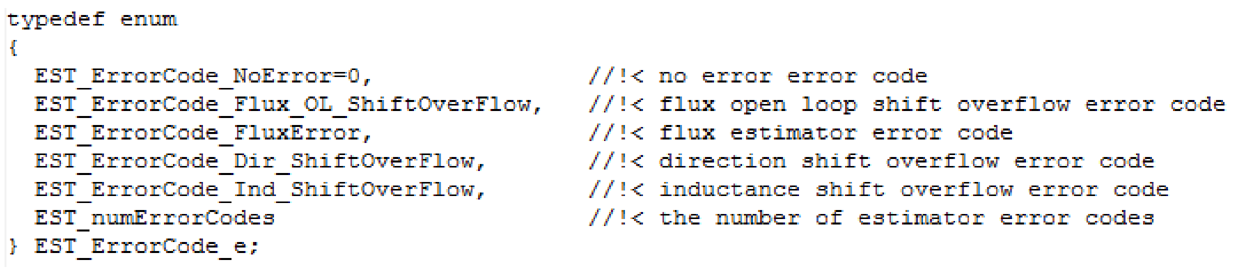
The controller error handling is managed through the CTRL_isError() function in the background thread. During each iteration of the background thread, the controller checked for errors by calling the CTRL_checkForErrors() function from within the CTRL_updateState() function. If an error is found by the CTRL_checkForErrors() function, a flag is set which is read by the CTRL_isError() function. Once an error is detected, the control system is disabled and the PWM driver is disabled.

Currently, there are only two types of errors identified by the controller, clipping of the Id current and an estimator. Clipping of the Id current can happened during parameter estimation and is reported by the controller. To determine the type of estimator error, the estimator must be queried using the EST_getErrorCode() function.

8.2 Estimator Errors

The estimator error handling is managed through the EST_isError() function within the estimator. During each iteration of the estimator, the estimator checks for errors by calling the EST_checkForErrors() function from within the EST_updateState() function. If an error is found by the EST_checkForErrors() function, a flag is set which is read the EST_isError() function. Once an error is detected, a flag is set with is detected by the CTRL_checkForErrors() function.

The errors that are detected by the estimator are as follows



```
typedef enum
{
  EST_ErrorCode_NoError=0,              //!< no error error code
  EST_ErrorCode_Flux_OL_ShiftOverFlow,  //!< flux open loop shift overflow error code
  EST_ErrorCode_FluxError,              //!< flux estimator error code
  EST_ErrorCode_Dir_ShiftOverFlow,      //!< direction shift overflow error code
  EST_ErrorCode_Ind_ShiftOverFlow,      //!< inductance shift overflow error code
  EST_numErrorCodes                     //!< the number of estimator error codes
} EST_ErrorCode_e;
```

9 Summary

As was stated in the introduction, the purpose of a software architecture is to provide a consistent development, testing, and maintenance platform for a development team working on a common collection of software. In this case, the common collection of software happened to be the MotorWare package. Hopefully, this document has demonstrated how the MotorWare software architecture addresses those needs. In doing so, this document discussed some of the important features of the software architecture, which are

- Well-defined Software Stack – driver, controller, estimator
- Modular software methodology – object oriented programming model
- Improved code execution - Queues for priority based code execution, inlined functions
- Software portability across processors
- Software re-use
- Embedded software documentation - Doxygen
- Coding standard
- Single source files

and has hopefully demonstrated how the MotorWare package meets the software development needs for today's demanding digital motor control applications.

10 References

1. *MotorWare Coding Standard*, Texas Instruments Document, February 2013.
2. *TMS320F2803x Piccolo System Control and Interrupts Reference Guide*, Texas Instruments Document, Literature Number SPRUGL8B, December 2009.
3. *TMS320x2802x, 2803x Piccolo Enhanced Pulse Width Modulator (ePWM) Module Reference Guide,* Texas Instruments Document, Literature Number SPRUGE9D, February 2010.
4. *TMS320x2802x, 2803x Piccolo Analog-to-Digital Converter (ADC) and Comparator Reference Guide,* Texas Instruments Document, Literature Number SPRUGE5B, December 2009.