RDK-ACIM Firmware Development Package

USER'S GUIDE

SW-RDK-ACIM-UG-8555

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.ti.com/stellaris

Revision Information

This is version 8555 of this document, last updated on January 28, 2012.

Table of Contents

1 Introduction

1.1 Overview

This document describes the functions, variables, structures, and symbolic constants in the software for the AC induction motor RDK. The firmware runs on a Stellaris® LM3S818 microcontroller, utilizing the provided six-channel motion control PWM module, six-channel ADC, UART, and quadrature encoder module. The Stellaris Peripheral Driver Library is used to configure and operate these peripherals.

The AC induction motor application has the following features:

- Drives three-phase AC induction motors
- Drives permanent split capacitor and shaded-pole single-phase AC induction motors
- Operation from 0 to 400 Hz with 0.1 Hz steps
- Acceleration and deceleration from 1 to 100 Hz/sec
- PWM frequency of 8 KHz, 12.5 KHz, 16 KHz, or 20 KHz
- V/f control
- DC bus ripple compensation
- Sine wave modulation
- Space vector modulation
- Closed-loop speed control
- Dynamic braking
- DC injection braking
- DC bus regeneration control
- Real-time data monitoring
- Fault monitoring and handling

See the *Stellaris AC Induction Motor Reference Design Kit User's Manual* for details of these features, how to run the application, and details of the various motor drive parameters.

1.2 Code Size

The size of the final application binary depends upon the source code, the compiler used, and the version of the compiler. This makes it difficult to give an absolute size for the application since changes to any of these variables will likely change the size of the application (if only slightly).

Typical numbers for the application are 16 KB of flash and 1.25 KB of SRAM. Of this, 5.5 KB of flash and 0.5 KB of SRAM is consumed by the user interface, which is much more complicated than what would typically be used in a final motor drive application (unless the final application is a generic motor drive such as the RDK). This leaves 10.5 KB of flash and 0.75 KB of SRAM for the actual motor drive application, parts of which are also more complicated than required due to the run-time reconfigurability.

1.3 Processor Usage

The two factors that have the largest impact on the processor usage are the PWM frequency and the PWM update rate. The following table provides some data points for a few combinations of these two factors; the actual processor usage may vary slightly, especially when other parameters are changed (this should be viewed only as indicative information).

Update	PWM Frequency			
	8 KHz	12.5 KHz	16 KHz	20 KHz
1	19%	30%	38%	47%
2	14%	21%	26%	33%
3	12%	18%	22%	28%
4	11%	16%	20%	25%
5	10%	15%	19%	24%

An update rate of 1 means that the waveform update frequency matches the PWM frequency. From the table above, a PWM frequency of 20 KHz with a waveform update frequency of 20 KHz consumes around 47% of the processor. An update rate of 5, on the other hand, is a waveform update frequency of 1/5 the PWM frequency; a PWM frequency of 20 KHz with a waveform update rate of 4 KHz consumes around 24% of the processor.

1.4 Memory Layout

The AC induction motor firmware works in cooperation with the Stellaris boot loader to provide a means of updating the firmware over the serial port. When a request is made to perform a firmware update, the AC induction motor firmware transfers control back to the boot loader. After a reset, the boot loader runs and simply passes control to the AC induction motor firmware.

In addition to the boot loader and the AC induction motor firmware, there is a region of flash reserved for storing the motor drive parameters. This allows these values to be persistent between power cycles of the board, though they are only written to flash based on an explicit request.

The 64 KB of flash on the LM3S818 is organized as follows:

0x0000.0000 0x0000.07ff	Boot Loader
0x0000.0800 0x0000.efff	ACIM Firmware
0x0000.f000 0x0000.ffff	Parameter Storage

1.5 Debugging

The AC induction motor firmware is a difficult application to debug. If the firmware is stopped while the motor is running, the PWM outputs shut off, causing the motor to start coasting. When the firmware starts executing again (either because of a run or a single step), the PWM outputs start up again as if they had never stopped. But, the motor will be running much slower at this point, so the slip will have increased (possibly significantly). This likely results in either a power module fault or a motor over current fault. Both faults result in the motor drive being shut off, meaning that the debug event has caused catastrophic results on the behavior of the application (but no permanent hardware damage). The use (and abuse) of the real-time data stream is typically the best way to "debug" a running system like this; the abuse of the real-time data stream is to replace the current frequency real-time data item value (for example) with a different internal variable and view its value in real time on the GUI. Obviously not ideal, but this is better than nothing.

The JTAG opto-isolation interface provided on the RDK provides only unidirectional transmission of the JTAG signals, from board to emulator for TDO and from emulator to board for the remaining signals. This allows JTAG to operate with no problems, so long as the signal edge rate and propagation delay of the opto-isolators does not become an issue. If JTAG fails to operate, or operates in an intermittent fashion, a slower JTAG clock rate should be tried.

The new Serial Wire Debug (SWD) available in the ARM® Cortex™-M3 microprocessor provides a two-signal debugging interface (as opposed to the four- or five-signal interface required by JTAG). One of these signals is a clock that is provided by the emulator. The other signal is a bidirectional data signal whose direction is known based on the current state of the transfer protocol. SWD provides the same capabilities as JTAG with fewer pins consumed on the microcontroller.

As of November 2008, the Sourcery G++ USB Debug Sprite for Stellaris from CodeSourcery uses the SWD debug protocol to communicate with the Stellaris microcontroller. It is therefore unable to operate on the AC induction motor RDK. An alternative for advanced users is the open source project OpenOCD (`http://openocd.berlios.de/web`), which utilizes JTAG to communicate with the Stellaris microcontroller but continues to use a standard GNU toolchain (such as CodeSourcery's) for the compiler and debugger.

The use of OpenOCD is recommended only for those capable of building open source projects, following limited on-line tutorials, and utilizing on-line forums and mailing lists for help. `http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects/openocd_-cortex/index.html` provides a good starting point for the use of OpenOCD with the CodeSourcery toolchain and a Stellaris microcontroller.

2 Applications

The boot loader (boot_serial) and quickstart (qs-acim) are programmed onto the MDL-ACIM. The boot loader can be used to update the quickstart application using the serial port, eliminating the need for a JTAG debugger.

There is an IAR workspace file (`rdk-acim.eww`) that contains the peripheral driver library project, along with the A/C Induction Motor Controller software project, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is a Keil multi-project workspace file (`rdk-acim.mpw`) that contains the peripheral driver library project, along with the A/C Induction Motor Controller software project, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/rdk-acim` subdirectory of the firmware development package source distribution.

2.1 Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

2.2 A/C Induction Motor Controller (qs-acim)

This application is a motor drive for single and three phase A/C induction motors. The following features are supported:

- V/f control
- Sine wave modulation
- Space vector modulation
- Closed loop speed control
- DC bus voltage monitoring and control
- AC in-rush current control
- Regenerative braking control
- DC braking control
- Simple on-board user interface (via a potentiometer and push button)
- Comprehensive serial user interface
- Over 30 configurable drive parameters
- Persistent storage of drive parameters in flash

3 Development System Utilities

These are tools that run on the development system, not on the embedded target. They are provided to assist in the development of firmware for Stellaris microcontrollers.

These tools reside in the `tools` subdirectory of the firmware development package source distribution.

Serial Flash Downloader

Usage:

```
sflash [OPTION]... [INPUT FILE]
```

Description:

Downloads a firmware image to a Stellaris board using a UART connection to the Stellaris Serial Flash Loader or the Stellaris Boot Loader. This has the same capabilities as the serial download portion of the Stellaris Flash Programmer.

The source code for this utility is contained in `tools/sflash`, with a pre-built binary contained in `tools/bin`.

Arguments:

-b BAUD specifies the baud rate. If not specified, the default of 115,200 will be used.

-c PORT specifies the COM port. If not specified, the default of COM1 will be used.

-d disables auto-baud.

-h displays usage information.

-l FILENAME specifies the name of the boot loader image file.

-p ADDR specifies the address at which to program the firmware. If not specified, the default of 0 will be used.

-r ADDR specifies the address at which to start processor execution after the firmware has been downloaded. If not specified, the processor will be reset after the firmware has been downloaded.

-s SIZE specifies the size of the data packets used to download the firmware date. This must be a multiple of four between 8 and 252, inclusive. If using the Serial Flash Loader, the maximum value that can be used is 76. If using the Boot Loader, the maximum value that can be used is dependent upon the configuration of the Boot Loader. If not specified, the default of 8 will be used.

INPUT FILE specifies the name of the firmware image file.

Example:

The following will download a firmware image to the board over COM2 without auto-baud support:

```
sflash -c 2 -d image.bin
```

4 ADC Control

4.1 Introduction

The ADC is used to monitor the motor current, DC bus voltage, and ambient temperature of the microcontroller. Each of these values is sampled every PWM period based on a trigger from the PWM module, which allows the motor current to be measured when the low side switch for each phase is turned on.

Each reading from the ADC is passed through a single-pole IIR low pass filter. This helps to reduce the effects of high frequency noise (such as switching noise) on the sampled data. A coefficient of 0.75 is used to simplify the integer math (requiring only a multiplication by 3, an addition, and a division by four).

The measured current in each motor phase is passed through a peak detect that resets every cycle of the output motor drive waveforms. The peak value is then divided by the square root of 2 (approximated by 1.4) in order to obtain the RMS current of each phase of the motor. The RMS current of the motor is the average of the RMS current though each phase.

The individual motor phase RMS currents, motor RMS current, DC bus voltage, and ambient temperature are used outside this module.

The code for handling the ADC is contained in `adc_ctrl.c`, with `adc_ctrl.h` containing the definitions for the variables and functions exported to the remainder of the application.

4.2 Definitions

Functions

- void ADC0IntHandler (void)
- void ADCInit (void)

Variables

- static unsigned short g_pusFilteredData[5]
- volatile unsigned short g_pusPhaseCurrentRMS[3]
- static unsigned short g_pusPhaseMax[3]
- volatile short g_sAmbientTemp
- static unsigned long g_ulPrevAngle
- volatile unsigned short g_usBusVoltage
- volatile unsigned short g_usMotorCurrent

4.2.1 Function Documentation

4.2.1.1 ADC0IntHandler

Handles the ADC sample sequence zero interrupt.

Prototype:

```
void
ADC0IntHandler(void)
```

Description:

This function is called when sample sequence zero asserts an interrupt. It handles clearing the interrupt and processing the new ADC data in the FIFO.

Returns:

None.

4.2.1.2 ADCInit

Initializes the ADC control routines.

Prototype:

```
void
ADCInit(void)
```

Description:

This function initializes the ADC module and the control routines, preparing them to monitor currents and voltages on the motor drive.

Returns:

None.

4.2.2 Variable Documentation

4.2.2.1 g_pusFilteredData [static]

Definition:

```
static unsigned short g_pusFilteredData[5]
```

Description:

An array containing the raw low pass filtered ADC readings. This is maintained in a raw form since it is required as an input to the next iteration of the IIR low pass filter.

4.2.2.2 g_pusPhaseCurrentRMS

Definition:

```
volatile unsigned short g_pusPhaseCurrentRMS[3]
```

Description:
The RMS current passing through the three phases of the motor, specified in amperes as an unsigned 8.8 fixed point value.

4.2.2.3 g_pusPhaseMax [static]

Definition:
```
static unsigned short g_pusPhaseMax[3]
```

Description:
An array containing the maximum phase currents seen during the last half cycle of each phase. This is used to perform a peak detect on the phase currents.

4.2.2.4 g_sAmbientTemp

Definition:
```
volatile short g_sAmbientTemp
```

Description:
The ambient case temperature of the microcontroller, specified in degrees Celsius.

4.2.2.5 g_ulPrevAngle [static]

Definition:
```
static unsigned long g_ulPrevAngle
```

Description:
The angle of the motor drive on the previous ADC interrupt.

4.2.2.6 g_usBusVoltage

Definition:
```
volatile unsigned short g_usBusVoltage
```

Description:
The DC bus voltage, specified in volts.

4.2.2.7 g_usMotorCurrent

Definition:
```
volatile unsigned short g_usMotorCurrent
```

Description:
The total RMS current passing through the motor, specified in amperes as an unsigned 8.8 fixed point value.

5 Dynamic Brake Control

5.1 Introduction

Dynamic braking is the application of a power resistor across the DC bus in order to control the increase in the DC bus voltage. The power resistor reduces the DC bus voltage by converting current into heat.

The dynamic braking routine is called every millisecond to monitor the DC bus voltage and handle the dynamic brake. When the DC bus voltage gets too high, the dynamic brake is applied to the DC bus. When the DC bus voltage drops enough, the dynamic brake is removed.

In order to control heat buildup in the power resistor, the amount of time the brake is applied is tracked. If the brake is applied for too long, it will be forced off for a period of time (regardless of the DC bus voltage) to prevent it from overheating. The amount of time on and off is tracked as an indirect measure of the heat buildup in the power resistor; the heat increases when on and decreases when off.

The code for handling dynamic braking is contained in `brake.c`, with `brake.h` containing the definitions for the functions exported to the remainder of the application.

5.2 Definitions

Defines

- STATE_BRAKE_COOL
- STATE_BRAKE_OFF
- STATE_BRAKE_ON

Functions

- void BrakeInit (void)
- void BrakeTick (void)

Variables

- static unsigned long g_ulBrakeCount
- static unsigned long g_ulBrakeState

5.2.1 Define Documentation

5.2.1.1 STATE_BRAKE_COOL

Definition:

```
#define STATE_BRAKE_COOL
```

Description:

The dynamic brake is forced off to allow the power resistor to cool. After the minimum cooling period has expired, an automatic transition to STATE_BRAKE_OFF will occur if the bus voltage is below the trigger level and to STATE_BRAKE_ON if the bus voltage is above the trigger level.

5.2.1.2 STATE_BRAKE_OFF

Definition:

```
#define STATE_BRAKE_OFF
```

Description:

The dynamic brake is turned off. The bus voltage going above the trigger level will cause a transition to the STATE_BRAKE_ON state.

5.2.1.3 STATE_BRAKE_ON

Definition:

```
#define STATE_BRAKE_ON
```

Description:

The dynamic brake is turned on. The bus voltage going below the trigger level will cause a transition to the STATE_BRAKE_OFF state, and the brake being on for too long will cause a transition to STATE_BRAKE_COOL.

5.2.2 Function Documentation

5.2.2.1 BrakeInit

Initializes the dynamic braking control routines.

Prototype:

```
void
BrakeInit(void)
```

Description:

This function initializes the ADC module and the control routines, preparing them to monitor currents and voltages on the motor drive.

Returns:

None.

5.2.2.2 BrakeTick

Updates the dynamic brake.

Prototype:

```
void
BrakeTick(void)
```

Description:

This function will update the state of the dynamic brake. It must be called at the PWM frequency to provide a time base for determining when to turn off the brake to avoid overheating.

Returns:

None.

5.2.3 Variable Documentation

5.2.3.1 g_ulBrakeCount [static]

Definition:

```
static unsigned long g_ulBrakeCount
```

Description:

The number of milliseconds that the dynamic brake has been on. For each brake update period, this is incremented if the brake is on and decremented if it is off. This effectively represents the heat buildup in the power resistor; when on heat will increase and when off it will decrease.

5.2.3.2 g_ulBrakeState [static]

Definition:

```
static unsigned long g_ulBrakeState
```

Description:

The current state of the dynamic brake. Will be one of STATE_BRAKE_OFF, STATE_BRAKE_ON, or STATE_BRAKE_COOL.

6 Faults

6.1 Introduction

There are several fault conditions that can occur during the operation of the motor drive. Those fault conditions are enumerated here and provide the definition of the fault status read-only parameter and real-time data item.

The faults are:

- Emergency stop: This occurs as a result of a command request. An emergency stop is one where the motor is stopped immediately without regard for trying to maintain normal control of it (this is, without the normal deceleration ramp). From the motor drive perspective, the motor is left to its own devices to stop, meaning it will coast to a stop under the influence of friction unless a mechanical braking mechanism is provided.

- DC bus under-voltage: This occurs when the voltage level of the DC bus drops too low. Typically, this is the result of the loss of mains power.

- DC bus over-voltage: This occurs when the voltage level of the DC bus rises too high. When the motor is being decelerated, it becomes a generator, increasing the voltage level of the DC bus. If the level of regeneration is more than can be controlled, the DC bus will rise to a dangerous level and could damage components on the board.

- Motor under-current: This occurs when the current through the motor drops too low. Typically, this is the result of an open connection to the motor.

- Motor over-current: This occurs when the current through the motor rises too high. When the motor is being accelerated, more current flows through the windings than when running at a set speed. If accelerated too quickly, the current through the motor may rise above the current rating of the motor or of the motor drive, possibly damaging either.

- Power module fault: This occurs when the smart power module detects a fault condition. The smart power module will signal a fault on a supply under-voltage condition and an output over-current condition.

- Ambient over-temperature: This occurs when the case temperature of the microcontrollers rises too high. The motor drive generates lots of heat; if in an enclosure with inadequate ventilation, the heat could rise high enough to exceed the operating range of the motor drive components and/or cause physical damage to the board. Note that the temperature measurement that is of more interest is directly on the heat sink where the smart power module is attached, though this would require an external thermocouple in order to measure.

The definitions for the fault conditions are contained in `faults.h`.

6.2 Definitions

Defines

6.2.1 Define Documentation

6.2.1.1 FAULT_CURRENT_HIGH

Definition:

```
#define FAULT_CURRENT_HIGH
```

Description:

The fault flag that indicates that the motor current rose too high.

6.2.1.2 FAULT_CURRENT_LOW

Definition:

```
#define FAULT_CURRENT_LOW
```

Description:

The fault flag that indicates that the motor current dropped too low.

6.2.1.3 FAULT_EMERGENCY_STOP

Definition:

```
#define FAULT_EMERGENCY_STOP
```

Description:

The fault flag that indicates that an emergency stop operation was performed.

6.2.1.4 FAULT_POWER_MODULE

Definition:

```
#define FAULT_POWER_MODULE
```

Description:

The fault flag that indicates that the power module asserted its fault signal.

6.2.1.5 FAULT_TEMPERATURE_HIGH

Definition:

```
#define FAULT_TEMPERATURE_HIGH
```

Description:

The fault flag that indicates that the ambient temperature rose too high.

6.2.1.6 FAULT_VBUS_HIGH

Definition:

```
#define FAULT_VBUS_HIGH
```

Description:

The fault flag that indicates that the DC bus voltage rose too high.

6.2.1.7 FAULT_VBUS_LOW

Definition:

```
#define FAULT_VBUS_LOW
```

Description:

The fault flag that indicates that the DC bus voltage dropped too low.

7 In-rush Current Control

7.1 Introduction

On initial power-up, an in-rush current limiting resistor is applied in series with the AC power line input. This slows the flow of current into the DC bus capacitors, preventing damage to the power supply section of the board.

Once the DC bus voltage reaches a reasonable level (200 V), the in-rush resistor is bypassed by closing a relay. At this point, the DC bus voltage quickly rises to its operating level.

This current limiting function is a one-time process that occurs when the application first starts. The in-rush resistor is sized such that it could remain active for extended periods of time (for example, if the flash of the microcontroller is erased and there is no code to turn on the relay). The motor should never be run when the in-rush resistor is active.

The code for handling in-rush current limiting is contained in `inrush.c`, with `inrush.h` containing the definition for the functions exported to the remainder of the application.

7.2 Definitions

Functions

- void InRushDelay (void)
- void InRushRelayAdjust (void)

7.2.1 Function Documentation

7.2.1.1 InRushDelay

Handles the in-rush current control.

Prototype:

```
void
InRushDelay(void)
```

Description:

This function delays while the in-rush current control resistor slows the buildup of voltage in the DC bus capacitors. Once the voltage is at an adequate level, the in-rush current control resistor is taken out of the circuit to allow current to freely flow from the AC line into the DC bus capacitors. This is called on startup to avoid excessive current into the DC bus.

Returns:
None.

7.2.1.2 InRushRelayAdjust

Adjusts the in-rush control relay drive signal for operating from the crystal.

Prototype:
```
void
InRushRelayAdjust(void)
```

Description:
This function adjusts the drive signal to the in-rush control relay to achieve the desired drive frequency when operating the microcontroller from the crystal instead of from the PLL.

Returns:
None.

8 Main Application

8.1 Introduction

This is the main AC induction motor application code. It contains a state machine that controls the operation of the drive, an interrupt handler for the waveform update software interrupt, an interrupt handler for the millisecond frequency update software interrupt, and the main application startup code.

The waveform update interrupt handler is responsible for computing new values for the waveforms being driven to the inverter bridge. Based on the update rate, it will advance the drive angle and recompute new waveforms. The V/f tables is used to determine the amplitude and the appropriate modulation is performed. The new waveform values are passed to the PWM module to be supplied to the PWM hardware at the correct time.

The millisecond frequency update interrupt handler is responsible for handling the dynamic brake, computing the new drive frequency, and checking for fault conditions. If the drive is just starting, this is where the precharging of the high-side gate drivers is handled. If the drive has just stopped, this is where the DC injection braking is handled. Dynamic braking is handled by simply calling the update function for the dynamic braking module.

When running, a variety of things are done to adjust the drive frequency. First, the target frequency is adjusted by a PI controller if closed-loop mode is enabled, moving the target frequency such that the rotor will reach the desired frequency. Then, the acceleration or deceleration rate is applied as appropriate to move the output frequency towards the target frequency. In the case of deceleration, the deceleration rate may be reduced based on the DC bus voltage. The result of this frequency adjustment is a new step angle, which is subsequently used by the waveform update interrupt handler to generate the output waveforms.

The over-temperature, DC bus under-voltage, DC bus over-voltage, motor under-current, and motor over-current faults are all checked for by examining the readings from the ADC. Fault conditions are handled by turning off the drive output and indicating the appropriate fault, which must be cleared before the drive will run again.

The state machine that controls the operation of the drive is woven throughout the millisecond frequency update interrupt handler and the routines that start, stop, and adjust the parameters of the motor drive. Together, it ensures that the motor drive responds to commands and parameter changes in a logical and predictable manner.

The application startup code performs high-level initialization of the microcontroller (such as enabling peripherals) and calls the initialization routines for the various support modules. Since all the work within the motor drive occurs with interrupt handlers, its final task is to go into an infinite loop that puts the processor into sleep mode. This serves two purposes; it allows the processor to wait until there is work to be done (for example, an interrupt) before it executes any further code, and it allows the processor usage meter to gather the data it needs to determine processor usage.

The main application code is contained in `main.c`, with `main.h` containing the definitions for the defines, variables, and functions exported to the remainder of the application.

8.2 Definitions

Defines

- CRYSTAL_CLOCK
- FLASH_PB_END
- FLASH_PB_START
- STATE_BACK_PRECHARGE
- STATE_BACK_REV
- STATE_BACK_RUN
- STATE_BACK_STOPPING
- STATE_BRAKE
- STATE_FLAG_BACKWARD
- STATE_FLAG_BRAKE
- STATE_FLAG_FORWARD
- STATE_FLAG_PRECHARGE
- STATE_FLAG_REV
- STATE_FLAG_RUN
- STATE_FLAG_STOPPING
- STATE_PRECHARGE
- STATE_REV
- STATE_RUN
- STATE_STOPPED
- STATE_STOPPING
- SYSTEM_CLOCK

Functions

- int main (void)
- static void MainCheckFaults (void)
- void MainClearFaults (void)
- static void MainDCBrakeHandler (void)
- void MainEmergencyStop (void)
- unsigned long MainFrequencyController (void)
- static void MainFrequencyHandler (unsigned long ulTarget)
- unsigned long MainIsFaulted (void)
- unsigned long MainIsRunning (void)
- static long MainLongMul (long lX, long lY)
- void MainMillisecondTick (void)
- static void MainPrechargeHandler (void)
- void MainRun (void)
- void MainSetDirection (tBoolean bForward)
- void MainSetFault (unsigned long ulFaultFlag)
- void MainSetFrequency (void)
- void MainSetLoopMode (tBoolean bClosed)

- void MainSetPWMFrequency (void)
- void MainStop (void)
- void MainUpdateFAdjI (long lNewFAdjI)
- void MainWaveformTick (void)

Variables

- static long g_lFrequencyIntegrator
- static long g_lFrequencyIntegratorMax
- unsigned char g_ucMotorStatus
- static unsigned long g_ulAccelRate
- unsigned long g_ulAngle
- static unsigned long g_ulAngleDelta
- static unsigned long g_ulDecelRate
- unsigned long g_ulFaultFlags
- static unsigned long g_ulFrequency
- static unsigned long g_ulFrequencyFract
- static unsigned long g_ulFrequencyWhole
- static unsigned long g_ulState
- static unsigned long g_ulStateCount
- static unsigned long g_ulTargetFrequency

8.2.1 Define Documentation

8.2.1.1 CRYSTAL_CLOCK

Definition:

```
#define CRYSTAL_CLOCK
```

Description:

The frequency of the crystal attached to the microcontroller. This must match the crystal value passed to SysCtlClockSet() in main().

8.2.1.2 FLASH_PB_END

Definition:

```
#define FLASH_PB_END
```

Description:

The address of the last block of flash to be used for storing parameters. Since the end of flash is used for parameters, this is actually the first address past the end of flash.

8.2.1.3 FLASH_PB_START

Definition:

```
#define FLASH_PB_START
```

Description:

The address of the first block of flash to be used for storing parameters.

8.2.1.4 STATE_BACK_PRECHARGE

Definition:

```
#define STATE_BACK_PRECHARGE
```

Description:

The motor drive is precharging the bootstrap capacitors on the high side gate drivers while running in the backward direction. Once the capacitors are charged, the state machine will automatically transition to STATE_BACK_RUN.

8.2.1.5 STATE_BACK_REV

Definition:

```
#define STATE_BACK_REV
```

Description:

The motor drive is decelerating down to a stop while running in the backward direction, at which point the state machine will automatically transition to STATE_RUN. This results in a direction change of the motor drive.

8.2.1.6 STATE_BACK_RUN

Definition:

```
#define STATE_BACK_RUN
```

Description:

The motor drive is running in the backward direction, either at the target frequency or slewing to the target frequency.

8.2.1.7 STATE_BACK_STOPPING

Definition:

```
#define STATE_BACK_STOPPING
```

Description:

The motor drive is decelerating down to a stop while running in the backward direction, at which point the state machine will automatically transition to STATE_STOPPED. This results in the motor drive being stopped.

8.2.1.8 STATE_BRAKE

Definition:

```
#define STATE_BRAKE
```

Description:

The motor drive is performing DC injection braking. Once the braking has completed, the state machine will automatically transition to STATE_STOPPED.

8.2.1.9 STATE_FLAG_BACKWARD

Definition:

```
#define STATE_FLAG_BACKWARD
```

Description:

A state flag that indicates that the motor drive is in the backward direction.

8.2.1.10 STATE_FLAG_BRAKE

Definition:

```
#define STATE_FLAG_BRAKE
```

Description:

A state flag that indicates that the motor drive is performing DC injection braking.

8.2.1.11 STATE_FLAG_FORWARD

Definition:

```
#define STATE_FLAG_FORWARD
```

Description:

A state flag that indicates that the motor drive is in the forward direction.

8.2.1.12 STATE_FLAG_PRECHARGE

Definition:

```
#define STATE_FLAG_PRECHARGE
```

Description:

A state flag that indicates that the motor drive is precharging the bootstrap capacitors on the high side gate drivers.

8.2.1.13 STATE_FLAG_REV

Definition:

```
#define STATE_FLAG_REV
```

Description:

A state flag that indicates that the motor drive is reversing direction.

8.2.1.14 STATE_FLAG_RUN

Definition:

```
#define STATE_FLAG_RUN
```

Description:

A state flag that indicates that the motor drive is running.

8.2.1.15 STATE_FLAG_STOPPING

Definition:

```
#define STATE_FLAG_STOPPING
```

Description:

A state flag that indicates that the motor drive is stopping.

8.2.1.16 STATE_PRECHARGE

Definition:

```
#define STATE_PRECHARGE
```

Description:

The motor drive is precharging the bootstrap capacitors on the high side gate drivers. Once the capacitors are charged, the state machine will automatically transition to STATE_RUN.

8.2.1.17 STATE_REV

Definition:

```
#define STATE_REV
```

Description:

The motor drive is decelerating down to a stop, at which point the state machine will automatically transition to STATE_BACK_RUN. This results in a direction change of the motor drive.

8.2.1.18 STATE_RUN

Definition:

```
#define STATE_RUN
```

Description:

The motor drive is running, either at the target frequency or slewing to the target frequency.

8.2.1.19 STATE_STOPPED

Definition:

```
#define STATE_STOPPED
```

Description:

The motor drive is stopped. A run request will cause a transition to the STATE_PRECHARGE or STATE_BACK_PRECHARGE states, depending upon the direction flag.

8.2.1.20 STATE_STOPPING

Definition:

```
#define STATE_STOPPING
```

Description:

The motor drive is decelerating down to a stop, at which point the state machine will automatically transition to STATE_STOPPED. This results in the motor drive being stopped.

8.2.1.21 SYSTEM_CLOCK

Definition:

```
#define SYSTEM_CLOCK
```

Description:

The frequency of the processor clock, which is also the clock rate of all the peripherals. This must match the value configured by SysCtlClockSet() in main().

8.2.2 Function Documentation

8.2.2.1 main

Handles setup of the application on the AC induction motor drive.

Prototype:

```
int
main(void)
```

Description:
This is the main application entry point for the AC induction motor drive. It is responsible for basic system configuration, initialization of the various application drivers and peripherals, and the main application loop.

Returns:
Never returns.

8.2.2.2 MainCheckFaults [static]

Checks for motor drive faults.

Prototype:
```
static void
MainCheckFaults(void)
```

Description:
This function checks for fault conditions that may occur during the operation of the motor drive. The ambient temperature, DC bus voltage, and motor current are all monitored for fault conditions.

Returns:
None.

8.2.2.3 MainClearFaults

Clears the latched fault conditions.

Prototype:
```
void
MainClearFaults(void)
```

Description:
This function will clear the latched fault conditions and turn off the fault LED.

Returns:
None.

8.2.2.4 MainDCBrakeHandler [static]

Handles the DC braking mode of the motor drive.

Prototype:
```
static void
MainDCBrakeHandler(void)
```

Description:
This function performs the processing and state transitions associated with the DC braking mode of the motor drive.

Returns:
None.

8.2.2.5 MainEmergencyStop

Emergency stops the motor drive.

Prototype:

```
void
MainEmergencyStop(void)
```

Description:

This function performs an emergency stop of the motor drive. The outputs will be shut down immediately, the drive put into the stopped state with the frequency at zero, and the emergency stop fault condition will be asserted.

Returns:

None.

8.2.2.6 MainFrequencyController

Adjusts the motor drive frequency based on the rotor frequency.

Prototype:

```
unsigned long
MainFrequencyController(void)
```

Description:

This function uses a PI controller to adjust the motor drive frequency in order to get the rotor frequency to match the target frequency (meaning that the motor drive frequency will actually be above the target frequency).

Returns:

Returns the new motor drive target frequency.

8.2.2.7 MainFrequencyHandler [static]

Adjusts the motor drive frequency based on the target frequency.

Prototype:

```
static void
MainFrequencyHandler(unsigned long ulTarget)
```

Parameters:

ulTarget is the target frequency of the motor drive, specified as a 16.16 fixed-point value.

Description:

This function adjusts the motor drive frequency towards a given target frequency. Limitations such as acceleration and deceleration rate, along with precautions such as limiting the deceleration rate to control the DC bus voltage, are handled by this function.

Returns:

None.

8.2.2.8 MainIsFaulted

Determines if a latched fault condition exists.

Prototype:

```
unsigned long
MainIsFaulted(void)
```

Description:

This function determines if a fault condition has occurred but not been cleared.

Returns:

Returns 1 if there is an uncleared fault condition and 0 otherwise.

8.2.2.9 MainIsRunning

Determines if the motor drive is currently running.

Prototype:

```
unsigned long
MainIsRunning(void)
```

Description:

This function will determine if the motor drive is currently running. By this definition, running means not stopped; the motor drive is considered to be running even when it is precharging before starting the waveforms and DC injection braking after stopping the waveforms.

Returns:

Returns 0 if the motor drive is stopped and 1 if it is running.

8.2.2.10 MainLongMul [static]

Multiplies two 16.16 fixed-point numbers.

Prototype:

```
static long
MainLongMul(long lX,
            long lY)
```

Parameters:

lX is the first multiplicand.

lY is the second multiplicand.

Description:

This function takes two fixed-point numbers, in 16.16 format, and multiplies them together, returning the 16.16 fixed-point result. It is the responsibility of the caller to ensure that the dynamic range of the integer portion of the value is not exceeded; if it is exceeded the result will not be correct.

Returns:

Returns the result of the multiplication, in 16.16 fixed-point format.

8.2.2.11 MainMillisecondTick

Handles the millisecond frequency update software interrupt.

Prototype:

```
void
MainMillisecondTick(void)
```

Description:

This function is called as a result of the frequency update software interrupt being asserted. This interrupted is asserted every millisecond by the PWM interrupt handler.

The frequency of the motor drive will be updated, along with handling state changes of the drive (such as initiating braking when the motor drive has come to a stop).

Note:

Since this interrupt is software triggered, there is no interrupt source to clear in this handler.

Returns:

None.

8.2.2.12 MainPrechargeHandler [static]

Handles the gate driver precharge mode of the motor drive.

Prototype:

```
static void
MainPrechargeHandler(void)
```

Description:

This function performs the processing and state transitions associated with the gate driver precharge mode of the motor drive.

Returns:

None.

8.2.2.13 MainRun

Starts the motor drive.

Prototype:

```
void
MainRun(void)
```

Description:

This function starts the motor drive. If the motor is currently stopped, it will begin the process of starting the motor. If the motor is currently stopping, it will cancel the stop operation and return the motor to the target frequency.

Returns:

None.

8.2.2.14 MainSetDirection

Sets the direction of the motor drive.

Prototype:

```
void
MainSetDirection(tBoolean bForward)
```

Parameters:

bForward is a boolean that is true if the motor drive should be run in the forward direction.

Description:

This function changes the direction of the motor drive. If required, the state machine will be transitioned to a new state in order to change the direction of the motor drive.

Returns:

None.

8.2.2.15 MainSetFault

Indicate that a fault condition has been detected.

Prototype:

```
void
MainSetFault(unsigned long ulFaultFlag)
```

Parameters:

ulFaultFlag is a flag that indicates the fault condition that was detected.

Description:

This function is called when a fault condition is detected. It will update the fault flags to indicate the fault condition that was detected, and cause the fault LED to blink to indicate a fault.

Returns:

None.

8.2.2.16 MainSetFrequency

Changes the target frequency of the motor drive.

Prototype:

```
void
MainSetFrequency(void)
```

Description:

This function changes the target frequency of the motor drive. If required, the state machine will be transitioned to a new state in order to move the motor drive to the target frequency.

Returns:

None.

8.2.2.17 MainSetLoopMode

Sets the open-/closed-loop mode of the motor drive.

Prototype:

```
void
MainSetLoopMode(tBoolean bClosed)
```

Parameters:

bClosed is a boolean that is true if the motor drive should be run in closed-loop mode.

Description:

This function changes the open-/closed-loop mode of the motor drive. When enabling closed-loop mode, the integrator is initialized as if the current motor frequency was achieved in closed-loop mode; this provides a smoother transition into closed-loop mode.

Returns:

None.

8.2.2.18 MainSetPWMFrequency

Changes the PWM frequency of the motor drive.

Prototype:

```
void
MainSetPWMFrequency(void)
```

Description:

This function changes the period of the PWM signals produced by the motor drive. It is simply a wrapper function around the PWMSetFrequency() function; the PWM frequency-based timing parameters of the motor drive are adjusted as part of the PWM frequency update.

Returns:

None.

8.2.2.19 MainStop

Stops the motor drive.

Prototype:

```
void
MainStop(void)
```

Description:

This function stops the motor drive. If the motor is currently running, it will begin the process of stopping the motor.

Returns:

None.

8.2.2.20 MainUpdateFAdjI

Updates the I coefficient of the frequency PI controller.

Prototype:

```
void
MainUpdateFAdjI(long lNewFAdjI)
```

Parameters:

lNewFAdjI is the new value of the I coefficient.

Description:

This function updates the value of the I coefficient of the frequency PI controller. In addition to updating the I coefficient, it recomputes the maximum value of the integrator and the current value of the integrator in terms of the new I coefficient (eliminating any instantaneous jump in the output of the PI controller).

Returns:

None.

8.2.2.21 MainWaveformTick

Handles the waveform update software interrupt.

Prototype:

```
void
MainWaveformTick(void)
```

Description:

This function is periodically called as a result of the waveform update software interrupt being asserted. This interrupt is asserted at the requested rate (based on the update rate parameter) by the PWM interrupt handler.

The angle of the motor drive will be updated, and new waveform values computed and supplied to the PWM module.

Note:

Since this interrupt is software triggered, there is no interrupt source to clear in this handler.

Returns:

None.

8.2.3 Variable Documentation

8.2.3.1 g_lFrequencyIntegrator [static]

Definition:

```
static long g_lFrequencyIntegrator
```

Description:

The accumulator for the integral term of the PI controller for the motor drive frequency.

8.2.3.2 g_lFrequencyIntegratorMax [static]

Definition:

```
static long g_lFrequencyIntegratorMax
```

Description:

The maximum value that of the PI controller accumulator (g_lFrequencyIntegrator). This limit is based on the I coefficient and the maximum frequency of the motor drive, and is used to avoid "integrator windup", a potential pitfall of PI controllers.

8.2.3.3 g_ucMotorStatus

Definition:

```
unsigned char g_ucMotorStatus
```

Description:

The current operation state of the motor drive.

8.2.3.4 g_ulAccelRate [static]

Definition:

```
static unsigned long g_ulAccelRate
```

Description:

The current rate of acceleration. This will start as the parameter value, but may be reduced in order to manage increases in the motor current.

8.2.3.5 g_ulAngle

Definition:

```
unsigned long g_ulAngle
```

Description:

The current angle of the motor drive output, expressed as a 0.32 fixed-point value that is the percentage of the way around a circle.

8.2.3.6 g_ulAngleDelta [static]

Definition:

```
static unsigned long g_ulAngleDelta
```

Description:

The amount by which the motor drive angle is updated for a single PWM period, expressed as a 0.32 fixed-point value. For example, if the motor drive is being updated every fifth PWM period, this value should be multiplied by five to determine the amount to adjust the angle.

8.2.3.7 g_ulDecelRate [static]

Definition:

```
static unsigned long g_ulDecelRate
```

Description:

The current rate of deceleration. This will start as the parameter value, but may be reduced in order to manage increases in the DC bus voltage.

8.2.3.8 g_ulFaultFlags

Definition:

```
unsigned long g_ulFaultFlags
```

Description:

The latched fault status flags for the motor drive, enumerated by FAULT_EMERGENCY_STOP, FAULT_VBUS_LOW, FAULT_VBUS_HIGH, FAULT_CURRENT_LOW, FAULT_CURRENT_HIGH, FAULT_POWER_MODULE, and FAULT_TEMPERATURE_HIGH.

8.2.3.9 g_ulFrequency [static]

Definition:

```
static unsigned long g_ulFrequency
```

Description:

The current motor drive frequency, expressed as a 16.16 fixed-point value.

8.2.3.10 g_ulFrequencyFract [static]

Definition:

```
static unsigned long g_ulFrequencyFract
```

Description:

The fractional part of the current motor drive frequency. This value is expressed as the numerator of a fraction whose denominator is the PWM frequency. This is used in conjunction with g_ulFrequencyWhole to compute the value for g_ulFrequency.

8.2.3.11 g_ulFrequencyWhole [static]

Definition:

```
static unsigned long g_ulFrequencyWhole
```

Description:

The whole part of the current motor drive frequency. This is used in conjunction with g_ulFrequencyFract to compute the value for g_ulFrequency.

8.2.3.12 g_ulState [static]

Definition:

```
static unsigned long g_ulState
```

Description:

The current state of the motor drive state machine. This state machine controls acceleration, deceleration, starting, stopping, braking, and reversing direction of the motor drive.

8.2.3.13 g_ulStateCount [static]

Definition:

```
static unsigned long g_ulStateCount
```

Description:

A count of the number of milliseconds to remain in a particular state.

8.2.3.14 g_ulTargetFrequency [static]

Definition:

```
static unsigned long g_ulTargetFrequency
```

Description:

The target frequency for the motor drive, expressed as a 16.16 fixed-point value.

9 On-board User Interface

9.1 Introduction

The on-board user interface consists of a push button and a potentiometer. The push button triggers actions when pressed, released, and when held for a period of time. The potentiometer specifies the value of a parameter.

The push button is debounced using a vertical counter. A vertical counter is a method where each bit of the counter is stored in a different word, and multiple counters can be incremented simultaneously. They work really well for debouncing switches; up to 32 switches can be debounced at the same time. Although only one switch is used, the code is already capable of debouncing an additional 31 switches.

A callback function can be called when the switch is pressed, when it is released, and when it is held. If held, the press function will not be called for that button press.

The potentiometer input is passed through a low-pass filter and then a stable value detector. The low-pass filter reduces the noise introduced by the potentiometer and the ADC. Even the low-pass filter does not remove all the noise and does not produce an unchanging value when the potentiometer is not being turned. Therefore, a stable value detector is used to find when the potentiometer value is only changing slightly. When this occurs, the output value is held constant until the potentiometer value has changed significantly. Because of this, the parameter value that is adjusted by the potentiometer will not jitter around when the potentiometer is left alone.

The application is responsible for reading the value of the switch(es) and the potentiometer on a periodic basis. The routines provided here perform all the processing of those values.

The code for handling the on-board user interface elements is contained in `ui_onboard.c`, with `ui_onboard.h` containing the definitions for the structures and functions exported to the remainder of the application.

9.2 Definitions

Data Structures

- tUIOnboardSwitch

Functions

- void UIOnboardInit (unsigned long ulSwitches, unsigned long ulPotentiometer)
- unsigned long UIOnboardPotentiometerFilter (unsigned long ulValue)
- void UIOnboardSwitchDebouncer (unsigned long ulSwitches)

Variables

- static unsigned long g_ulUIOnboardClockA
- static unsigned long g_ulUIOnboardClockB
- static unsigned long g_ulUIOnboardFilteredPotValue
- static unsigned long g_ulUIOnboardPotCount
- static unsigned long g_ulUIOnboardPotMax
- static unsigned long g_ulUIOnboardPotMin
- static unsigned long g_ulUIOnboardPotSum
- static unsigned long g_ulUIOnboardPotValue
- static unsigned long g_ulUIOnboardSwitches

9.2.1 Data Structure Documentation

9.2.1.1 tUIOnboardSwitch

Definition:

```
typedef struct
{
    unsigned char ucBit;
    unsigned long ulHoldTime;
    void (*pfnPress)(void);
    void (*pfnRelease)(void);
    void (*pfnHold)(void);
}
tUIOnboardSwitch
```

Members:

ucBit The bit position of this switch.

ulHoldTime The number of sample periods which the switch must be held in order to invoke the hold function.

pfnPress A pointer to the function to be called when the switch is pressed. For switches that do not have a hold function, this is called as soon as the switch is pressed. For switches that have a hold function, it is called when the switch is released only if it was held for less than the hold time (if held longer, this function will not be called). If no press function is required then this can be NULL.

pfnRelease A pointer to the function to be called when the switch is released. if no release function is required then this can be NULL.

pfnHold A pointer to the function to be called when the switch is held for the hold time. If no hold function is required then this can be NULL.

Description:

This structure contains a set of variables that describe the properties of a switch.

9.2.2 Function Documentation

9.2.2.1 UIOnboardInit

Initializes the on-board user interface elements.

Prototype:

```
void
UIOnboardInit(unsigned long ulSwitches,
              unsigned long ulPotentiometer)
```

Parameters:

ulSwitches is the initial state of the switches.

ulPotentiometer is the initial state of the potentiometer.

Description:

This function initializes the internal state of the on-board user interface handlers. The initial state of the switches are used to avoid spurious switch presses/releases, and the initial state of the potentiometer is used to make the filtered potentiometer value track more accurately when first starting (after a short period of time it will track correctly regardless of the initial state).

Returns:

None.

9.2.2.2 UIOnboardPotentiometerFilter

Filters the value of a potentiometer.

Prototype:

```
unsigned long
UIOnboardPotentiometerFilter(unsigned long ulValue)
```

Parameters:

ulValue is the current sample for the potentiometer.

Description:

This function performs filtering on the sampled value of a potentiometer. First, a single pole IIR low pass filter is applied to the raw sampled value. Then, the filtered value is examined to determine when the potentiometer is being turned and when it is not. When the potentiometer is not being turned (and variations in the value are therefore the result of noise in the system), a constant value is returned instead of the filtered value. When the potentiometer is being turned, the filtered value is returned unmodified.

This second filtering step eliminates the flutter when the potentiometer is not being turned so that processes that are driven from its value (such as a motor position) do not result in the motor jiggling back and forth to the potentiometer flutter. The downside to this filtering is a larger turn of the potentiometer being required before the output value changes.

Returns:

Returns the filtered potentiometer value.

9.2.2.3 UIOnboardSwitchDebouncer

Debounces a set of switches.

Prototype:

```
void
UIOnboardSwitchDebouncer(unsigned long ulSwitches)
```

Parameters:

ulSwitches is the current state of the switches.

Description:

This function takes a set of switch inputs and performs software debouncing of their state. Changes in the debounced state of a switch are reflected back to the application via callback functions. For each switch, a press can be distinguished from a hold, allowing two functions to coexist on a single switch; a separate callback function is called for a hold as opposed to a press.

For best results, the switches should be sampled and passed to this function on a periodic basis. Randomness in the sampling time may result in degraded performance of the debouncing routine.

Returns:

None.

9.2.3 Variable Documentation

9.2.3.1 g_ulUIOnboardClockA [static]

Definition:

```
static unsigned long g_ulUIOnboardClockA
```

Description:

This is the low order bit of the clock used to count the number of samples with the switches in the non-debounced state.

9.2.3.2 g_ulUIOnboardClockB [static]

Definition:

```
static unsigned long g_ulUIOnboardClockB
```

Description:

This is the high order bit of the clock used to count the number of samples with the switches in the non-debounced state.

9.2.3.3 g_ulUIOnboardFilteredPotValue [static]

Definition:

```
static unsigned long g_ulUIOnboardFilteredPotValue
```

Description:

The detected stable value of the potentiometer. This will be 0xffff.ffff when the value of the potentiometer is changing and will be a value within the potentiometer range when the potentiometer value is stable.

9.2.3.4 g_ulUIOnboardPotCount [static]

Definition:

```
static unsigned long g_ulUIOnboardPotCount
```

Description:

The count of samples that have been collected into the accumulator (g_ulUIOnboardPotSum).

9.2.3.5 g_ulUIOnboardPotMax [static]

Definition:

```
static unsigned long g_ulUIOnboardPotMax
```

Description:

The maximum value of the potentiometer over a small period. This is used to detect a stable value of the potentiometer.

9.2.3.6 g_ulUIOnboardPotMin [static]

Definition:

```
static unsigned long g_ulUIOnboardPotMin
```

Description:

The minimum value of the potentiometer over a small period. This is used to detect a stable value of the potentiometer.

9.2.3.7 g_ulUIOnboardPotSum [static]

Definition:

```
static unsigned long g_ulUIOnboardPotSum
```

Description:

An accumulator of the low pass filtered potentiometer values for a small period. When a stable potentiometer value is detected, this is used to compute the average value (and therefore the stable value of the potentiometer).

9.2.3.8 g_ulUIOnboardPotValue [static]

Definition:

```
static unsigned long g_ulUIOnboardPotValue
```

Description:

The value of the potentiometer after being passed through the single pole IIR low pass filter.

9.2.3.9 g_ulUIOnboardSwitches [static]

Definition:

```
static unsigned long g_ulUIOnboardSwitches
```

Description:

The debounced state of the switches.

10 Pin Definitions

10.1 Introduction

The pins on the microcontroller are connected to a variety of external devices. The defines in this file provide a name more descriptive than "PB0" for the various devices connected to the microcontroller pins.

These defines are provided in `pins.h`.

10.2 Definitions

Defines

- PIN_BRAKE_PIN
- PIN_BRAKE_PORT
- PIN_CCP0_PIN
- PIN_CCP0_PORT
- PIN_CCP1_PIN
- PIN_CCP1_PORT
- PIN_ENCA_PIN
- PIN_ENCA_PORT
- PIN_ENCB_PIN
- PIN_ENCB_PORT
- PIN_FAULT_PIN
- PIN_FAULT_PORT
- PIN_I_PHASEU
- PIN_I_PHASEV
- PIN_I_PHASEW
- PIN_INDEX_PIN
- PIN_INDEX_PORT
- PIN_ISENSE_PIN
- PIN_ISENSE_PORT
- PIN_LEDFAULT_PIN
- PIN_LEDFAULT_PORT
- PIN_LEDRUN_PIN
- PIN_LEDRUN_PORT
- PIN_LEDSTATUS1_PIN
- PIN_LEDSTATUS1_PORT

- PIN_LEDSTATUS2_PIN
- PIN_LEDSTATUS2_PORT
- PIN_PHASEU_HIGH_PIN
- PIN_PHASEU_HIGH_PORT
- PIN_PHASEU_LOW_PIN
- PIN_PHASEU_LOW_PORT
- PIN_PHASEV_HIGH_PIN
- PIN_PHASEV_HIGH_PORT
- PIN_PHASEV_LOW_PIN
- PIN_PHASEV_LOW_PORT
- PIN_PHASEW_HIGH_PIN
- PIN_PHASEW_HIGH_PORT
- PIN_PHASEW_LOW_PIN
- PIN_PHASEW_LOW_PORT
- PIN_POTENTIOMETER_PIN
- PIN_POTENTIOMETER_PORT
- PIN_SWITCH_PIN
- PIN_SWITCH_PIN_BIT
- PIN_SWITCH_PORT
- PIN_UART0RX_PIN
- PIN_UART0RX_PORT
- PIN_UART0TX_PIN
- PIN_UART0TX_PORT
- PIN_VSENSE

10.2.1 Define Documentation

10.2.1.1 PIN_BRAKE_PIN

Definition:

```
#define PIN_BRAKE_PIN
```

Description:

The GPIO pin on which the brake pin resides.

10.2.1.2 PIN_BRAKE_PORT

Definition:

```
#define PIN_BRAKE_PORT
```

Description:

The GPIO port on which the brake pin resides.

10.2.1.3 PIN_CCP0_PIN

Definition:

```
#define PIN_CCP0_PIN
```

Description:

The GPIO pin on which the CCP0 pin resides.

10.2.1.4 PIN_CCP0_PORT

Definition:

```
#define PIN_CCP0_PORT
```

Description:

The GPIO port on which the CCP0 pin resides.

10.2.1.5 PIN_CCP1_PIN

Definition:

```
#define PIN_CCP1_PIN
```

Description:

The GPIO pin on which the CCP1 pin resides.

10.2.1.6 PIN_CCP1_PORT

Definition:

```
#define PIN_CCP1_PORT
```

Description:

The GPIO port on which the CCP1 pin resides.

10.2.1.7 PIN_ENCA_PIN

Definition:

```
#define PIN_ENCA_PIN
```

Description:

The GPIO pin on which the quadrature encoder channel A pin resides.

10.2.1.8 PIN_ENCA_PORT

Definition:

```
#define PIN_ENCA_PORT
```

Description:

The GPIO port on which the quadrature encoder channel A pin resides.

10.2.1.9 PIN_ENCB_PIN

Definition:

```
#define PIN_ENCB_PIN
```

Description:

The GPIO pin on which the quadrature encoder channel B pin resides.

10.2.1.10 PIN_ENCB_PORT

Definition:

```
#define PIN_ENCB_PORT
```

Description:

The GPIO port on which the quadrature encoder channel B pin resides.

10.2.1.11 PIN_FAULT_PIN

Definition:

```
#define PIN_FAULT_PIN
```

Description:

The GPIO pin on which the PWM fault pin resides.

10.2.1.12 PIN_FAULT_PORT

Definition:

```
#define PIN_FAULT_PORT
```

Description:

The GPIO port on which the PWM fault pin resides.

10.2.1.13 PIN_I_PHASEU

Definition:

```
#define PIN_I_PHASEU
```

Description:

The ADC channel on which the phase U current sense resides.

10.2.1.14 PIN_I_PHASEV

Definition:

```
#define PIN_I_PHASEV
```

Description:

The ADC channel on which the phase V current sense resides.

10.2.1.15 PIN_I_PHASEW

Definition:

```
#define PIN_I_PHASEW
```

Description:

The ADC channel on which the phase W current sense resides.

10.2.1.16 PIN_INDEX_PIN

Definition:

```
#define PIN_INDEX_PIN
```

Description:

The GPIO pin on which the quadrature encoder index pin resides.

10.2.1.17 PIN_INDEX_PORT

Definition:

```
#define PIN_INDEX_PORT
```

Description:

The GPIO port on which the quadrature encoder index pin resides.

10.2.1.18 PIN_ISENSE_PIN

Definition:

```
#define PIN_ISENSE_PIN
```

Description:

The GPIO pin on which the overall current sense pin resides.

10.2.1.19 PIN_ISENSE_PORT

Definition:

```
#define PIN_ISENSE_PORT
```

Description:

The GPIO port on which the overall current sense pin resides.

10.2.1.20 PIN_LEDFAULT_PIN

Definition:

```
#define PIN_LEDFAULT_PIN
```

Description:

The GPIO pin on which the fault LED resides.

10.2.1.21 PIN_LEDFAULT_PORT

Definition:

```
#define PIN_LEDFAULT_PORT
```

Description:

The GPIO port on which the fault LED resides.

10.2.1.22 PIN_LEDRUN_PIN

Definition:

```
#define PIN_LEDRUN_PIN
```

Description:

The GPIO pin on which the run LED resides.

10.2.1.23 PIN_LEDRUN_PORT

Definition:

```
#define PIN_LEDRUN_PORT
```

Description:

The GPIO port on which the run LED resides.

10.2.1.24 PIN_LEDSTATUS1_PIN

Definition:

```
#define PIN_LEDSTATUS1_PIN
```

Description:

The GPIO pin on which the status one LED resides.

10.2.1.25 PIN_LEDSTATUS1_PORT

Definition:

```
#define PIN_LEDSTATUS1_PORT
```

Description:

The GPIO port on which the status one LED resides.

10.2.1.26 PIN_LEDSTATUS2_PIN

Definition:

```
#define PIN_LEDSTATUS2_PIN
```

Description:

The GPIO pin on which the status two LED resides.

10.2.1.27 PIN_LEDSTATUS2_PORT

Definition:

```
#define PIN_LEDSTATUS2_PORT
```

Description:

The GPIO port on which the status two LED resides.

10.2.1.28 PIN_PHASEU_HIGH_PIN

Definition:

```
#define PIN_PHASEU_HIGH_PIN
```

Description:

The GPIO pin on which the phase U high side pin resides.

10.2.1.29 PIN_PHASEU_HIGH_PORT

Definition:

```
#define PIN_PHASEU_HIGH_PORT
```

Description:

The GPIO port on which the phase U high side pin resides.

10.2.1.30 PIN_PHASEU_LOW_PIN

Definition:

```
#define PIN_PHASEU_LOW_PIN
```

Description:

The GPIO pin on which the phase U low side pin resides.

10.2.1.31 PIN_PHASEU_LOW_PORT

Definition:

```
#define PIN_PHASEU_LOW_PORT
```

Description:

The GPIO port on which the phase U low side pin resides.

10.2.1.32 PIN_PHASEV_HIGH_PIN

Definition:

```
#define PIN_PHASEV_HIGH_PIN
```

Description:

The GPIO pin on which the phase V high side pin resides.

10.2.1.33 PIN_PHASEV_HIGH_PORT

Definition:

```
#define PIN_PHASEV_HIGH_PORT
```

Description:

The GPIO port on which the phase V high side pin resides.

10.2.1.34 PIN_PHASEV_LOW_PIN

Definition:

```
#define PIN_PHASEV_LOW_PIN
```

Description:

The GPIO pin on which the phase V low side pin resides.

10.2.1.35 PIN_PHASEV_LOW_PORT

Definition:

```
#define PIN_PHASEV_LOW_PORT
```

Description:

The GPIO port on which the phase V low side pin resides.

10.2.1.36 PIN_PHASEW_HIGH_PIN

Definition:

```
#define PIN_PHASEW_HIGH_PIN
```

Description:

The GPIO pin on which the phase W high side pin resides.

10.2.1.37 PIN_PHASEW_HIGH_PORT

Definition:

```
#define PIN_PHASEW_HIGH_PORT
```

Description:

The GPIO port on which the phase W high side pin resides.

10.2.1.38 PIN_PHASEW_LOW_PIN

Definition:

```
#define PIN_PHASEW_LOW_PIN
```

Description:

The GPIO pin on which the phase W low side pin resides.

10.2.1.39 PIN_PHASEW_LOW_PORT

Definition:

```
#define PIN_PHASEW_LOW_PORT
```

Description:

The GPIO port on which the phase W low side pin resides.

10.2.1.40 PIN_POTENTIOMETER_PIN

Definition:

```
#define PIN_POTENTIOMETER_PIN
```

Description:

The GPIO pin on which the potentiometer oscillator resides.

10.2.1.41 PIN_POTENTIOMETER_PORT

Definition:

```
#define PIN_POTENTIOMETER_PORT
```

Description:

The GPIO port on which the potentiometer oscillator resides.

10.2.1.42 PIN_SWITCH_PIN

Definition:

```
#define PIN_SWITCH_PIN
```

Description:

The GPIO pin on which the user push button resides.

10.2.1.43 PIN_SWITCH_PIN_BIT

Definition:

```
#define PIN_SWITCH_PIN_BIT
```

Description:

The bit lane of the GPIO pin on which the user push button resides.

10.2.1.44 PIN_SWITCH_PORT

Definition:

```
#define PIN_SWITCH_PORT
```

Description:

The GPIO port on which the user push button resides.

10.2.1.45 PIN_UART0RX_PIN

Definition:

```
#define PIN_UART0RX_PIN
```

Description:

The GPIO pin on which the UART0 Rx pin resides.

10.2.1.46 PIN_UART0RX_PORT

Definition:

```
#define PIN_UART0RX_PORT
```

Description:

The GPIO port on which the UART0 Rx pin resides.

10.2.1.47 PIN_UART0TX_PIN

Definition:

```
#define PIN_UART0TX_PIN
```

Description:

The GPIO pin on which the UART0 Tx pin resides.

10.2.1.48 PIN_UART0TX_PORT

Definition:

```
#define PIN_UART0TX_PORT
```

Description:

The GPIO port on which the UART0 Tx pin resides.

10.2.1.49 PIN_VSENSE

Definition:

```
#define PIN_VSENSE
```

Description:

The ADC channel on which the DC bus voltage sense resides.

11 PWM Control

11.1 Introduction

The generated motor drive waveforms are driven to the inverter bridge with the PWM module. The PWM generators are run in a fully synchronous manner; the counters are synchronized (that is, the values of the three counters are always the same) and updates to the duty cycle registers are synchronized to the zero value of the PWM counters.

The dead-band unit in each PWM generator is used to prevent shoot-through current in the inverter bridge when switching between the high side to the low of a phase. Shoot-through occurs because the turn-on time of one gate doesn't always match the turn-off time of the other, so both may be on for a short period despite the fact that only one of their inputs is on. By providing a period of time where both inputs are off when making the transition, shoot-through is not possible.

The PWM outputs can be in one of four modes during the operation of the motor drive. The first is off, where all six outputs are in the inactive state. This is the state used when the motor drive is stopped; the motor is electrically disconnected during this time (effectively the same as disconnecting the cable) and is free to spin as if it were unplugged.

The next mode is precharge, where the three outputs to the high side switches are inactive and the three outputs to the low side switches are switches at a 50% duty cycle. The high side gate drivers have a bootstrap circuit for generating the voltage to drive the gates that only charges when the low side is switching; this precharge mode allows the bootstrap circuit to generate the required gate drive voltage before real waveforms are driven. Failure to precharge the high side gate drivers would simply result in distortion of the first part of the output waveform (until the bootstrap circuit generates a voltage high enough to turn on the high side gate). This mode is used briefly when going from a non-driving state to a driving state.

The next mode is running, where all six outputs are actively toggling. This will create a magnetic field in the stator of the motor, inducing a magnetic field in the rotor and causing it to spin. This mode is used to drive the motor.

The final mode is DC injection braking, where the first PWM pair are actively toggling, the low side of the second PWM pair is always on, and the third PWM pair is inactive. This results in a fixed DC voltage being applied across the motor, resulting in braking. This mode is optionally used briefly when going from a driving state to a non-driving state in order to completely stop the rotation of the rotor. For loads with high inertia, or low friction rotors, this can reduce the rotor stop time from minutes to seconds. Applying a DC voltage to an AC induction motor does generate a lot of heat in the windings, so it should only be used for as long as required to stop the rotor and no longer.

The PWM outputs are configured to immediately switch to the inactive state when the processor is stopped by a debugger. This prevents the current PWM state from becoming a DC voltage (since the processor is no longer running to change the duty cycles) and damaging the motor. In general, though, it is not a good idea to stop the processor when the motor is running. When no longer driven, the motor will start to slow down due to friction; when the processor is restarted, it will continue driving at the previous drive frequency. The difference between rotor and stator frequency (that is, the slip) has become much greater due to the time that the motor was not being driven.

This will likely result in an immediate motor over-current fault since the increased slip will result in a rise motor current. While not harmful, it does not allow the typically desired behavior of being able to stop the application, look at the internal state, and then restart the application as if nothing had happened.

An interrupt is generated at each zero value of the counter in PWM generator zero; this is used as a time base for the generation of waveforms as well as a time to queue the next duty cycle update into the hardware. At any given time, the PWM module is outputting the duty cycle for period N, has the duty cycle for period N+1 queued in its holding registers waiting for the next zero value of the counter, and the microcontroller is computing the duty cycle for period N+2.

Two “software” interrupts are generated by the PWM interrupt handler. One is used to update the waveform; this occurs at a configurable rate of every X PWM period. The other is used to update the drive frequency and perform other periodic system tasks such as fault monitoring; this occurs every millisecond. The unused interrupts from the second and third PWM generator are used for these “software” interrupts; the ability to fake the assertion of an interrupt through the NVIC software interrupt trigger register is used to generate these “software” interrupts.

The code for handling the PWM module is contained in `pwm_ctrl.c`, with `pwm_ctrl.h` containing the definitions for the variables and functions exported to the remainder of the application.

11.2 Definitions

Defines

- PWM_CLOCK
- PWM_CLOCK_WIDTH
- PWM_FLAG_NEW_DUTY_CYCLE
- PWM_FLAG_NEW_FREQUENCY

Functions

- void GPIOBIntHandler (void)
- void PWM0IntHandler (void)
- void PWMFaultHandler (void)
- unsigned long PWMGetPeriodCount (void)
- void PWMInit (void)
- void PWMOutputDCBrake (unsigned long ulVoltage)
- void PWMOutputOff (void)
- void PWMOutputOn (void)
- void PWMOutputPrecharge (void)
- void PWMReducePeriodCount (unsigned long ulCount)
- void PWMSetDeadBand (void)
- void PWMSetDutyCycle (unsigned long ulDutyCycleU, unsigned long ulDutyCycleV, unsigned long ulDutyCycleW)
- void PWMSetFrequency (void)
- void PWMSetMinPulseWidth (void)
- void PWMSetUpdateRate (unsigned char ucUpdateRate)
- static void PWMUpdateDutyCycle (void)

Variables

- static unsigned long g_ulMinPulseWidth
- static unsigned long g_ulPWMClock
- static unsigned long g_ulPWMDutyCycleU
- static unsigned long g_ulPWMDutyCycleV
- static unsigned long g_ulPWMDutyCycleW
- static unsigned long g_ulPWMFlags
- unsigned long g_ulPWMFrequency
- static unsigned long g_ulPWMMillisecondCount
- static unsigned long g_ulPWMPeriodCount

11.2.1 Define Documentation

11.2.1.1 PWM_CLOCK

Definition:

```
#define PWM_CLOCK
```

Description:

The frequency of the clock that drives the PWM generators.

11.2.1.2 PWM_CLOCK_WIDTH

Definition:

```
#define PWM_CLOCK_WIDTH
```

Description:

The width of a single PWM clock, in nanoseconds.

11.2.1.3 PWM_FLAG_NEW_DUTY_CYCLE

Definition:

```
#define PWM_FLAG_NEW_DUTY_CYCLE
```

Description:

The bit number of the flag in g_ulPWMFlags that indicates that a new duty cycle (that is, compare) is ready to be supplied to the PWM module.

11.2.1.4 PWM_FLAG_NEW_FREQUENCY

Definition:

```
#define PWM_FLAG_NEW_FREQUENCY
```

Description:

The bit number of the flag in g_ulPWMFlags that indicates that a new PWM frequency (that is, period) is ready to be supplied to the PWM module.

11.2.2 Function Documentation

11.2.2.1 GPIOBIntHandler

Handles the PWM fault interrupt.

Prototype:

```
void
GPIOBIntHandler(void)
```

Description:

This function is called as a result of the interrupt generated by the assertion of the PWM fault input. It is treated as a sticky fault condition and will emergency stop the motor drive.

Returns:

None.

11.2.2.2 PWM0IntHandler

Handles the PWM interrupt.

Prototype:

```
void
PWM0IntHandler(void)
```

Description:

This function is called as a result of the interrupt generated by the PWM module when the counter reaches zero. If an updated PWM frequency or duty cycle is available, they will be updated in the hardware by this function.

Returns:

None.

11.2.2.3 PWMFaultHandler

Handles the PWM fault interrupt.

Prototype:

```
void
PWMFaultHandler(void)
```

Description:

This function is called as a result of the interrupt generated by the assertion of the PWM fault input. It is treated as a sticky fault condition and will emergency stop the motor drive.

Returns:

None.

11.2.2.4 PWMGetPeriodCount

Gets the number of PWM interrupts that have occurred.

Prototype:

```
unsigned long
PWMGetPeriodCount(void)
```

Description:

This function returns the number of PWM interrupts that have been counted. Used in conjunction with the desired update rate, missed waveform updates can be detected and compensated for.

Returns:

The number of PWM interrupts that have been counted.

11.2.2.5 PWMInit

Initializes the PWM control routines.

Prototype:

```
void
PWMInit(void)
```

Description:

This function initializes the PWM module and the control routines, preparing them to produce PWM waveforms to drive the power module.

Returns:

None.

11.2.2.6 PWMOutputDCBrake

Sets the PWM outputs to DC injection brake the motor.

Prototype:

```
void
PWMOutputDCBrake(unsigned long ulVoltage)
```

Parameters:

ulVoltage is the DC voltage to be applied to the motor. This value must be less than 160V, half the nominal DC bus voltage (and likely much less than that).

Description:

This function configures the PWM outputs such that they will provide DC injection braking of the motor.

Note:

Once the motor comes to a complete stop, DC injection braking will simply generate heat within the motor, likely causing damage. It is important that the DC injection braking be disabled to avoid this situation.

Returns:
None.

11.2.2.7 PWMOutputOff

Turns off all the PWM outputs.

Prototype:
```
void
PWMOutputOff(void)
```

Description:
This function turns off all of the PWM outputs, preventing them from being propagates to the gate drivers.

Returns:
None.

11.2.2.8 PWMOutputOn

Turns on all the PWM outputs.

Prototype:
```
void
PWMOutputOn(void)
```

Description:
This function turns on all of the PWM outputs, allowing them to be propagated to the gate drivers.

Returns:
None.

11.2.2.9 PWMOutputPrecharge

Sets the PWM outputs to precharge the high side gate drives.

Prototype:
```
void
PWMOutputPrecharge(void)
```

Description:
This function configures the PWM outputs such that they will start charging the bootstrap capacitor on the high side gate drives. Without this step, the high side gates will not turn on properly for the first several PWM cycles when starting the motor drive.

Returns:
None.

11.2.2.10 PWMReducePeriodCount

Reduces the count of PWM interrupts.

Prototype:

```
void
PWMReducePeriodCount(unsigned long ulCount)
```

Parameters:

ulCount is the number by which to reduce the PWM interrupt count.

Description:

This function reduces the PWM interrupt count by a given number. When the waveform values are updated, the interrupt count can be reduced by the appropriate amount to maintain a proper indication of when the next waveform update should occur.

If the PWM interrupt count is not reduced when the waveforms are recomputed, the waveform update software interrupt will not be triggered as desired.

Returns:

None.

11.2.2.11 PWMSetDeadBand

Configures the dead timers for the PWM generators.

Prototype:

```
void
PWMSetDeadBand(void)
```

Description:

This function configures the dead timers for all three PWM generators based on the dead time parameter.

Returns:

None.

11.2.2.12 PWMSetDutyCycle

Sets the duty cycle of the generated PWM waveforms.

Prototype:

```
void
PWMSetDutyCycle(unsigned long ulDutyCycleU,
                unsigned long ulDutyCycleV,
                unsigned long ulDutyCycleW)
```

Parameters:

ulDutyCycleU is the duty cycle of the waveform for the U phase of the motor, specified as a 16.16 fixed point value between 0.0 and 1.0.

ulDutyCycleV is the duty cycle of the waveform for the V phase of the motor, specified as a 16.16 fixed point value between 0.0 and 1.0.

ulDutyCycleW is the duty cycle of the waveform for the W phase of the motor, specified as a 16.16 fixed point value between 0.0 and 1.0.

Description:

This function configures the duty cycle of the generated PWM waveforms. The duty cycle update will not occur immediately; the change will be registered for synchronous application to the output waveforms to avoid discontinuities.

Returns:

None.

11.2.2.13 PWMSetFrequency

Sets the frequency of the generated PWM waveforms.

Prototype:

```
void
PWMSetFrequency(void)
```

Description:

This function configures the frequency of the generated PWM waveforms. The frequency update will not occur immediately; the change will be registered for synchronous application to the output waveforms to avoid discontinuities.

Returns:

None.

11.2.2.14 PWMSetMinPulseWidth

Computes the minimum PWM pulse width.

Prototype:

```
void
PWMSetMinPulseWidth(void)
```

Description:

This function computes the minimum PWM pulse width based on the minimum pulse width parameter and the dead time parameter. The dead timers will reduce the width of a PWM pulse, so their value must be considered to avoid pulses shorter than the parameter value being produced.

Returns:

None.

11.2.2.15 PWMSetUpdateRate

Changes the update rate of the motor drive.

Prototype:

```
void
PWMSetUpdateRate(unsigned char ucUpdateRate)
```

Parameters:

ucUpdateRate is the number of PWM periods between updates.

Description:

This function changes the rate at which the motor drive waveforms are recomputed. Lower update values recompute the waveforms more frequently, providing more accurate waveforms at the cost of increased processor usage.

Returns:

None.

11.2.2.16 PWMUpdateDutyCycle [static]

Updates the duty cycle in the PWM module.

Prototype:

```
static void
PWMUpdateDutyCycle(void)
```

Description:

This function programs the duty cycle of the PWM waveforms into the PWM module. The changes will be written to the hardware and the hardware instructed to start using the new values the next time its counters reach zero.

Returns:

None.

11.2.3 Variable Documentation

11.2.3.1 g_ulMinPulseWidth [static]

Definition:

```
static unsigned long g_ulMinPulseWidth
```

Description:

The minimum width of an output PWM pulse, in PWM clocks.

11.2.3.2 g_ulPWMClock [static]

Definition:

```
static unsigned long g_ulPWMClock
```

Description:
The number of PWM clocks in a single PWM period.

11.2.3.3 g_ulPWMDutyCycleU [static]

Definition:
```
static unsigned long g_ulPWMDutyCycleU
```

Description:
The duty cycle of the waveform output to the U phase of the bridge.

11.2.3.4 g_ulPWMDutyCycleV [static]

Definition:
```
static unsigned long g_ulPWMDutyCycleV
```

Description:
The duty cycle of the waveform output to the V phase of the bridge.

11.2.3.5 g_ulPWMDutyCycleW [static]

Definition:
```
static unsigned long g_ulPWMDutyCycleW
```

Description:
The duty cycle of the waveform output to the W phase of the bridge.

11.2.3.6 g_ulPWMFlags [static]

Definition:
```
static unsigned long g_ulPWMFlags
```

Description:
A set of flags that control the operation of the PWM control routines. The flags are PWM_FLAG_NEW_FREQUENCY, and PWM_FLAG_NEW_DUTY_CYCLE.

11.2.3.7 g_ulPWMFrequency

Definition:
```
unsigned long g_ulPWMFrequency
```

Description:
The frequency of the output PWM waveforms.

11.2.3.8 g_ulPWMMillisecondCount [static]

Definition:

```
static unsigned long g_ulPWMMillisecondCount
```

Description:

A counter that is used to determine when a millisecond has passed. The millisecond software interrupt is triggered based on this count.

11.2.3.9 g_ulPWMPeriodCount [static]

Definition:

```
static unsigned long g_ulPWMPeriodCount
```

Description:

A count of the number of PWM periods have occurred, based on the number of PWM module interrupts. This is incremented when a PWM interrupt is handled and decremented by the waveform generation handler.

12 Serial Interface

12.1 Introduction

A generic, packet-based serial protocol is utilized for communicating with the motor drive board. This provides a method to control the motor drive, adjust its parameters, and retrieve real-time performance data. The serial interface is run at 115,200 baud, with an 8-N-1 data format. Some of the factors that influenced the design of this protocol include:

- The same serial protocol should be used for all motor drive boards, regardless of the motor type (that is, AC induction, stepper, and so on).
- The protocol should make reasonable attempts to protect against invalid commands being acted upon.
- It should be possible to connect to a running motor drive board and lock on to the real-time data stream without having to restart the data stream.

The code for handling the serial protocol is contained in `ui_serial.c`, with `ui_serial.h` containing the definitions for the structures, functions, and variables exported to the remainder of the application. The file `commands.h` contains the definitions for the commands, parameters, real-time data items, and responses that are used in the serial protocol.

12.1.1 Command Message Format

Commands are sent to the motor drive with the following format:

```
{tag} {length} {command} {optional command data byte(s)} {checksum}
```

- The {tag} byte is 0xff.
- The {length} byte contains the overall length of the command packet, starting with the {tag} and ending with the {checksum}. The maximum packet length is 255 bytes.
- The {command} byte is the command being sent. Based on the command, there may be optional command data bytes that follow.
- The {checksum} byte is the value such that the sum of all bytes in the command packet (including the checksum) will be zero. This is used to validate a command packet and allow the target to synchronize with the command stream being sent by the host.

For example, the 0x01 command with no data bytes would be sent as follows:

```
0xff 0x04 0x01 0xfc
```

And the 0x02 command with two data bytes (0xab and 0xcd) would be sent as follows:

```
0xff 0x06 0x02 0xab 0xcd 0x81
```

12.1.2 Status Message Format

Status messages are sent from the motor drive with the following format:

```
{tag} {length} {data bytes} {checksum}
```

- The {tag} byte is 0xfe for command responses and 0xfd for real-time data.
- The {length} byte contains the overall length of the status packet, starting with the {tag} byte and ending with the {checksum}.
- The contents of the data bytes are dependent upon the tag byte.
- The {checksum} is the value such that the sum of all bytes in the status packet (including the checksum) will be zero. This is used to validate a status packet and allow the user interface to synchronize with the status stream being sent by the target.

For command responses ({tag} = 0xfe), the first data byte is the command that is being responded to. The remaining bytes are the response, and are dependent upon the command.

For real-time data messages ({tag} = 0xfd), each real-time data item is transmitted as a little-endian value (for example, for a 16-bit value, the lower 8 bits first then the upper 8 bits). The data items are in the same order as returned by the data item list (CMD_GET_DATA_ITEMS) regardless of the order that they were enabled.

For example, if data items 1, 5, and 17 were enabled, and each was two bytes in length, there would be 6 data bytes in the packet:

```
0xfd 0x09 {d1[0:7]} {d1[8:15]} {d5[0:7]} {d5[8:15]} {d17[0:7]}
     {d17[8:15]} {checksum}
```

12.1.3 Parameter Interpretation

The size and units of the parameters are dependent upon the motor drive; the units are not conveyed in the serial protocol. Each parameter value is transmitted in little endian format. Not all parameters are necessarily supported by a motor drive, only those that are appropriate.

12.1.4 Interface To The Application

The serial protocol handler takes care of all the serial communications and command interpretation. A set of functions provided by the application and an array of structures that describe the parameters and real-time data items supported by the motor drive. The functions are used when an application-specific action needs to take place as a result of the serial communication (such as starting the motor drive). The structures are used to handle the parameters and real-time data items of the motor drive.

12.2 Definitions

Defines

- CMD_DISABLE_DATA_ITEM
- CMD_DISCOVER_TARGET
- CMD_EMERGENCY_STOP
- CMD_ENABLE_DATA_ITEM
- CMD_GET_DATA_ITEMS
- CMD_GET_PARAM_DESC
- CMD_GET_PARAM_VALUE
- CMD_GET_PARAMS
- CMD_ID_TARGET
- CMD_LOAD_PARAMS
- CMD_RUN
- CMD_SAVE_PARAMS
- CMD_SET_PARAM_VALUE
- CMD_START_DATA_STREAM
- CMD_STOP
- CMD_STOP_DATA_STREAM
- CMD_UPGRADE
- DATA_ANALOG_INPUT
- DATA_BUS_VOLTAGE
- DATA_DEBUG_INFO
- DATA_DIRECTION
- DATA_FAULT_STATUS
- DATA_MOTOR_CURRENT
- DATA_MOTOR_POSITION
- DATA_MOTOR_POWER
- DATA_MOTOR_STATUS
- DATA_NUM_ITEMS
- DATA_PHASE_A_CURRENT
- DATA_PHASE_B_CURRENT
- DATA_PHASE_C_CURRENT
- DATA_PROCESSOR_USAGE
- DATA_ROTOR_SPEED
- DATA_STATOR_SPEED
- DATA_TEMPERATURE
- MOTOR_STATUS_ACCEL
- MOTOR_STATUS_DECEL
- MOTOR_STATUS_RUN
- MOTOR_STATUS_STOP
- PARAM_ACCEL
- PARAM_ACCEL_CURRENT
- PARAM_ACCEL_POWER

- PARAM_BEMF_SKIP_COUNT
- PARAM_BLANK_OFF
- PARAM_BRAKE_COOL_TIME
- PARAM_BRAKE_OFF_VOLTAGE
- PARAM_BRAKE_ON_VOLTAGE
- PARAM_CAN_RX_COUNT
- PARAM_CAN_TX_COUNT
- PARAM_CLOSED_LOOP
- PARAM_CONTROL_MODE
- PARAM_CURRENT_POS
- PARAM_CURRENT_POWER
- PARAM_CURRENT_SPEED
- PARAM_DATA_RATE
- PARAM_DC_BRAKE_TIME
- PARAM_DC_BRAKE_V
- PARAM_DECAY_MODE
- PARAM_DECEL
- PARAM_DECEL_POWER
- PARAM_DECEL_VOLTAGE
- PARAM_DIRECTION
- PARAM_ENCODER_PRESENT
- PARAM_ETH_RX_COUNT
- PARAM_ETH_TCP_TIMEOUT
- PARAM_ETH_TX_COUNT
- PARAM_FAULT_STATUS
- PARAM_FIRMWARE_VERSION
- PARAM_FIXED_ON_TIME
- PARAM_GPIO_DATA
- PARAM_HOLDING_CURRENT
- PARAM_MAX_BRAKE_TIME
- PARAM_MAX_BUS_VOLTAGE
- PARAM_MAX_CURRENT
- PARAM_MAX_POWER
- PARAM_MAX_SPEED
- PARAM_MAX_TEMPERATURE
- PARAM_MIN_BUS_VOLTAGE
- PARAM_MIN_CURRENT
- PARAM_MIN_POWER
- PARAM_MIN_SPEED
- PARAM_MODULATION
- PARAM_MOTOR_STATUS
- PARAM_MOTOR_TYPE
- PARAM_NUM_LINES
- PARAM_NUM_POLES
- PARAM_POWER_I
- PARAM_POWER_P

- PARAM_PRECHARGE_TIME
- PARAM_PWM_DEAD_TIME
- PARAM_PWM_FREQUENCY
- PARAM_PWM_MIN_PULSE
- PARAM_PWM_UPDATE
- PARAM_RESISTANCE
- PARAM_SENSOR_POLARITY
- PARAM_SENSOR_PRESENT
- PARAM_SENSOR_TYPE
- PARAM_SPEED_I
- PARAM_SPEED_P
- PARAM_STARTUP_COUNT
- PARAM_STARTUP_DUTY
- PARAM_STARTUP_ENDSP
- PARAM_STARTUP_ENDV
- PARAM_STARTUP_RAMP
- PARAM_STARTUP_STARTSP
- PARAM_STARTUP_STARTV
- PARAM_STARTUP_THRESH
- PARAM_STEP_MODE
- PARAM_TARGET_CURRENT
- PARAM_TARGET_POS
- PARAM_TARGET_POWER
- PARAM_TARGET_SPEED
- PARAM_USE_BUS_COMP
- PARAM_USE_DC_BRAKE
- PARAM_USE_DYNAM_BRAKE
- PARAM_USE_ONBOARD_UI
- PARAM_VF_RANGE
- PARAM_VF_TABLE
- RESP_ID_TARGET_ACIM
- RESP_ID_TARGET_BLDC
- RESP_ID_TARGET_STEPPER
- TAG_CMD
- TAG_DATA
- TAG_STATUS
- UISERIAL_MAX_RECV
- UISERIAL_MAX_XMIT

Functions

- void UART0IntHandler (void)
- static unsigned long UISerialFindParameter (unsigned char ucID)
- void UISerialInit (void)
- static void UISerialRangeCheck (unsigned long ulIdx)
- static void UISerialScanReceive (void)
- void UISerialSendRealTimeData (void)
- static tBoolean UISerialTransmit (unsigned char ∗pucBuffer)

Variables

- static tBoolean g_bEnableRealTimeData
- static unsigned char g_pucUISerialData[UISERIAL_MAX_XMIT]
- static unsigned char g_pucUISerialReceive[UISERIAL_MAX_RECV]
- static unsigned char g_pucUISerialResponse[UISERIAL_MAX_XMIT]
- static unsigned char g_pucUISerialTransmit[UISERIAL_MAX_XMIT]
- static unsigned long g_pulUIRealTimeData[(DATA_NUM_ITEMS+31)/32]
- static unsigned long g_ulUISerialReceiveRead
- static unsigned long g_ulUISerialReceiveWrite
- static unsigned long g_ulUISerialTransmitRead
- static unsigned long g_ulUISerialTransmitWrite

12.2.1 Define Documentation

12.2.1.1 CMD_DISABLE_DATA_ITEM

Definition:

```
#define CMD_DISABLE_DATA_ITEM
```

Description:

Removes a real-time data item from the real-time data output stream. To avoid a change in the real-time data output stream at an unexpected time, this command should only be issued when the real-time data output stream is disabled.

Command:

```
TAG_CMD 0x05 CMD_DISABLE_DATA_ITEM {item} {checksum}
```

- `{item}` is the real-time data item to be removed from the real-time data output stream; must be one of the **DATA_xxx** values.

Response:

```
TAG_STATUS 0x04 CMD_DISABLE_DATA_ITEM {checksum}
```

12.2.1.2 CMD_DISCOVER_TARGET

Definition:

```
#define CMD_DISCOVER_TARGET
```

Description:

This command is used to discover the motor drive board(s) that may be connected to the networked communication channel (e.g. CAN, Ethernet). This command is similar to the CMD_ID_TARGET command, but intended for networked operation. Additional parameters are available in the response that will allow the networked device to provide board-specific information (e.g. configuration switch settings) that can be used to identify which board is to be selected for operation.

Command:

```
TAG_CMD 0x04 CMD_DISCOVER_TARGET {checksum}
```

Response:

```
TAG_STATUS 0x0A CMD_DISCOVER_TARGET {type} {id} {remote-ip} {checksum}
```

- `{type}` identifies the motor drive type; will be one of RESP_ID_TARGET_BLDC, RESP_ID_TARGET_STEPPER, or RESP_ID_TARGET_ACIM.
- `{id}` is a board-specific identification value; will typically be the setting read from a set of configuration switches on the board.
- `{config}` is used to provide additional (if needed) board configuration information. The interpretation of this field will vary with the board type.

12.2.1.3 CMD_EMERGENCY_STOP

Definition:

```
#define CMD_EMERGENCY_STOP
```

Description:

Stops the motor, if it is not already stopped. This may take more aggressive action than CMD_STOP at the cost of precision. For example, for a stepper motor, the stop command would ramp the speed down before stopping the motor while emergency stop would stop stepping immediately; in the later case, it is possible that the motor will spin a couple of additional steps, so position accuracy is sacrificed. This is needed for safety reasons.

Command:

```
TAG_CMD 0x04 CMD_EMERGENCY_STOP {checksum}
```

Response:

```
TAG_STATUS 0x04 CMD_EMERGENCY_STOP {checksum}
```

12.2.1.4 CMD_ENABLE_DATA_ITEM

Definition:

```
#define CMD_ENABLE_DATA_ITEM
```

Description:

Adds a real-time data item to the real-time data output stream. To avoid a change in the real-time data output stream at an unexpected time, this command should only be issued when the real-time data output stream is disabled.

Command:

```
TAG_CMD 0x05 CMD_ENABLE_DATA_ITEM {item} {checksum}
```

- `{item}` is the real-time data item to be added to the real-time data output stream; must be one of the **DATA_xxx** values.

Response:

```
TAG_STATUS 0x04 CMD_ENABLE_DATA_ITEM {checksum}
```

12.2.1.5 CMD_GET_DATA_ITEMS

Definition:

```
#define CMD_GET_DATA_ITEMS
```

Description:

Gets a list of the real-time data items supported by this motor drive. This command returns a list of real-time data item numbers, in no particular order, along with the size of the data item; each data item will be one of the **DATA_xxx** values.

Command:

```
TAG_CMD 0x04 CMD_GET_DATA_ITEMS {checksum}
```

Response:

```
TAG_STATUS {length} CMD_GET_DATA_ITEMS {item} {size}
    [{item} {size} ...] {checksum}
```

- `{item}` is a list of one or more **DATA_xxx** values.
- `{size}` is the size of the data item immediately preceding.

12.2.1.6 CMD_GET_PARAM_DESC

Definition:

```
#define CMD_GET_PARAM_DESC
```

Description:

Gets the description of a parameter. The size of the parameter value, the minimum and maximum values for the parameter, and the step between valid values for the parameter. If the minimum, maximum, and step values don't make sense for a parameter, they may be omitted from the response, leaving only the size.

Command:

```
TAG_CMD 0x05 CMD_GET_PARAM_DESC {param} {checksum}
```

- `{param}` is one of the **PARAM_xxx** values.

Response:

```
TAG_STATUS {length} CMD_GET_PARAM_DESC {size} {min} [{min} ...]
    {max} [{max} ...] {step} [{step} ...] {checksum}
```

- `{size}` is the size of the parameter in bytes.
- `{min}` is the minimum valid value for this parameter. The number of bytes for this value is determined by the size of the parameter.
- `{max}` is the maximum valid value for this parameter. The number of bytes for this value is determined by the size of the parameter.
- `{step}` is the increment between valid values for this parameter. It should be the case that "min + (step $*$ N) = max" for some positive integer N. The number of bytes for this value is determined by the size of the parameter.

12.2.1.7 CMD_GET_PARAM_VALUE

Definition:

```
#define CMD_GET_PARAM_VALUE
```

Description:

Gets the value of a parameter.

Command:

```
TAG_CMD 0x05 CMD_GET_PARAM_VALUE {param} {checksum}
```

- `{param}` is the parameter whose value should be returned; must be one of the parameters returned by CMD_GET_PARAMS.

Response:

```
TAG_STATUS {length} CMD_GET_PARAM_VALUE {value} [{value} ...]
    {checksum}
```

- `{value}` is the current value of the parameter. All bytes of the value will always be returned.

12.2.1.8 CMD_GET_PARAMS

Definition:

```
#define CMD_GET_PARAMS
```

Description:

Gets a list of the parameters supported by this motor drive. This command returns a list of parameter numbers, in no particular order; each will be one of the **PARAM_xxx** values.

Command:

```
TAG_CMD 0x04 CMD_GET_PARAMS {checksum}
```

Response:

```
TAG_STATUS {length} CMD_GET_PARAMS {param} [{param} ...] {checksum}
```

- `{param}` is a list of one or more **PARAM_xxx** values.

12.2.1.9 CMD_ID_TARGET

Definition:

```
#define CMD_ID_TARGET
```

Description:

This command is used to determine the type of motor driven by the board. In this context, the type of motor is a broad statement; for example, both single-phase and three-phase AC induction motors can be driven by a single AC induction motor board (not simultaneously, of course).

Command:

```
TAG_CMD 0x04 CMD_ID_TARGET {checksum}
```

Response:

```
TAG_STATUS 0x05 CMD_ID_TARGET {type} {checksum}
```

- `{type}` identifies the motor drive type; will be one of RESP_ID_TARGET_BLDC, RESP_ID_TARGET_STEPPER, or RESP_ID_TARGET_ACIM.

12.2.1.10 CMD_LOAD_PARAMS

Definition:

```
#define CMD_LOAD_PARAMS
```

Description:

Loads the most recent parameter set from flash, causing the current parameter values to be lost. This can be used to recover from parameter changes that do not work very well. For example, if a set of parameter changes are made during experimentation and they turn out to cause the motor to perform poorly, this will restore the last-saved parameter set (which is presumably, but not necessarily, of better quality).

Command:

```
TAG_CMD 0x04 CMD_LOAD_PARAMS {checksum}
```

Response:

```
TAG_STATUS 0x04 CMD_LOAD_PARAMS {checksum}
```

12.2.1.11 CMD_RUN

Definition:

```
#define CMD_RUN
```

Description:

Starts the motor running based on the current parameter set, if it is not already running.

Command:

```
TAG_CMD 0x04 CMD_RUN {checksum}
```

Response:

```
TAG_STATUS 0x04 CMD_RUN {checksum}
```

12.2.1.12 CMD_SAVE_PARAMS

Definition:

```
#define CMD_SAVE_PARAMS
```

Description:

Saves the current parameter set to flash. Only the most recently saved parameter set is available for use, and it contains the default settings of all the parameters at power-up.

Command:

```
TAG_CMD 0x04 CMD_SAVE_PARAMS {checksum}
```

Response:

```
TAG_STATUS 0x04 CMD_SAVE_PARAMS {checksum}
```

12.2.1.13 CMD_SET_PARAM_VALUE

Definition:

```
#define CMD_SET_PARAM_VALUE
```

Description:

Sets the value of a parameter. For parameters that have values larger than a single byte, not all bytes of the parameter value need to be supplied; value bytes that are not supplied (that is, the more significant bytes) are treated as if a zero was transmitted. If more bytes than required for the parameter value are supplied, the extra bytes are ignored.

Command:

```
TAG_CMD {length} CMD_SET_PARAM_VALUE {param} {value} [{value} ...]
    {checksum}
```

- `{param}` is the parameter whose value should be set; must be one of the parameters returned by CMD_GET_PARAMS.
- `{value}` is the new value for the parameter.

Response:

```
TAG_STATUS 0x04 CMD_SET_PARAM_VALUE {checksum}
```

12.2.1.14 CMD_START_DATA_STREAM

Definition:

```
#define CMD_START_DATA_STREAM
```

Description:

Starts the real-time data output stream. Only those values that have been added to the output stream will be provided, and it will continue to run (regardless of any other motor drive state) until stopped.

Command:

```
TAG_CMD 0x04 CMD_START_DATA_STREAM {checksum}
```

Response:

```
TAG_STATUS 0x04 CMD_START_DATA_STREAM {checksum}
```

12.2.1.15 CMD_STOP

Definition:

```
#define CMD_STOP
```

Description:

Stops the motor, if it is not already stopped.

Command:

```
TAG_CMD 0x04 CMD_STOP {checksum}
```

Response:

```
TAG_STATUS 0x04 CMD_STOP {checksum}
```

12.2.1.16 CMD_STOP_DATA_STREAM

Definition:

```
#define CMD_STOP_DATA_STREAM
```

Description:

Stops the real-time data output stream. The output stream should be stopped before real-time data items are added to or removed from the stream to avoid unexpected changes in the stream data (it will all be valid data, there is simply no easy way to know what real-time data items are in a TAG_DATA packet if changes are made while the output stream is running).

Command:

```
TAG_CMD 0x04 CMD_STOP_DATA_STREAM {checksum}
```

Response:

```
TAG_STATUS 0x04 CMD_STOP_DATA_STREAM {checksum}
```

12.2.1.17 CMD_UPGRADE

Definition:

```
#define CMD_UPGRADE
```

Description:

Starts an upgrade of the firmware on the target. There is no response to this command; once received, the target will return to the control of the Stellaris boot loader and its serial protocol.

Command:

```
TAG_CMD 0x04 CMD_UPGRADE {checksum}
```

Response:

```
<none>
```

12.2.1.18 DATA_ANALOG_INPUT

Definition:

```
#define DATA_ANALOG_INPUT
```

Description:

This real-time data item provides the ambient temperature of the microcontroller.

12.2.1.19 DATA_BUS_VOLTAGE

Definition:

```
#define DATA_BUS_VOLTAGE
```

Description:

This real-time data item provides the bus voltage.

12.2.1.20 DATA_DEBUG_INFO

Definition:

```
#define DATA_DEBUG_INFO
```

Description:

This real-time data item provides application-specific debug information. The format of this data will vary from one application to the next. It is the responsibility of the user to ensure that the motor drive board and host application are in sync with the data format.

12.2.1.21 DATA_DIRECTION

Definition:

```
#define DATA_DIRECTION
```

Description:

This real-time data item provides the direction the motor drive is running.

12.2.1.22 DATA_FAULT_STATUS

Definition:

```
#define DATA_FAULT_STATUS
```

Description:

This real-time data item provides the current fault status of the motor drive.

12.2.1.23 DATA_MOTOR_CURRENT

Definition:

```
#define DATA_MOTOR_CURRENT
```

Description:

This real-time data item provides the current through the motor (that is, the sum of the phases).

12.2.1.24 DATA_MOTOR_POSITION

Definition:

```
#define DATA_MOTOR_POSITION
```

Description:

This real-time data item provides the position of the motor.

12.2.1.25 DATA_MOTOR_POWER

Definition:

```
#define DATA_MOTOR_POWER
```

Description:

This real-time data item provides the power supplied to the motor.

12.2.1.26 DATA_MOTOR_STATUS

Definition:

```
#define DATA_MOTOR_STATUS
```

Description:

This real-time data item provides the current operating mode of the motor drive. This value will be one of MOTOR_STATUS_STOP, MOTOR_STATUS_RUN, MOTOR_STATUS_ACCEL, or MOTOR_STATUS_DECEL.

12.2.1.27 DATA_NUM_ITEMS

Definition:

```
#define DATA_NUM_ITEMS
```

Description:

The number of real-time data items.

12.2.1.28 DATA_PHASE_A_CURRENT

Definition:

```
#define DATA_PHASE_A_CURRENT
```

Description:

This real-time data item provides the current through phase A of the motor.

12.2.1.29 DATA_PHASE_B_CURRENT

Definition:

```
#define DATA_PHASE_B_CURRENT
```

Description:

This real-time data item provides the current through phase B of the motor.

12.2.1.30 DATA_PHASE_C_CURRENT

Definition:

```
#define DATA_PHASE_C_CURRENT
```

Description:

This real-time data item provides the current through phase C of the motor.

12.2.1.31 DATA_PROCESSOR_USAGE

Definition:

```
#define DATA_PROCESSOR_USAGE
```

Description:

This real-time data item provides the percentage of the processor that is being utilized.

12.2.1.32 DATA_ROTOR_SPEED

Definition:

```
#define DATA_ROTOR_SPEED
```

Description:

This real-time data item provides the speed of the rotor (in other words, the motor shaft). For asynchronous motors, this will differ from the stator speed.

12.2.1.33 DATA_STATOR_SPEED

Definition:

```
#define DATA_STATOR_SPEED
```

Description:
This real-time data item provides the speed of the motor drive. This will only be available for asynchronous motors, where the rotor speed does not match the stator speed.

12.2.1.34 DATA_TEMPERATURE

Definition:
```
#define DATA_TEMPERATURE
```

Description:
This real-time data item provides the ambient temperature of the microcontroller.

12.2.1.35 MOTOR_STATUS_ACCEL

Definition:
```
#define MOTOR_STATUS_ACCEL
```

Description:
This is the motor status when the motor drive is accelerating.

12.2.1.36 MOTOR_STATUS_DECEL

Definition:
```
#define MOTOR_STATUS_DECEL
```

Description:
This is the motor status when the motor drive is decelerating.

12.2.1.37 MOTOR_STATUS_RUN

Definition:
```
#define MOTOR_STATUS_RUN
```

Description:
This is the motor status when the motor drive is running at a fixed speed.

12.2.1.38 MOTOR_STATUS_STOP

Definition:
```
#define MOTOR_STATUS_STOP
```

Description:
This is the motor status when the motor drive is stopped.

12.2.1.39 PARAM_ACCEL

Definition:

```
#define PARAM_ACCEL
```

Description:

Specifies the rate at which the speed of the motor is changed when increasing its speed.

12.2.1.40 PARAM_ACCEL_CURRENT

Definition:

```
#define PARAM_ACCEL_CURRENT
```

Description:

Specifies the motor current at which the acceleration of the motor drive is reduced in order to control increases in the motor current.

12.2.1.41 PARAM_ACCEL_POWER

Definition:

```
#define PARAM_ACCEL_POWER
```

Description:

Specifies the rate at which the power of the motor is changed when increasing its power.

12.2.1.42 PARAM_BEMF_SKIP_COUNT

Definition:

```
#define PARAM_BEMF_SKIP_COUNT
```

Description:

Contains the skip count for BEMF zero crossing detect hold-off.

12.2.1.43 PARAM_BLANK_OFF

Definition:

```
#define PARAM_BLANK_OFF
```

Description:

Specifies the blanking time after the current is removed.

12.2.1.44 PARAM_BRAKE_COOL_TIME

Definition:

```
#define PARAM_BRAKE_COOL_TIME
```

Description:

Specifies the time at which the dynamic braking leaves cooling mode if entered.

12.2.1.45 PARAM_BRAKE_OFF_VOLTAGE

Definition:

```
#define PARAM_BRAKE_OFF_VOLTAGE
```

Description:

Specifies the bus voltage at which the brake circuit is disengaged. If the brake circuit is engaged and the bus voltage drops below this value, then the brake circuit is disengaged.

12.2.1.46 PARAM_BRAKE_ON_VOLTAGE

Definition:

```
#define PARAM_BRAKE_ON_VOLTAGE
```

Description:

Specifies the bus voltage at which the brake circuit is first applied. If the bus voltage goes above this value, then the brake circuit is engaged.

12.2.1.47 PARAM_CAN_RX_COUNT

Definition:

```
#define PARAM_CAN_RX_COUNT
```

Description:

Indicates the number of CAN messages that have been received on the CAN bus.

12.2.1.48 PARAM_CAN_TX_COUNT

Definition:

```
#define PARAM_CAN_TX_COUNT
```

Description:

Indicates the number of CAN messages that have been transmitted on the CAN bus.

12.2.1.49 PARAM_CLOSED_LOOP

Definition:

```
#define PARAM_CLOSED_LOOP
```

Description:

Selects between open-loop and closed-loop mode of the motor drive.

12.2.1.50 PARAM_CONTROL_MODE

Definition:

```
#define PARAM_CONTROL_MODE
```

Description:

Specifies the motor control mode.

12.2.1.51 PARAM_CURRENT_POS

Definition:

```
#define PARAM_CURRENT_POS
```

Description:

Contains the current position of the motor. This is a read-only value and matches the corresponding real-time data item.

12.2.1.52 PARAM_CURRENT_POWER

Definition:

```
#define PARAM_CURRENT_POWER
```

Description:

Contains the current power of the motor. This is a read-only value and matches the corresponding real-time data item.

12.2.1.53 PARAM_CURRENT_SPEED

Definition:

```
#define PARAM_CURRENT_SPEED
```

Description:

Contains the current speed of the motor. This is a read-only value and matches the corresponding real-time data item.

12.2.1.54 PARAM_DATA_RATE

Definition:

```
#define PARAM_DATA_RATE
```

Description:

Specifies the rate at which the real-time data is provided by the motor drive.

12.2.1.55 PARAM_DC_BRAKE_TIME

Definition:

```
#define PARAM_DC_BRAKE_TIME
```

Description:

Specifies the amount of time to apply DC injection braking.

12.2.1.56 PARAM_DC_BRAKE_V

Definition:

```
#define PARAM_DC_BRAKE_V
```

Description:

Specifies the voltage to be applied during DC injection braking.

12.2.1.57 PARAM_DECAY_MODE

Definition:

```
#define PARAM_DECAY_MODE
```

Description:

Specifies the motor winding current decay mode.

12.2.1.58 PARAM_DECEL

Definition:

```
#define PARAM_DECEL
```

Description:

Specifies the rate at which the speed of the motor is changed when decreasing its speed.

12.2.1.59 PARAM_DECEL_POWER

Definition:

```
#define PARAM_DECEL_POWER
```

Description:

Specifies the rate at which the power of the motor is changed when decreasing its power.

12.2.1.60 PARAM_DECEL_VOLTAGE

Definition:

```
#define PARAM_DECEL_VOLTAGE
```

Description:

Specifies the bus voltage at which the deceleration of the motor drive is reduced in order to control increases in the bus voltage.

12.2.1.61 PARAM_DIRECTION

Definition:

```
#define PARAM_DIRECTION
```

Description:

Specifies the direction of rotation for the motor.

12.2.1.62 PARAM_ENCODER_PRESENT

Definition:

```
#define PARAM_ENCODER_PRESENT
```

Description:

Indicates whether or not an encoder feedback is present on the motor. Things that require the encoder feedback in order to operate (for example, closed-loop speed control) will be automatically disabled when there is no encoder feedback present.

12.2.1.63 PARAM_ETH_RX_COUNT

Definition:

```
#define PARAM_ETH_RX_COUNT
```

Description:

Indicates the number of Ethernet messages that have been received on the Ethernet interface.

12.2.1.64 PARAM_ETH_TCP_TIMEOUT

Definition:

```
#define PARAM_ETH_TCP_TIMEOUT
```

Description:

The timeout for an IDLE TCP connection.

12.2.1.65 PARAM_ETH_TX_COUNT

Definition:

```
#define PARAM_ETH_TX_COUNT
```

Description:

Indicates the number of Ethernet messages that have been transmitted on the Ethernet interface.

12.2.1.66 PARAM_FAULT_STATUS

Definition:

```
#define PARAM_FAULT_STATUS
```

Description:

Provides the fault status of the motor drive. This value matches the corresponding real-time data item; writing it will clear all latched fault status.

12.2.1.67 PARAM_FIRMWARE_VERSION

Definition:

```
#define PARAM_FIRMWARE_VERSION
```

Description:

Specifies the version of the firmware on the motor drive.

12.2.1.68 PARAM_FIXED_ON_TIME

Definition:

```
#define PARAM_FIXED_ON_TIME
```

Description:

Specifies the fixed on duration for application of motor winding current.

12.2.1.69 PARAM_GPIO_DATA

Definition:

```
#define PARAM_GPIO_DATA
```

Description:

Indicates the value(s) of the various GPIO signals on the motor drive board.

12.2.1.70 PARAM_HOLDING_CURRENT

Definition:

```
#define PARAM_HOLDING_CURRENT
```

Description:

Specifies the motor winding holding current.

12.2.1.71 PARAM_MAX_BRAKE_TIME

Definition:

```
#define PARAM_MAX_BRAKE_TIME
```

Description:
Specifies the maximum time that dynamic braking can be performed (in order to prevent circuit or motor damage).

12.2.1.72 PARAM_MAX_BUS_VOLTAGE

Definition:
```
#define PARAM_MAX_BUS_VOLTAGE
```

Description:
Specifies the maximum bus voltage when the motor is operating. If the bus voltage goes above this value, then an overvoltage alarm is asserted.

12.2.1.73 PARAM_MAX_CURRENT

Definition:
```
#define PARAM_MAX_CURRENT
```

Description:
Specifies the maximum current supplied to the motor when operating. If the current goes above this value, then an overcurrent alarm is asserted.

12.2.1.74 PARAM_MAX_POWER

Definition:
```
#define PARAM_MAX_POWER
```

Description:
Specifies the maximum power at which the motor can be run.

12.2.1.75 PARAM_MAX_SPEED

Definition:
```
#define PARAM_MAX_SPEED
```

Description:
Specifies the maximum speed at which the motor can be run.

12.2.1.76 PARAM_MAX_TEMPERATURE

Definition:
```
#define PARAM_MAX_TEMPERATURE
```

Description:
Specifies the maximum ambient temperature of the microcontroller. If the ambient temperature goes above this value, then an overtemperature alarm is asserted.

12.2.1.77 PARAM_MIN_BUS_VOLTAGE

Definition:

```
#define PARAM_MIN_BUS_VOLTAGE
```

Description:

Specifies the minimum bus voltage when the motor is operating. If the bus voltage drops below this value, then an undervoltage alarm is asserted.

12.2.1.78 PARAM_MIN_CURRENT

Definition:

```
#define PARAM_MIN_CURRENT
```

Description:

Specifies the minimum current supplied to the motor when operating. If the current drops below this value, then an undercurrent alarm is asserted.

12.2.1.79 PARAM_MIN_POWER

Definition:

```
#define PARAM_MIN_POWER
```

Description:

Specifies the minimum power at which the motor can be run.

12.2.1.80 PARAM_MIN_SPEED

Definition:

```
#define PARAM_MIN_SPEED
```

Description:

Specifies the minimum speed at which the motor can be run.

12.2.1.81 PARAM_MODULATION

Definition:

```
#define PARAM_MODULATION
```

Description:

Specifies the type of waveform modulation to be used to drive the motor.

12.2.1.82 PARAM_MOTOR_STATUS

Definition:

```
#define PARAM_MOTOR_STATUS
```

Description:

Provides the status of the motor drive, indicating the operating mode of the drive. This value will be one of MOTOR_STATUS_STOP, MOTOR_STATUS_RUN, MOTOR_STATUS_ACCEL, or MOTOR_STATUS_DECEL.

12.2.1.83 PARAM_MOTOR_TYPE

Definition:

```
#define PARAM_MOTOR_TYPE
```

Description:

Specifies the wiring configuration of the motor. For example, for an AC induction motor, this could be one phase or three phase; for a stepper motor, this could be unipolar or bipolar.

12.2.1.84 PARAM_NUM_LINES

Definition:

```
#define PARAM_NUM_LINES
```

Description:

Specifies the number of lines in the (optional) optical encoder attached to the motor.

12.2.1.85 PARAM_NUM_POLES

Definition:

```
#define PARAM_NUM_POLES
```

Description:

Specifies the number of pole pairs in the motor.

12.2.1.86 PARAM_POWER_I

Definition:

```
#define PARAM_POWER_I
```

Description:

Specifies the I coefficient for the PI controller used to adjust the motor power to track to the requested power.

12.2.1.87 PARAM_POWER_P

Definition:

```
#define PARAM_POWER_P
```

Description:

Specifies the P coefficient for the PI controller used to adjust the motor power to track to the requested power.

12.2.1.88 PARAM_PRECHARGE_TIME

Definition:

```
#define PARAM_PRECHARGE_TIME
```

Description:

Specifies the amount of time to precharge the bridge before starting the motor drive.

12.2.1.89 PARAM_PWM_DEAD_TIME

Definition:

```
#define PARAM_PWM_DEAD_TIME
```

Description:

Specifies the dead time between the high- and low-side PWM signals for a motor phase when using complimentary PWM outputs.

12.2.1.90 PARAM_PWM_FREQUENCY

Definition:

```
#define PARAM_PWM_FREQUENCY
```

Description:

Specifies the base PWM frequency used to generate the motor drive waveforms.

12.2.1.91 PARAM_PWM_MIN_PULSE

Definition:

```
#define PARAM_PWM_MIN_PULSE
```

Description:

Specifies the minimum width of a PWM pulse; pulses shorter than this value (either positive or negative) are removed from the output. A high pulse shorter than this value will result in the PWM signal remaining low, and a low pulse shorter than this value will result in the PWM signal remaining high.

12.2.1.92 PARAM_PWM_UPDATE

Definition:

```
#define PARAM_PWM_UPDATE
```

Description:

Specifies the rate at which the PWM duty cycle is updated.

12.2.1.93 PARAM_RESISTANCE

Definition:

```
#define PARAM_RESISTANCE
```

Description:

Specifies the winding resistance.

12.2.1.94 PARAM_SENSOR_POLARITY

Definition:

```
#define PARAM_SENSOR_POLARITY
```

Description:

Indicates the polarity of the GPIO/Digital Hall Sensor Inputs.

12.2.1.95 PARAM_SENSOR_PRESENT

Definition:

```
#define PARAM_SENSOR_PRESENT
```

Description:

Indicates whether or not Hall Effect sensor feedback is present on the motor. Things that require the sensor feedback in order to operate (for example, closed-loop speed control) will be automatically disabled when there is no sensor feedback present.

12.2.1.96 PARAM_SENSOR_TYPE

Definition:

```
#define PARAM_SENSOR_TYPE
```

Description:

Indicates the type of Hall Effect sensor feedback that is present on the motor. The Hall Effect sensor can be the Digital/GPIO type that is typically used, or can be the Analog/Linear type.

12.2.1.97 PARAM_SPEED_I

Definition:

```
#define PARAM_SPEED_I
```

Description:

Specifies the I coefficient for the PI controller used to adjust the motor speed to track to the requested speed.

12.2.1.98 PARAM_SPEED_P

Definition:

```
#define PARAM_SPEED_P
```

Description:

Specifies the P coefficient for the PI controller used to adjust the motor speed to track to the requested speed.

12.2.1.99 PARAM_STARTUP_COUNT

Definition:

```
#define PARAM_STARTUP_COUNT
```

Description:

Indicates the startup count for sensorless operation.

12.2.1.100 PARAM_STARTUP_DUTY

Definition:

```
#define PARAM_STARTUP_DUTY
```

Description:

Indicates the duty cycle for startup phase.

12.2.1.101 PARAM_STARTUP_ENDSP

Definition:

```
#define PARAM_STARTUP_ENDSP
```

Description:

Contains the ending speed for sensorless startup operation.

12.2.1.102 PARAM_STARTUP_ENDV

Definition:

```
#define PARAM_STARTUP_ENDV
```

Description:

Contains the ending voltage for sensorless startup operation.

12.2.1.103 PARAM_STARTUP_RAMP

Definition:

```
#define PARAM_STARTUP_RAMP
```

Description:

Contains the length of time for the open-loop sensorless startup.

12.2.1.104 PARAM_STARTUP_STARTSP

Definition:

```
#define PARAM_STARTUP_STARTSP
```

Description:

Contains the starting speed for sensorless startup operation.

12.2.1.105 PARAM_STARTUP_STARTV

Definition:

```
#define PARAM_STARTUP_STARTV
```

Description:

Contains the starting voltage for sensorless startup operation.

12.2.1.106 PARAM_STARTUP_THRESH

Definition:

```
#define PARAM_STARTUP_THRESH
```

Description:

Contains the back EMF threshhold voltage for sensorless startup.

12.2.1.107 PARAM_STEP_MODE

Definition:

```
#define PARAM_STEP_MODE
```

Description:

Specifies the motor stepping mode.

12.2.1.108 PARAM_TARGET_CURRENT

Definition:

```
#define PARAM_TARGET_CURRENT
```

Description:

Specifies the target running current of the motor.

12.2.1.109 PARAM_TARGET_POS

Definition:

```
#define PARAM_TARGET_POS
```

Description:

Specifies the target position of the motor.

12.2.1.110 PARAM_TARGET_POWER

Definition:

```
#define PARAM_TARGET_POWER
```

Description:

Specifies the target power supplied to the motor when operating.

12.2.1.111 PARAM_TARGET_SPEED

Definition:

```
#define PARAM_TARGET_SPEED
```

Description:

Specifies the desired speed of the the motor.

12.2.1.112 PARAM_USE_BUS_COMP

Definition:

```
#define PARAM_USE_BUS_COMP
```

Description:

Specifies whether DC bus voltage compensation should be performed.

12.2.1.113 PARAM_USE_DC_BRAKE

Definition:

```
#define PARAM_USE_DC_BRAKE
```

Description:

Specifies whether DC injection braking should be performed.

12.2.1.114 PARAM_USE_DYNAM_BRAKE

Definition:

```
#define PARAM_USE_DYNAM_BRAKE
```

Description:

Specifies whether dynamic braking should be performed.

12.2.1.115 PARAM_USE_ONBOARD_UI

Definition:

```
#define PARAM_USE_ONBOARD_UI
```

Description:

Specifies whether the on-board user interface should be active or inactive.

12.2.1.116 PARAM_VF_RANGE

Definition:

```
#define PARAM_VF_RANGE
```

Description:

Specifies the range of the V/f table.

12.2.1.117 PARAM_VF_TABLE

Definition:

```
#define PARAM_VF_TABLE
```

Description:

Specifies the mapping of motor drive frequency to motor drive voltage (commonly referred to as the V/f table).

12.2.1.118 RESP_ID_TARGET_ACIM

Definition:

```
#define RESP_ID_TARGET_ACIM
```

Description:

The response returned by the CMD_ID_TARGET command for an AC induction motor drive.

12.2.1.119 RESP_ID_TARGET_BLDC

Definition:

```
#define RESP_ID_TARGET_BLDC
```

Description:

The response returned by the CMD_ID_TARGET command for a BLDC motor drive.

12.2.1.120 RESP_ID_TARGET_STEPPER

Definition:

```
#define RESP_ID_TARGET_STEPPER
```

Description:

The response returned by the CMD_ID_TARGET command for a stepper motor drive.

12.2.1.121 TAG_CMD

Definition:

```
#define TAG_CMD
```

Description:

The value of the `{tag}` byte for a command packet.

12.2.1.122 TAG_DATA

Definition:

```
#define TAG_DATA
```

Description:

The value of the `{tag}` byte for a real-time data packet.

12.2.1.123 TAG_STATUS

Definition:

```
#define TAG_STATUS
```

Description:

The value of the `{tag}` byte for a status packet.

12.2.1.124 UISERIAL_MAX_RECV

Definition:

```
#define UISERIAL_MAX_RECV
```

Description:

The size of the UART receive buffer. This should be appropriately sized such that the maximum size command packet can be contained in this buffer. This value should be a power of two in order to make the modulo arithmetic be fast (that is, an AND instead of a divide).

12.2.1.125 UISERIAL_MAX_XMIT

Definition:

```
#define UISERIAL_MAX_XMIT
```

Description:

The size of the UART transmit buffer. This should be appropriately sized such that the maximum burst of output data can be contained in this buffer. This value should be a power of two in order to make the modulo arithmetic be fast (that is, an AND instead of a divide).

12.2.2 Function Documentation

12.2.2.1 UART0IntHandler

Handles the UART interrupt.

Prototype:

```
void
UART0IntHandler(void)
```

Description:

This is the interrupt handler for the UART. It will write new data to the UART when there is data to be written, and read new data from the UART when it is available. Reception of new data results in the receive buffer being scanned for command packets.

Returns:

None.

12.2.2.2 UISerialFindParameter [static]

Finds a parameter by ID.

Prototype:

```
static unsigned long
UISerialFindParameter(unsigned char ucID)
```

Parameters:

ucID is the ID of the parameter to locate.

Description:

This function searches the list of parameters looking for one that matches the provided ID.

Returns:

Returns the index of the parameter found, or 0xffff.ffff if the parameter does not exist in the parameter list.

12.2.2.3 UISerialInit

Initializes the serial user interface.

Prototype:

```
void
UISerialInit(void)
```

Description:

This function prepares the serial user interface for operation. The UART is configured for 115,200, 8-N-1 operation. This function should be called before any other serial user interface operations.

Returns:

None.

12.2.2.4 UISerialRangeCheck [static]

Performs range checking on the value of a parameter.

Prototype:

```
static void
UISerialRangeCheck(unsigned long ulIdx)
```

Parameters:

ulIdx is the index of the parameter to check.

Description:

This function will perform range checking on the value of a parameter, adjusting the parameter value if necessary to make it reside within the predetermined range.

Returns:

None.

12.2.2.5 UISerialScanReceive [static]

Scans for packets in the receive buffer.

Prototype:

```
static void
UISerialScanReceive(void)
```

Description:

This function will scan through g_pucUISerialReceive looking for valid command packets. When found, the command packets will be handled.

Returns:

None.

12.2.2.6 UISerialSendRealTimeData

Sends a real-time data packet.

Prototype:

```
void
UISerialSendRealTimeData(void)
```

Description:

This function will construct a real-time data packet with the current values of the enabled real-time data items. Once constructed, the packet will be sent out.

Returns:

None.

12.2.2.7 UISerialTransmit [static]

Transmits a packet to the UART.

Prototype:

```
static tBoolean
UISerialTransmit(unsigned char *pucBuffer)
```

Parameters:

pucBuffer is a pointer to the packet to be transmitted.

Description:

This function will send a packet via the UART. It will compute the checksum of the packet (based on the length in the second byte) and place it at the end of the packet before sending the packet. If g_pucUISerialTransmit is empty and there is space in the UART's FIFO, as much of the packet as will fit will be written directly to the UART's FIFO. The remainder of the packet will be buffered for later transmission when space becomes available in the UART's FIFO (which will then be written by the UART interrupt handler).

Returns:

Returns **true** if the entire packet fit into the combination of the UART's FIFO and g_pucUISerialTransmit, and **false** otherwise.

12.2.3 Variable Documentation

12.2.3.1 g_bEnableRealTimeData [static]

Definition:

```
static tBoolean g_bEnableRealTimeData
```

Description:

A boolean that is true when the real-time data stream is enabled.

12.2.3.2 g_pucUISerialData [static]

Definition:

```
static unsigned char g_pucUISerialData[UISERIAL_MAX_XMIT]
```

Description:

A buffer used to construct real-time data packets before they are written to the UART and/or g_pucUISerialTransmit.

12.2.3.3 g_pucUISerialReceive [static]

Definition:

```
static unsigned char g_pucUISerialReceive[UISERIAL_MAX_RECV]
```

Description:

A buffer to contain data received from the UART. A packet is processed out of this buffer once the entire packet is contained within the buffer.

12.2.3.4 g_pucUISerialResponse [static]

Definition:

```
static unsigned char g_pucUISerialResponse[UISERIAL_MAX_XMIT]
```

Description:

A buffer used to construct status packets before they are written to the UART and/or g_pucUISerialTransmit.

12.2.3.5 g_pucUISerialTransmit [static]

Definition:

```
static unsigned char g_pucUISerialTransmit[UISERIAL_MAX_XMIT]
```

Description:

A buffer to contain data to be written to the UART.

12.2.3.6 g_pulUIRealTimeData [static]

Definition:

```
static unsigned long g_pulUIRealTimeData[(DATA_NUM_ITEMS+31)/32]
```

Description:

A bit array that contains a flag for each real-time data item. When the corresponding flag is set, that real-time data item is enabled in the real-time data stream; when the flag is clear, that real-time data item is not part of the real-time data stream.

12.2.3.7 g_ulUISerialReceiveRead [static]

Definition:

```
static unsigned long g_ulUISerialReceiveRead
```

Description:

The offset of the next byte to be read from g_pucUISerialReceive.

12.2.3.8 g_ulUISerialReceiveWrite [static]

Definition:

```
static unsigned long g_ulUISerialReceiveWrite
```

Description:

The offset of the next byte to be written to g_pucUISerialReceive.

12.2.3.9 g_ulUISerialTransmitRead [static]

Definition:

```
static unsigned long g_ulUISerialTransmitRead
```

Description:

The offset of the next byte to be read from g_pucUISerialTransmit.

12.2.3.10 g_ulUISerialTransmitWrite [static]

Definition:

```
static unsigned long g_ulUISerialTransmitWrite
```

Description:

The offset of the next byte to be written to g_pucUISerialTransmit.

13 Sine Wave Modulation

13.1 Introduction

Sine wave modulation is used for driving single-phase AC induction motors and is a method of driving three-phase AC induction motors. Two or three sine waves, with the appropriate phase shift (180 degrees for single-phase motors and 120 degrees for three-phase motors) are produced.

For single-phase motors, this produces an alternating current in the single motor winding, exactly as would be seen by simply connecting the motor to the mains power. The amplitude of the voltage applied to the motor is the full DC bus voltage.

For three-phase motors, this produces an alternating current between each winding pair. The difference between sine waves that are 120 degrees out of phase is a sine wave with an amplitude of $\sim$86.6% the amplitude of the original sine waves. Therefore, the full DC bus is not utilized.

In order to obtain full DC bus utilization with three-phase motors, over-modulation is supported by specifying an amplitude greater than one. With over-modulation, the portion of the sine wave that is greater than one is clipped to one and the portion less than negative one is clipped to negative one. During the portion of the sine wave that has been flat-topped, the phase-to-phase current will exceed 86.6%; once the pair of flat-tops start to line up, full DC bus utilization will be achieved. This downside to over-modulation is an increase in the harmonic distortion of the drive waveforms.

The code for producing sine wave modulated waveforms is contained in `sinemod.c`, with `sinemod.h` containing the definition for the function exported to the remainder of the application.

13.2 Definitions

Functions

- void SineModulate (unsigned long ulAngle, unsigned long ulAmplitude, unsigned long *pul-DutyCycles)

13.2.1 Function Documentation

13.2.1.1 SineModulate

Computes sine wave modulated waveforms.

Prototype:

```
void
SineModulate(unsigned long ulAngle,
```

```
                unsigned long ulAmplitude,
                unsigned long *pulDutyCycles)
```

Parameters:

ulAngle is the current angle of the waveform expressed as a 0.32 fixed point value that is the percentage of the way around a circle.

ulAmplitude is the amplitude of the waveform, as a 16.16 fixed point value.

pulDutyCycles is a pointer to an array of three unsigned longs to be filled in with the duty cycles of the waveforms, in 16.16 fixed point values between zero and one.

Description:

This function finds the duty cycle percentage of the sine waveforms for the given angle. For three-phase operation, there are three waveforms produced, each 120 degrees apart. For single-phase operation, there are two waveforms produced, each 180 degrees apart. If the amplitude of the waveform is larger than one, the waveform will be clipped after scaling (flat-topping).

Returns:

None.

14 Space Vector Modulation

14.1 Introduction

Space vector modulation is a method used for driving three-phase AC induction motors. For each phase of the motor, the corresponding gates will be in one of two states; either the high-side will be on or the low-side will be on. Therefore, for the three phases, there are eight possible states for the gates (indicating which gate is on):

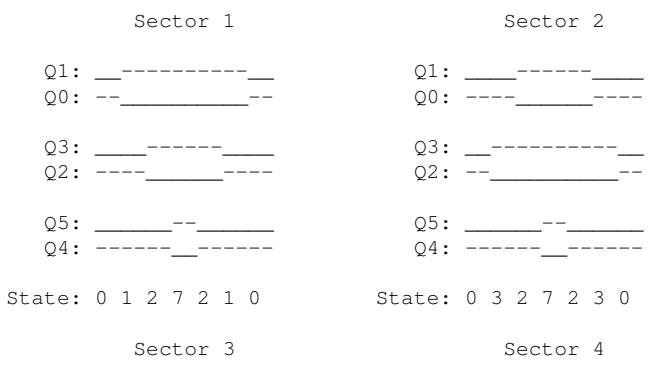
State	U Gate	V Gate	W Gate
0	low	low	low
1	high	low	low
2	high	high	low
3	low	high	low
4	low	high	high
5	low	low	high
6	high	low	high
7	high	high	high

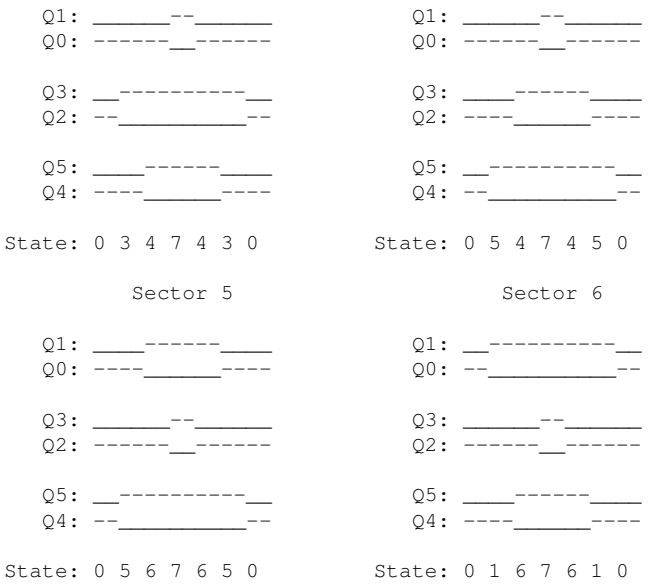
Two of those state vectors (state 0 and 7) result in no current flowing through the motor and are referred to as the zero vectors. The remaining six state vectors result in current flow, and each is spaced every 60 degrees around the circle. Between these state vectors is a sector of the circle.

Every angle will fall into one of these sectors, which is bound by two of the state vectors. Outputting the two state vectors for the appropriate time, and using the zero vectors for the remaining time in the PWM period, any angle and amplitude can be produced.

This process results in full utilization of the DC bus; for any angle, the two active state vectors are scaled such that the combined vector reaches the desired amplitude, and is capable of reaching the full DC bus amplitude.

The following waveforms show the appearance of the PWM signals in each sector of the circle, along with the state vectors in use. In each drawing, Q0 is the low-side gate for the U phase, Q1 is the high-side gate drive for the U phase, Q2 is the low-side gate for the V phase, Q3 is the high-side gate for the V phase, Q4 is the low-side gate for the W phase, and Q5 is the high-side gate for the W phase.





Proper balancing of these states results in phase-to-phase sinusoidal waveforms being presented to the motor, just as occurs with sine wave modulation. The real benefit is full utilization of the DC bus, providing more torque from the motor.

The code for producing space vector modulated waveforms is contained in `svm.c`, with `svm.h` containing the definition for the function exported to the remainder of the application.

14.2 Definitions

Functions

- void SpaceVectorModulate (unsigned long ulAngle, unsigned long ulAmplitude, unsigned long *pulDutyCycles)

14.2.1 Function Documentation

14.2.1.1 SpaceVectorModulate

Computes space vector modulated waveforms.

Prototype:

```
void
SpaceVectorModulate(unsigned long ulAngle,
                    unsigned long ulAmplitude,
                    unsigned long *pulDutyCycles)
```

Parameters:

ulAngle is the current angle of the waveform expressed as a 0.32 fixed point value that is the percentage of the way around a circle.

ulAmplitude is the amplitude of the waveform, as a 16.16 fixed point value.

pulDutyCycles is a pointer to an array of three unsigned longs to be filled in with the duty cycles of the waveforms, in 16.16 fixed point values between zero and one.

Description:

This function finds the duty cycle percentages of the space vector modulated waveforms for the given angle. If the input amplitude is greater than one, it will be clipped to one before computing the waveforms.

Returns:

None.

15 Speed Sensing

15.1 Introduction

Since an AC induction motor is not a synchronous machine, the speed of the rotor does not match the speed of the drive. Therefore, an input is provided for connecting to either an optical encoder or a tachogenerator from which the rotor speed can be determined.

When running at slow speeds, the time between input edges is measured to determine the speed of the rotor (referred to as edge timing mode). The edge triggering capability of the GPIO module is used for this measurement.

When running at higher speeds, the number of edges in a fixed time period are counted to determine the speed of the rotor (referred to as edge count mode). The velocity capture feature of the quadrature encoder module is used for this measurement.

The transition between the two speed capture modes is performed based on the measured speed. If in edge timing mode, when the edge time gets too small (that is, there are too many edges per second), it will change into edge count mode. If in edge count mode, when the number of edges in the time period gets too small (that is, there are not enough edges per time period), it will change into edge timing mode. There is a bit of hysteresis on the changeover point to avoid constantly switching between modes if the rotor is running near the changeover point.

The code for sensing the rotor speed is contained in `speed_sense.c`, with `speed_sense.h` containing the definitions for the variable and functions exported to the remainder of the application.

15.2 Definitions

Defines

- EDGE_DELTA
- FLAG_COUNT_BIT
- FLAG_EDGE_BIT
- FLAG_SKIP_BIT
- MAX_EDGE_COUNT
- QEI_INT_RATE

Functions

- void GPIOCIntHandler (void)
- void QEIIntHandler (void)
- static void SpeedNewValue (unsigned short usFrequency)

- void SpeedSenseInit (void)

Variables

- static unsigned long g_ulSpeedFlags
- static unsigned long g_ulSpeedPrevious
- static unsigned long g_ulSpeedTime
- unsigned short g_usRotorFrequency

15.2.1 Define Documentation

15.2.1.1 EDGE_DELTA

Definition:

```
#define EDGE_DELTA
```

Description:

The hysteresis applied to MAX_EDGE_COUNT when changing between the two speed determination modes.

15.2.1.2 FLAG_COUNT_BIT

Definition:

```
#define FLAG_COUNT_BIT
```

Description:

The bit number of the flag in g_ulSpeedFlags that indicates that edge counting mode is being used to determine the speed.

15.2.1.3 FLAG_EDGE_BIT

Definition:

```
#define FLAG_EDGE_BIT
```

Description:

The bit number of the flag in g_ulSpeedFlags that indicates that an edge has been seen by the edge timing mode. If an edge hasn't been seen during a QEI velocity interrupt period, the speed is forced to zero.

15.2.1.4 FLAG_SKIP_BIT

Definition:

```
#define FLAG_SKIP_BIT
```

Description:
The bit number of the flag in g_ulSpeedFlags that indicates that the next edge should be ignored by the edge timing mode. This is used when the edge timing mode is first enabled since there is no previous edge time to be used to calculate the time between edges.

15.2.1.5 MAX_EDGE_COUNT

Definition:
```
#define MAX_EDGE_COUNT
```

Description:
The maximum number of edges per second allowed when using the edge timing mode of speed determination (which is also the minimum number of edges per second allowed when using the edge count mode).

15.2.1.6 QEI_INT_RATE

Definition:
```
#define QEI_INT_RATE
```

Description:
The rate at which the QEI velocity interrupt occurs.

15.2.2 Function Documentation

15.2.2.1 GPIOCIntHandler

Handles the GPIO port C interrupt.

Prototype:
```
void
GPIOCIntHandler(void)
```

Description:
This function is called when GPIO port C asserts its interrupt. GPIO port C is configured to generate an interrupt on the rising edge of the encoder input signal. The time between the current edge and the previous edge is computed and used as a measure of the rotor frequency.

Returns:
None.

15.2.2.2 QEIIntHandler

Handles the QEI velocity interrupt.

Prototype:
```
void
QEIIntHandler(void)
```

Description:
This function is called when the QEI velocity timer expires. If using the edge counting mode for rotor frequency determination, the number of edges counted during the last velocity period is used as a measure of the rotor frequency.

Returns:
None.

15.2.2.3 SpeedNewValue [static]

Updates the current rotor frequency.

Prototype:
```
static void
SpeedNewValue(unsigned short usFrequency)
```

Parameters:
usFrequency is the newly measured frequency.

Description:
This function takes a newly measured rotor frequency and uses it to update the current rotor frequency. If the new frequency is different from the current frequency by too large a margin, the new frequency measurement is discarded (a noise filter). If the new frequency is accepted, it is passed through a single-pole IIR low pass filter with a coefficient of 0.75.

Returns:
None.

15.2.2.4 SpeedSenseInit

Initializes the speed sensing routines.

Prototype:
```
void
SpeedSenseInit(void)
```

Description:
This function will initialize the peripherals used determine the speed of the motor's rotor.

Returns:
None.

15.2.3 Variable Documentation

15.2.3.1 g_ulSpeedFlags [static]

Definition:
```
static unsigned long g_ulSpeedFlags
```

Description:
A set of flags that indicate the current state of the motor speed determination routines.

15.2.3.2 g_ulSpeedPrevious [static]

Definition:
```
static unsigned long g_ulSpeedPrevious
```

Description:
In edge timing mode, this is the time at which the previous edge was seen and is used to determine the time between edges. In edge count mode, this is the count of edges during the previous timing period and is used to average the edge count from two periods.

15.2.3.3 g_ulSpeedTime [static]

Definition:
```
static unsigned long g_ulSpeedTime
```

Description:
The time accumulated during the QEI velocity interrupts. This is used to extend the precision of the QEI timer.

15.2.3.4 g_usRotorFrequency

Definition:
```
unsigned short g_usRotorFrequency
```

Description:
The current frequency of the motor's rotor.

16 User Interface

16.1 Introduction

There are two user interfaces for the the AC induction motor application. One uses an on-board potentiometer and push button for basic control of the motor and four LEDs for basic status feedback, and the other uses the serial port to provide complete control of all aspects of the motor drive as well as monitoring of real-time performance data.

The on-board user interface consists of a potentiometer, push button, and four LEDs. The potentiometer is not directly sampled; it controls the frequency of an oscillator whose output is passed through the isolation barrier. The potentiometer value is determined by measuring the time between edges from the oscillator. The potentiometer controls the frequency of the motor drive, and the push button cycles between run forward, stop, run backward, stop. Holding the push button for five seconds while the motor drive is stopped will toggle between sine wave modulation and space vector modulation.

The “Run” LED flashes the entire time the application is running. The LED is off most of the time if the motor drive is stopped and on most of the time if it is running. The “Fault” LED is normally off but flashes at a fast rate when a fault occurs. Also, it flashes slowly when the in-rush current limiter is operating on application startup. The “S1” LED is on when the dynamic brake is active and off when it is not active. And the “S2” LED is on when space vector modulation is being used and off when sine wave modulation is being used.

A periodic interrupt is used to poll the state of the push button and perform debouncing. A separate edge-triggered GPIO interrupt is used to measure the time between edges from the potentiometer-controlled oscillator.

The serial user interface is entirely handled by the serial user interface module. The only thing provided here is the list of parameters and real-time data items, plus a set of helper functions that are required in order to properly set the values of some of the parameters.

This user interface (and the accompanying serial and on-board user interface modules) is more complicated and consumes more program space than would typically exist in a real motor drive application. The added complexity allows a great deal of flexibility to configure and evaluate the motor drive, its capabilities, and adjust it for the target motor.

The code for the user interface is contained in `ui.c`, with `ui.h` containing the definitions for the structures, defines, variables, and functions exported to the remainder of the application.

16.2 Definitions

Data Structures

- tDriveParameters

Defines

- FLAG_BRAKE_BIT
- FLAG_BRAKE_OFF
- FLAG_BRAKE_ON
- FLAG_BUS_COMP_BIT
- FLAG_BUS_COMP_OFF
- FLAG_BUS_COMP_ON
- FLAG_DC_BRAKE_BIT
- FLAG_DC_BRAKE_OFF
- FLAG_DC_BRAKE_ON
- FLAG_DIR_BACKWARD
- FLAG_DIR_BIT
- FLAG_DIR_FORWARD
- FLAG_DRIVE_BIT
- FLAG_DRIVE_SINE
- FLAG_DRIVE_SPACE_VECTOR
- FLAG_ENCODER_ABSENT
- FLAG_ENCODER_BIT
- FLAG_ENCODER_PRESENT
- FLAG_LOOP_BIT
- FLAG_LOOP_CLOSED
- FLAG_LOOP_OPEN
- FLAG_MOTOR_TYPE_1PHASE
- FLAG_MOTOR_TYPE_3PHASE
- FLAG_MOTOR_TYPE_BIT
- FLAG_PWM_FREQUENCY_12K
- FLAG_PWM_FREQUENCY_16K
- FLAG_PWM_FREQUENCY_20K
- FLAG_PWM_FREQUENCY_8K
- FLAG_PWM_FREQUENCY_MASK
- FLAG_VF_RANGE_100
- FLAG_VF_RANGE_400
- FLAG_VF_RANGE_BIT
- NUM_SWITCHES
- UI_INT_RATE
- UI_POT_MAX
- UI_POT_MIN

Functions

- void GPIODIntHandler (void)
- void SysTickIntHandler (void)
- static void UIBusComp (void)
- static void UIButtonHold (void)

- static void UIButtonPress (void)
- static void UIDCBrake (void)
- static void UIDirectionSet (void)
- static void UIDynamicBrake (void)
- void UIEmergencyStop (void)
- static void UIEncoderPresent (void)
- static void UIFAdjI (void)
- void UIFaultLEDBlink (unsigned short usRate, unsigned short usPeriod)
- void UIInit (void)
- static void UILEDBlink (unsigned long ulIdx, unsigned short usRate, unsigned short usPeriod)
- static void UILoopMode (void)
- static void UIModulationType (void)
- static void UIMotorType (void)
- void UIParamLoad (void)
- void UIParamSave (void)
- static void UIPWMFrequencySet (void)
- void UIRun (void)
- void UIRunLEDBlink (unsigned short usRate, unsigned short usPeriod)
- void UIStatus1LEDBlink (unsigned short usRate, unsigned short usPeriod)
- void UIStatus2LEDBlink (unsigned short usRate, unsigned short usPeriod)
- void UIStop (void)
- static void UIUpdateRate (void)
- void UIUpgrade (void)
- static void UIVfRange (void)

Variables

- static long g_lFAdjI
- static const unsigned char g_pucLEDPin[4]
- static const unsigned long g_pulLEDBase[4]
- unsigned long g_pulUIHoldCount[NUM_SWITCHES]
- static unsigned short g_pusBlinkPeriod[4]
- static unsigned short g_pusBlinkRate[4]
- tDriveParameters g_sParameters
- const tUIParameter g_sUIParameters[ ]
- const tUIRealTimeData g_sUIRealTimeData[ ]
- const tUIOnboardSwitch g_sUISwitches[ ]
- static unsigned char g_ucBusComp
- unsigned char g_ucCPUUsage
- static unsigned char g_ucDCBrake
- static unsigned char g_ucDirection
- static unsigned char g_ucDynamicBrake
- static unsigned char g_ucEncoder
- static unsigned char g_ucFrequency
- static unsigned char g_ucLoop

- static unsigned char g_ucModulation
- static unsigned char g_ucType
- static unsigned char g_ucUpdateRate
- static unsigned char g_ucVfRange
- static unsigned long g_ulBlinkCount
- const unsigned long g_ulUINumButtons
- const unsigned long g_ulUINumParameters
- const unsigned long g_ulUINumRealTimeData
- static unsigned long g_ulUIPotEdgeTime
- static unsigned long g_ulUIPotPreviousTime
- const unsigned long g_ulUITargetType
- static unsigned long g_ulUIUseOnboard
- unsigned short g_usCurrentFrequency
- const unsigned short g_usFirmwareVersion
- unsigned short g_usTargetFrequency

16.2.1 Data Structure Documentation

16.2.1.1 tDriveParameters

Definition:

```
typedef struct
{
    unsigned char ucSequenceNum;
    unsigned char ucCRC;
    unsigned char ucVersion;
    unsigned char ucMinPulseWidth;
    unsigned char ucDeadTime;
    unsigned char ucUpdateRate;
    unsigned char ucNumPoles;
    unsigned char ucAccel;
    unsigned char ucDecel;
    unsigned char ucMinCurrent;
    unsigned char ucMaxCurrent;
    unsigned char ucPrechargeTime;
    unsigned char ucMaxTemperature;
    unsigned short usFlags;
    unsigned short usNumEncoderLines;
    unsigned short usMinFrequency;
    unsigned short usMaxFrequency;
    unsigned short usMinVBus;
    unsigned short usMaxVBus;
    unsigned short usBrakeOnV;
    unsigned short usBrakeOffV;
    unsigned short usDCBrakeV;
    unsigned short usDCBrakeTime;
    unsigned short usDecelV;
    unsigned short usVFTable[21];
    long lFAdjP;
```

```
    long lFAdjI;
    unsigned long ulBrakeMax;
    unsigned long ulBrakeCool;
    unsigned char ucAccelCurrent;
    unsigned char ucReserved[31];
}
tDriveParameters
```

Members:

ucSequenceNum The sequence number of this parameter block. When in RAM, this value is not used. When in flash, this value is used to determine the parameter block with the most recent information.

ucCRC The CRC of the parameter block. When in RAM, this value is not used. When in flash, this value is used to validate the contents of the parameter block (to avoid using a partially written parameter block).

ucVersion The version of this parameter block. This can be used to distinguish saved parameters that correspond to an old version of the parameter block.

ucMinPulseWidth The minimum width of a PWM pulse, specified in 0.1 us periods.

ucDeadTime The dead time between inverting the high and low side of a motor phase, specified in 20 ns periods.

ucUpdateRate The rate at which the PWM pulse width is updated, specified in the number of PWM periods.

ucNumPoles The number of pole pairs in the motor.

ucAccel The rate of acceleration, specified in Hertz per second.

ucDecel The rate of deceleration, specified in Hertz per second.

ucMinCurrent The minimum current through the motor drive during operation, specified in 1/10ths of an ampere.

ucMaxCurrent The maximum current through the motor drive during operation, specified in 1/10ths of an ampere.

ucPrechargeTime The amount of time to precharge the bootstrap capacitor on the high side gate drivers, specified in milliseconds.

ucMaxTemperature The maximum ambient temperature of the microcontroller, specified in degrees Celsius.

usFlags A set of flags, enumerated by FLAG_PWM_FREQUENCY_MASK, FLAG_MOTOR_TYPE_BIT, FLAG_LOOP_BIT, FLAG_DRIVE_BIT, FLAG_DIR_BIT, FLAG_ENCODER_BIT, FLAG_VF_RANGE_BIT, FLAG_BUS_COMP_BIT, FLAG_BRAKE_BIT, and FLAG_DC_BRAKE_BIT.

usNumEncoderLines The number of lines in the (optional) optical encoder.

usMinFrequency The minimum frequency of the motor drive, specified in 1/10ths of a Hertz.

usMaxFrequency The maximum frequency of the motor drive, specified in 1/10ths of a Hertz.

usMinVBus The minimum bus voltage during operation, specified in volts.

usMaxVBus The maximum bus voltage during operation, specified in volts.

usBrakeOnV The bus voltage at which the braking circuit is engaged, specified in volts.

usBrakeOffV The bus voltage at which the braking circuit is disengaged, specified in volts.

usDCBrakeV The voltage to be applied to the motor when performing DC injection braking, specified in volts.

usDCBrakeTime The amount of time to apply DC injection braking, specified in milliseconds.

usDecelV The DC bus voltage at which the deceleration rate is reduced, specified in volts.

usVFTable An array of coefficients that map from motor frequency to waveform amplitude, known as V/f control. The first entry of this array corresponds to the minimum motor drive frequency, the last entry corresponds to the nominal motor drive frequency, and the other entries are equally spaced between the first and last. For frequencies that do not appear in this table, linear interpolation is used to approximate the appropriate amplitude. Each entry is in a 1.15 fixed point format.

lFAdjP The P coefficient of the frequency adjust PI controller.

lFAdjI The I coefficient of the frequency adjust PI controller.

ulBrakeMax The amount of time (assuming continuous application) that the dynamic braking can be utilized, specified in milliseconds.

ulBrakeCool The amount of accumulated time that the dynamic brake can have before the cooling period will end, specified in milliseconds.

ucAccelCurrent The motor current at which the acceleration rate is reduced, specified in 1/10ths of an ampere. Note: This parameter only exists in the version one parameter block.

ucReserved The amount of unused space in the structure in order to pad it to 128 bytes for storage into flash.

Description:

This structure contains the AC induction motor parameters that are saved to flash. A copy exists in RAM for use during the execution of the application, which is loaded form flash at startup. The modified parameter block can also be written back to flash for use on the next power cycle.

Note: All parameters exist in the version zero parameter block unless it is explicitly stated otherwise. If an older parameter block is loaded from flash, the new parameters will get filled in with default values. When the parameter block is written to flash, it will always be written with the latest parameter block version.

16.2.2 Define Documentation

16.2.2.1 FLAG_BRAKE_BIT

Definition:

```
#define FLAG_BRAKE_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the application of dynamic brake to handle regeneration onto DC bus. This field will be one of FLAG_BRAKE_ON or FLAG_BRAKE_OFF.

16.2.2.2 FLAG_BRAKE_OFF

Definition:

```
#define FLAG_BRAKE_OFF
```

Description:

The value of the FLAG_BRAKE_BIT flag that indicates that the dynamic brake is disabled.

16.2.2.3 FLAG_BRAKE_ON

Definition:

```
#define FLAG_BRAKE_ON
```

Description:

The value of the FLAG_BRAKE_BIT flag that indicates that the dynamic brake is enabled.

16.2.2.4 FLAG_BUS_COMP_BIT

Definition:

```
#define FLAG_BUS_COMP_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the application of amplitude compensation for fluctuations in the DC bus voltage. This field will be one of FLAG_BUS_COMP_ON or FLAG_BUS_COMP_OFF.

16.2.2.5 FLAG_BUS_COMP_OFF

Definition:

```
#define FLAG_BUS_COMP_OFF
```

Description:

The value of the FLAG_BUS_COMP_BIT flag that indicates that the DC bus compensation is disabled.

16.2.2.6 FLAG_BUS_COMP_ON

Definition:

```
#define FLAG_BUS_COMP_ON
```

Description:

The value of the FLAG_BUS_COMP_BIT flag that indicates that the DC bus compensation is enabled.

16.2.2.7 FLAG_DC_BRAKE_BIT

Definition:

```
#define FLAG_DC_BRAKE_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the application of the DC injection brake to stop the motor. This field will be one of FLAG_DC_BRAKE_ON or FLAG_DC_BRAKE_OFF.

16.2.2.8 FLAG_DC_BRAKE_OFF

Definition:

```
#define FLAG_DC_BRAKE_OFF
```

Description:

The value of the FLAG_DC_BRAKE_BIT flag that indicates that the DC injection brake is disabled.

16.2.2.9 FLAG_DC_BRAKE_ON

Definition:

```
#define FLAG_DC_BRAKE_ON
```

Description:

The value of the FLAG_DC_BRAKE_BIT flag that indicates that the DC injection brake is enabled.

16.2.2.10 FLAG_DIR_BACKWARD

Definition:

```
#define FLAG_DIR_BACKWARD
```

Description:

The value of the FLAG_DIR_BIT flag that indicates that the motor is to be driven in the backward direction.

16.2.2.11 FLAG_DIR_BIT

Definition:

```
#define FLAG_DIR_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the direction the motor is to be driven. The field will be one of FLAG_DIR_FORWARD or FLAG_DIR_BACKWARD.

16.2.2.12 FLAG_DIR_FORWARD

Definition:

```
#define FLAG_DIR_FORWARD
```

Description:

The value of the FLAG_DIR_BIT flag that indicates that the motor is to be driven in the forward direction.

16.2.2.13 FLAG_DRIVE_BIT

Definition:

```
#define FLAG_DRIVE_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the type of drive waveform for the motor drive. This field will be one of FLAG_DRIVE_SINE or FLAG_DRIVE_SPACE_VECTOR.

16.2.2.14 FLAG_DRIVE_SINE

Definition:

```
#define FLAG_DRIVE_SINE
```

Description:

The value of the FLAG_DRIVE_BIT flag that indicates that the motor is to be driven with sine wave modulation.

16.2.2.15 FLAG_DRIVE_SPACE_VECTOR

Definition:

```
#define FLAG_DRIVE_SPACE_VECTOR
```

Description:

The value of the FLAG_DRIVE_BIT flag that indicates that the motor is to be driven with space vector modulation.

16.2.2.16 FLAG_ENCODER_ABSENT

Definition:

```
#define FLAG_ENCODER_ABSENT
```

Description:

The value of the FLAG_ENCODER_BIT flag that indicates that the encoder is absent.

16.2.2.17 FLAG_ENCODER_BIT

Definition:

```
#define FLAG_ENCODER_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the presence of an encoder for speed feedback. This field will be one of FLAG_ENCODER_ABSENT or FLAG_ENCODER_PRESENT.

16.2.2.18 FLAG_ENCODER_PRESENT

Definition:

```
#define FLAG_ENCODER_PRESENT
```

Description:

The value of the FLAG_ENCODER_BIT flag that indicates that the encoder is present.

16.2.2.19 FLAG_LOOP_BIT

Definition:

```
#define FLAG_LOOP_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the mode of operation for the motor drive. This field will be one of FLAG_LOOP_OPEN or FLAG_LOOP_-CLOSED.

16.2.2.20 FLAG_LOOP_CLOSED

Definition:

```
#define FLAG_LOOP_CLOSED
```

Description:

The value of the FLAG_LOOP_BIT flag field that indicates that the motor is operated in closed-loop mode.

16.2.2.21 FLAG_LOOP_OPEN

Definition:

```
#define FLAG_LOOP_OPEN
```

Description:

The value of the FLAG_LOOP_BIT flag that indicates that the motor is operated in open-loop mode.

16.2.2.22 FLAG_MOTOR_TYPE_1PHASE

Definition:

```
#define FLAG_MOTOR_TYPE_1PHASE
```

Description:

The value of the FLAG_MOTOR_TYPE_BIT flag that indicates that the motor is a single phase motor.

16.2.2.23 FLAG_MOTOR_TYPE_3PHASE

Definition:

```
#define FLAG_MOTOR_TYPE_3PHASE
```

Description:

The value of the FLAG_MOTOR_TYPE_BIT flag that indicates that the motor is a three phase motor.

16.2.2.24 FLAG_MOTOR_TYPE_BIT

Definition:

```
#define FLAG_MOTOR_TYPE_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the type of the motor. This field will be one of FLAG_MOTOR_TYPE_3PHASE or FLAG_MOTOR_-TYPE_1PHASE.

16.2.2.25 FLAG_PWM_FREQUENCY_12K

Definition:

```
#define FLAG_PWM_FREQUENCY_12K
```

Description:

The value of the FLAG_PWM_FREQUENCY_MASK bit field that indicates that the PWM frequency is 12.5 KHz.

16.2.2.26 FLAG_PWM_FREQUENCY_16K

Definition:

```
#define FLAG_PWM_FREQUENCY_16K
```

Description:

The value of the FLAG_PWM_FREQUENCY_MASK bit field that indicates that the PWM frequency is 16 KHz.

16.2.2.27 FLAG_PWM_FREQUENCY_20K

Definition:

```
#define FLAG_PWM_FREQUENCY_20K
```

Description:

The value of the FLAG_PWM_FREQUENCY_MASK bit field that indicates that the PWM frequency is 20 KHz.

16.2.2.28 FLAG_PWM_FREQUENCY_8K

Definition:

```
#define FLAG_PWM_FREQUENCY_8K
```

Description:

The value of the FLAG_PWM_FREQUENCY_MASK bit field that indicates that the PWM frequency is 8 KHz.

16.2.2.29 FLAG_PWM_FREQUENCY_MASK

Definition:

```
#define FLAG_PWM_FREQUENCY_MASK
```

Description:

The mask for the bits in the usFlags member of tDriveParameters that define the PWM output frequency. This field will be one of FLAG_PWM_FREQUENCY_8K, FLAG_PWM_FREQUENCY_12K, FLAG_PWM_FREQUENCY_16K, or FLAG_PWM_FREQUENCY_20K.

16.2.2.30 FLAG_VF_RANGE_100

Definition:

```
#define FLAG_VF_RANGE_100
```

Description:

The value of the FLAG_VF_RANGE_BIT flag that indicates that the V/f table ranges from 0 Hz to 100 Hz.

16.2.2.31 FLAG_VF_RANGE_400

Definition:

```
#define FLAG_VF_RANGE_400
```

Description:

The value of the FLAG_VF_RANGE_BIT flag that indicates that the V/f table ranges from 0 Hz to 400 Hz.

16.2.2.32 FLAG_VF_RANGE_BIT

Definition:

```
#define FLAG_VF_RANGE_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the range of the V/f table. This field will be one of FLAG_VF_RANGE_100 or FLAG_VF_RANGE_400.

16.2.2.33 NUM_SWITCHES

Definition:

```
#define NUM_SWITCHES
```

Description:

The number of switches in the g_sUISwitches array. This value is automatically computed based on the number of entries in the array.

16.2.2.34 UI_INT_RATE

Definition:

```
#define UI_INT_RATE
```

Description:

The rate at which the user interface interrupt occurs.

16.2.2.35 UI_POT_MAX

Definition:

```
#define UI_POT_MAX
```

Description:

The maximum value that can be read from the potentiometer. This corresponds to the value read when the wiper is all the way to the right.

16.2.2.36 UI_POT_MIN

Definition:

```
#define UI_POT_MIN
```

Description:

The minimum value that can be read from the potentiometer. This corresponds to the value read when the wiper is all the way to the left.

16.2.3 Function Documentation

16.2.3.1 GPIODIntHandler

Handles the GPIO port D interrupt.

Prototype:

```
void
GPIODIntHandler(void)
```

Description:
This function is called when GPIO port D asserts its interrupt. GPIO port D is configured to generate an interrupt on either edge of the signal from the potentiometer oscillator. The time between the current edge and the previous edge is computed.

Returns:
None.

16.2.3.2 SysTickIntHandler

Handles the SysTick interrupt.

Prototype:
```
void
SysTickIntHandler(void)
```

Description:
This function is called when SysTick asserts its interrupt. It is responsible for handling the on-board user interface elements (push button and potentiometer) if enabled, and the processor usage computation.

Returns:
None.

16.2.3.3 UIBusComp [static]

Updates the DC bus compensation bit of the motor drive.

Prototype:
```
static void
UIBusComp(void)
```

Description:
This function is called when the variable controlling the DC bus compensation is updated. The value is then reflected into the usFlags member of g_sParameters.

Returns:
None.

16.2.3.4 UIButtonHold [static]

Handles button holds.

Prototype:
```
static void
UIButtonHold(void)
```

Description:
This function is called when a hold of the on-board push button has been detected. The modulation type of the motor will be toggled between sine wave and space vector modulation, but only if a three phase motor is in use.

Returns:
None.

16.2.3.5 UIButtonPress [static]

Handles button presses.

Prototype:
```
static void
UIButtonPress(void)
```

Description:
This function is called when a press of the on-board push button has been detected. If the motor drive is running, it will be stopped. If it is stopped, the direction will be reversed and the motor drive will be started.

Returns:
None.

16.2.3.6 UIDCBrake [static]

Update the DC brake bit of the motor drive.

Prototype:
```
static void
UIDCBrake(void)
```

Description:
This function is called when the variable controlling the DC braking is updated. The value is then reflected into the usFlags member of g_sParameters.

Returns:
None.

16.2.3.7 UIDirectionSet [static]

Updates the motor drive direction bit.

Prototype:
```
static void
UIDirectionSet(void)
```

Description:
This function is called when the variable controlling the motor drive direction is updated. The value is then reflected into the usFlags member of g_sParameters.

Returns:
None.

16.2.3.8 UIDynamicBrake [static]

Updates the dynamic brake bit of the motor drive.

Prototype:

```
static void
UIDynamicBrake(void)
```

Description:

This function is called when the variable controlling the dynamic braking is updated. The value is then reflected into the usFlags member of g_sParameters.

Returns:

None.

16.2.3.9 UIEmergencyStop

Emergency stops the motor drive.

Prototype:

```
void
UIEmergencyStop(void)
```

Description:

This function is called by the serial user interface when the emergency stop command is received. In the case of an AC induction motor, an emergency stop is treated as a "protect the motor drive" command; mechanical braking must be utilized in an emergency stop situation.

Returns:

None.

16.2.3.10 UIEncoderPresent [static]

Updates the encoder presence bit of the motor drive.

Prototype:

```
static void
UIEncoderPresent(void)
```

Description:

This function is called when the variable controlling the presence of an encoder is updated. The value is then reflected into the usFlags member of g_sParameters.

Returns:

None.

16.2.3.11 UIFAdjI [static]

Updates the I coefficient of the frequency PI controller.

Prototype:

```
static void
UIFAdjI(void)
```

Description:

This function is called when the variable containing the I coefficient of the frequency PI controller is updated. The value is then reflected into the parameter block.

Returns:

None.

16.2.3.12 UIFaultLEDBlink

Sets the blink rate for the fault LED.

Prototype:

```
void
UIFaultLEDBlink(unsigned short usRate,
                unsigned short usPeriod)
```

Parameters:

usRate is the rate to blink the fault LED.

usPeriod is the amount of time to turn on the fault LED.

Description:

This function sets the rate at which the fault LED should be blinked. A blink period of zero means that the LED should be turned off, and a blink period equal to the blink rate means that the LED should be turned on. Otherwise, the blink rate determines the number of user interface interrupts during the blink cycle of the fault LED, and the blink period is the number of those user interface interrupts during which the LED is turned on.

Returns:

None.

16.2.3.13 UIInit

Initializes the user interface.

Prototype:

```
void
UIInit(void)
```

Description:

This function initializes the user interface modules (on-board and serial), preparing them to operate and control the motor drive.

Returns:

None.

16.2.3.14 UILEDBlink [static]

Sets the blink rate for an LED.

Prototype:

```
static void
UILEDBlink(unsigned long ulIdx,
           unsigned short usRate,
           unsigned short usPeriod)
```

Parameters:

ulIdx is the number of the LED to configure.

usRate is the rate to blink the LED.

usPeriod is the amount of time to turn on the LED.

Description:

This function sets the rate at which an LED should be blinked. A blink period of zero means that the LED should be turned off, and a blink period equal to the blink rate means that the LED should be turned on. Otherwise, the blink rate determines the number of user interface interrupts during the blink cycle of the LED, and the blink period is the number of those user interface interrupts during which the LED is turned on.

Returns:

None.

16.2.3.15 UILoopMode [static]

Updates the open-/closed-loop mode bit of the motor drive.

Prototype:

```
static void
UILoopMode(void)
```

Description:

This function is called when the variable controlling open-/closed-loop mode of the motor drive is updated. The value is then reflected into the usFlags member of g_sParameters.

Returns:

None.

16.2.3.16 UIModulationType [static]

Updates the modulation waveform type bit in the motor drive.

Prototype:

```
static void
UIModulationType(void)
```

Description:

This function is called when the variable controlling the modulation waveform type is updated. The value is then reflected into the usFlags member of g_sParameters.

Returns:
None.

16.2.3.17 UIMotorType [static]

Updates the type of motor connected to the motor drive.

Prototype:
```
static void
UIMotorType(void)
```

Description:
This function is called when the variable specifying the type of motor connected to the motor drive is updated. This value is then reflected into the usFlags member of g_sParameters.

Returns:
None.

16.2.3.18 UIParamLoad

Loads the motor drive parameter block from flash.

Prototype:
```
void
UIParamLoad(void)
```

Description:
This function is called by the serial user interface when the load parameter block function is called. If the motor drive is running, the parameter block is not loaded (since that may result in detrimental changes, such as changing the motor type from three phase to single phase). If the motor drive is not running and a valid parameter block exists in flash, the contents of the parameter block are loaded from flash.

Returns:
None.

16.2.3.19 UIParamSave

Saves the motor drive parameter block to flash.

Prototype:
```
void
UIParamSave(void)
```

Description:
This function is called by the serial user interface when the save parameter block function is called. The parameter block is written to flash for use the next time a load occurs (be it from an explicit request or a power cycle of the drive).

Returns:
None.

16.2.3.20 UIPWMFrequencySet [static]

Updates the PWM frequency of the motor drive.

Prototype:

```
static void
UIPWMFrequencySet(void)
```

Description:

This function is called when the variable controlling the PWM frequency of the motor drive is updated. The value is then reflected into the usFlags member of g_sParameters.

Returns:

None.

16.2.3.21 UIRun

Starts the motor drive.

Prototype:

```
void
UIRun(void)
```

Description:

This function is called by the serial user interface when the run command is received. The motor drive will be started as a result; this is a no operation if the motor drive is already running.

Returns:

None.

16.2.3.22 UIRunLEDBlink

Sets the blink rate for the run LED.

Prototype:

```
void
UIRunLEDBlink(unsigned short usRate,
              unsigned short usPeriod)
```

Parameters:

usRate is the rate to blink the run LED.
usPeriod is the amount of time to turn on the run LED.

Description:

This function sets the rate at which the run LED should be blinked. A blink period of zero means that the LED should be turned off, and a blink period equal to the blink rate means that the LED should be turned on. Otherwise, the blink rate determines the number of user interface interrupts during the blink cycle of the run LED, and the blink period is the number of those user interface interrupts during which the LED is turned on.

Returns:

None.

16.2.3.23 UIStatus1LEDBlink

Sets the blink rate for the status1 LED.

Prototype:

```
void
UIStatus1LEDBlink(unsigned short usRate,
                  unsigned short usPeriod)
```

Parameters:

usRate is the rate to blink the status1 LED.

usPeriod is the amount of time to turn on the status1 LED.

Description:

This function sets the rate at which the status1 LED should be blinked. A blink period of zero means that the LED should be turned off, and a blink period equal to the blink rate means that the LED should be turned on. Otherwise, the blink rate determines the number of user interface interrupts during the blink cycle of the status1 LED, and the blink period is the number of those user interface interrupts during which the LED is turned on.

Returns:

None.

16.2.3.24 UIStatus2LEDBlink

Sets the blink rate for the status2 LED.

Prototype:

```
void
UIStatus2LEDBlink(unsigned short usRate,
                  unsigned short usPeriod)
```

Parameters:

usRate is the rate to blink the status2 LED.

usPeriod is the amount of time to turn on the status2 LED.

Description:

This function sets the rate at which the status2 LED should be blinked. A blink period of zero means that the LED should be turned off, and a blink period equal to the blink rate means that the LED should be turned on. Otherwise, the blink rate determines the number of user interface interrupts during the blink cycle of the status2 LED, and the blink period is the number of those user interface interrupts during which the LED is turned on.

Returns:

None.

16.2.3.25 UIStop

Stops the motor drive.

Prototype:

```
void
UIStop(void)
```

Description:

This function is called by the serial user interface when the stop command is received. The motor drive will be stopped as a result; this is a no operation if the motor drive is already stopped.

Returns:

None.

16.2.3.26 UIUpdateRate [static]

Sets the update rate of the motor drive.

Prototype:

```
static void
UIUpdateRate(void)
```

Description:

This function is called when the variable specifying the update rate of the motor drive is updated. This allows the motor drive to perform a synchronous change of the update rate to avoid discontinuities in the output waveform.

Returns:

None.

16.2.3.27 UIUpgrade

Starts a firmware upgrade.

Prototype:

```
void
UIUpgrade(void)
```

Description:

This function is called by the serial user interface when a firmware upgrade has been requested. This will branch directly to the boot loader and relinquish all control, never returning.

Returns:

Never returns.

16.2.3.28 UIVfRange [static]

Updates the V/f table range of the motor drive.

Prototype:

```
static void
UIVfRange(void)
```

Description:
This function is called when the variable controlling the V/f table range is updated. The value is then reflected into the usFlags member of g_sParameters.

Returns:
None.

16.2.4 Variable Documentation

16.2.4.1 g_lFAdjI [static]

Definition:
```
static long g_lFAdjI
```

Description:
The I coefficient of the frequency PI controller. This variable is used by the serial interface as a staging area before the value gets placed into the parameter block by UIFAdjI().

16.2.4.2 g_pucLEDPin [static]

Definition:
```
static const unsigned char g_pucLEDPin[4]
```

Description:
This array contains the pin numbers of the four LEDs on the board.

16.2.4.3 g_pulLEDBase [static]

Definition:
```
static const unsigned long g_pulLEDBase[4]
```

Description:
This array contains the base address of the GPIO blocks for the four LEDs on the board.

16.2.4.4 g_pulUIHoldCount

Definition:
```
unsigned long g_pulUIHoldCount
```

Description:
This is the count of the number of samples during which the switches have been pressed; it is used to distinguish a switch press from a switch hold. This array is used by the on-board user interface module.

16.2.4.5 g_pusBlinkPeriod [static]

Definition:

```
static unsigned short g_pusBlinkPeriod[4]
```

Description:

The blink period of the four LEDs on the board; this is the number of user interface interrupts for which the LED will be turned on. The run LED is the first entry of the array, the fault LED is the second entry of the array, the status1 LED is the third entry of the array, and the status2 LED is the fourth entry of the array.

16.2.4.6 g_pusBlinkRate [static]

Definition:

```
static unsigned short g_pusBlinkRate[4]
```

Description:

The blink rate of the four LEDs on the board; this is the number of user interface interrupts for an entire blink cycle. The run LED is the first entry of the array, the fault LED is the second entry of the array, the status1 LED is the third entry of the array, and the status2 LED is the fourth entry of the array.

16.2.4.7 g_sParameters

Definition:

```
tDriveParameters g_sParameters
```

Description:

This structure instance contains the configuration values for the AC induction motor drive.

16.2.4.8 g_sUIParameters

Definition:

```
const tUIParameter g_sUIParameters[]
```

Description:

An array of structures describing the AC induction motor drive parameters to the serial user interface module.

16.2.4.9 g_sUIRealTimeData

Definition:

```
const tUIRealTimeData g_sUIRealTimeData[]
```

Description:

An array of structures describing the AC induction motor drive real-time data items to the serial user interface module.

16.2.4.10 g_sUISwitches

Definition:

```
const tUIOnboardSwitch g_sUISwitches[]
```

Description:

An array of structures describing the on-board switches.

16.2.4.11 g_ucBusComp [static]

Definition:

```
static unsigned char g_ucBusComp
```

Description:

A boolean that is true when the DC bus voltage compensation should be active and false when it should not be. This variable is used by the serial interface as a staging area before the value gets placed into the flags in the parameter block by UIBusComp().

16.2.4.12 g_ucCPUUsage

Definition:

```
unsigned char g_ucCPUUsage
```

Description:

The processor usage for the most recent measurement period. This is a value between 0 and 100, inclusive.

16.2.4.13 g_ucDCBrake [static]

Definition:

```
static unsigned char g_ucDCBrake
```

Description:

A boolean that is true when DC injection braking should be utilized. This variable is used by the serial interface as a staging area before the value gets placed into the flags in the parameter block by UIDCBrake().

16.2.4.14 g_ucDirection [static]

Definition:

```
static unsigned char g_ucDirection
```

Description:

The specification of the motor drive direction. This variable is used by the serial interface as a staging area before the value gets placed into the flags in the parameter block by UIDirectionSet().

16.2.4.15 g_ucDynamicBrake [static]

Definition:

```
static unsigned char g_ucDynamicBrake
```

Description:

A boolean that is true when dynamic braking should be utilized. This variable is used by the serial interface as a staging area before the value gets placed into the flags in the parameter block by UIDynamicBrake().

16.2.4.16 g_ucEncoder [static]

Definition:

```
static unsigned char g_ucEncoder
```

Description:

The specification of the encoder presence on the motor. This variable is used by the serial interface as a staging area before the value gets placed into the flags in the parameter block by UIEncoderPresent().

16.2.4.17 g_ucFrequency [static]

Definition:

```
static unsigned char g_ucFrequency
```

Description:

The specification of the PWM frequency for the motor drive. This variable is used by the serial interface as a staging area before the value gets placed into the flags in the parameter block by UIPWMFrequencySet().

16.2.4.18 g_ucLoop [static]

Definition:

```
static unsigned char g_ucLoop
```

Description:

The specification of open-loop or closed-loop mode of the motor drive. This variable is used by the serial interface as a staging area before the value gets placed into the flags in the parameter block by UILoopMode().

16.2.4.19 g_ucModulation [static]

Definition:

```
static unsigned char g_ucModulation
```

Description:

The specification of the modulation waveform type for the motor drive. This variable is used by the serial interface as a staging area before the value gets placed into the flags in the parameter block by UIModulationType().

16.2.4.20 g_ucType [static]

Definition:

```
static unsigned char g_ucType
```

Description:

The specification of the type of motor connected to the motor drive. This variable is used by the serial interface as a staging area before the value gets placed into the flags in the parameter block by UIMotorType().

16.2.4.21 g_ucUpdateRate [static]

Definition:

```
static unsigned char g_ucUpdateRate
```

Description:

The specification of the update rate for the motor drive. This variable is used by the serial interface as a staging area before the value gets updated in a synchronous manner by UIUpdateRate().

16.2.4.22 g_ucVfRange [static]

Definition:

```
static unsigned char g_ucVfRange
```

Description:

A boolean that is true when the V/f table ranges from 0 Hz to 400 Hz and false when it ranges from 0 Hz to 100 Hz. This variable is used by the serial interface as a staging area before the value gets placed into the flags in the parameter block by UIVfRange().

16.2.4.23 g_ulBlinkCount [static]

Definition:

```
static unsigned long g_ulBlinkCount
```

Description:

The count of count of user interface interrupts that have occurred. This is used to determine when to toggle the LEDs that are blinking.

16.2.4.24 g_ulUINumButtons

Definition:

```
const unsigned long g_ulUINumButtons
```

Description:

The number of switches on this target. This value is used by the on-board user interface module.

16.2.4.25 g_ulUINumParameters

Definition:

```
const unsigned long g_ulUINumParameters
```

Description:

The number of motor drive parameters. This is used by the serial user interface module.

16.2.4.26 g_ulUINumRealTimeData

Definition:

```
const unsigned long g_ulUINumRealTimeData
```

Description:

The number of motor drive real-time data items. This is used by the serial user interface module.

16.2.4.27 g_ulUIPotEdgeTime [static]

Definition:

```
static unsigned long g_ulUIPotEdgeTime
```

Description:

The time between the last two edges on the potentiometer input. The potentiometer controls a variable frequency oscillator whose output is passed through the electrical isolation barrier; measuring the time between edges provides an approximation of the value of the potentiometer.

16.2.4.28 g_ulUIPotPreviousTime [static]

Definition:

```
static unsigned long g_ulUIPotPreviousTime
```

Description:

The value of the SysTick timer when the most recent edge was received on the potentiometer. When a new edge is detected, this is used to determine the time between the edges.

16.2.4.29 g_ulUITargetType

Definition:

```
const unsigned long g_ulUITargetType
```

Description:

The target type for this drive. This is used by the serial user interface module.

16.2.4.30 g_ulUIUseOnboard [static]

Definition:

```
static unsigned long g_ulUIUseOnboard
```

Description:

A boolean that is true when the on-board user interface should be active and false when it should not be.

16.2.4.31 g_usCurrentFrequency

Definition:

```
unsigned short g_usCurrentFrequency
```

Description:

The current drive frequency. This is updated by the speed control routine as it ramps the speed of the motor drive.

16.2.4.32 g_usFirmwareVersion

Definition:

```
const unsigned short g_usFirmwareVersion
```

Description:

The version of the firmware. Changing this value will make it much more difficult for Texas Instruments support personnel to determine the firmware in use when trying to provide assistance; it should only be changed after careful consideration.

16.2.4.33 g_usTargetFrequency

Definition:

```
unsigned short g_usTargetFrequency
```

Description:

The target drive frequency.

17 V/f Control

17.1 Introduction

In order to maintain a fixed torque over the operating frequency of the motor, the voltage applied to the motor must be varied in proportion to the drive frequency. This module provides an adjustable V/f curve so that the torque can be held approximately constant across the operating frequency of any given motor.

The V/f curve consists of 21 points that provide the amplitude (effectively voltage) based on the drive frequency. The points are evenly spaced between 0 Hz and either 100 Hz or 400 Hz (based on a configuration value); this provides a point every 5 Hz or 20 Hz. For frequencies between those in the curve, linear interpolation is used to compute the amplitude.

The code for handling the V/f curve is contained in `vf.c`, with `vf.h` containing the definition for the function exported to the remainder of the application.

17.2 Definitions

Functions

- unsigned long VFGetAmplitude (unsigned long ulFrequency)

17.2.1 Function Documentation

17.2.1.1 VFGetAmplitude

Gets the amplitude based on the frequency.

Prototype:

```
unsigned long
VFGetAmplitude(unsigned long ulFrequency)
```

Parameters:

ulFrequency is the current motor frequency as a 16.16 fixed point value.

Description:

This function performs the V/f computation to convert a motor frequency into the amplitude of the waveform. A V/f table is used to define the mapping of frequency to amplitude; linear interpolation is utilized for frequencies that are not directly defined in the V/f table.

Returns:

The amplitude as a 16.16 fixed point value.

18 CPU Usage Module

18.1 Introduction

The CPU utilization module uses one of the system timers and peripheral clock gating to determine the percentage of the time that the processor is being clocked. For the most part, the processor is executing code whenever it is being clocked (exceptions occur when the clocking is being configured, which only happens at startup, and when entering/exiting an interrupt handler, when the processor is performing stacking operations on behalf of the application).

The specified timer is configured to run when the processor is in run mode and to not run when the processor is in sleep mode. Therefore, the timer will only count when the processor is being clocked. Comparing the number of clocks the timer counted during a fixed period to the number of clocks in the fixed period provides the percentage utilization.

In order for this to be effective, the application must put the processor to sleep when it has no work to do (instead of busy waiting). If the processor never goes to sleep (either because of a continual stream of work to do or a busy loop), the processor utilization will be reported as 100%.

Since deep-sleep mode changes the clocking of the system, the computed processor usage may be incorrect if deep-sleep mode is utilized. The number of clocks the processor spends in run mode will be properly counted, but the timing period may not be accurate (unless extraordinary measures are taken to ensure timing period accuracy).

The accuracy of the computed CPU utilization depends upon the regularity with which CPUUsageTick() is called by the application. If the CPU usage is constant, but CPUUsageTick() is called sporadically, the reported CPU usage will fluctuate as well despite the fact that the CPU usage is actually constant.

This module is contained in `utils/cpu_usage.c`, with `utils/cpu_usage.h` containing the API definitions for use by applications.

18.2 API Functions

Functions

- void CPUUsageInit (unsigned long ulClockRate, unsigned long ulRate, unsigned long ulTimer)
- unsigned long CPUUsageTick (void)

18.2.1 Function Documentation

18.2.1.1 CPUUsageInit

Initializes the CPU usage measurement module.

Prototype:

```
void
CPUUsageInit(unsigned long ulClockRate,
             unsigned long ulRate,
             unsigned long ulTimer)
```

Parameters:

ulClockRate is the rate of the clock supplied to the timer module.

ulRate is the number of times per second that CPUUsageTick() is called.

ulTimer is the index of the timer module to use.

Description:

This function prepares the CPU usage measurement module for measuring the CPU usage of the application.

Returns:

None.

18.2.1.2 CPUUsageTick

Updates the CPU usage for the new timing period.

Prototype:

```
unsigned long
CPUUsageTick(void)
```

Description:

This function, when called at the end of a timing period, will update the CPU usage.

Returns:

Returns the CPU usage percentage as a 16.16 fixed-point value.

18.3 Programming Example

The following example shows how to use the CPU usage module to measure the CPU usage where the foreground simply burns some cycles.

```
//
// The CPU usage for the most recent time period.
//
unsigned long g_ulCPUUsage;

//
// Handles the SysTick interrupt.
```

```
//
void
SysTickIntHandler(void)
{
    //
    // Compute the CPU usage for the last time period.
    //
    g_ulCPUUsage = CPUUsageTick();
}

//
// The main application.
//
int
main(void)
{
    //
    // Initialize the CPU usage module, using timer 0.
    //
    CPUUsageInit(8000000, 100, 0);

    //
    // Initialize SysTick to interrupt at 100 Hz.
    //
    SysTickPeriodSet(8000000 / 100);
    SysTickIntEnable();
    SysTickEnable();

    //
    // Loop forever.
    //
    while(1)
    {
        //
        // Delay for a little bit so that CPU usage is not zero.
        //
        SysCtlDelay(100);

        //
        // Put the processor to sleep.
        //
        SysCtlSleep();
    }
}
```

19 Flash Parameter Block Module

19.1 Introduction

The flash parameter block module provides a simple, fault-tolerant, persistent storage mechanism for storing parameter information for an application.

The FlashPBInit() function is used to initialize a parameter block. The primary conditions for the parameter block are that flash region used to store the parameter blocks must contain at least two erase blocks of flash to ensure fault tolerance, and the size of the parameter block must be an integral divisor of the the size of an erase block. FlashPBGet() and FlashPBSave() are used to read and write parameter block data into the parameter region. The only constraints on the content of the parameter block are that the first two bytes of the block are reserved for use by the read/write functions as a sequence number and checksum, respectively.

This module is contained in `utils/flash_pb.c`, with `utils/flash_pb.h` containing the API definitions for use by applications.

19.2 API Functions

Functions

- unsigned char * FlashPBGet (void)
- void FlashPBInit (unsigned long ulStart, unsigned long ulEnd, unsigned long ulSize)
- static unsigned long FlashPBIsValid (unsigned char *pucOffset)
- void FlashPBSave (unsigned char *pucBuffer)

19.2.1 Function Documentation

19.2.1.1 FlashPBGet

Gets the address of the most recent parameter block.

Prototype:

```
unsigned char *
FlashPBGet(void)
```

Description:

This function returns the address of the most recent parameter block that is stored in flash.

Returns:

Returns the address of the most recent parameter block, or NULL if there are no valid parameter blocks in flash.

19.2.1.2 FlashPBInit

Initializes the flash parameter block.

Prototype:

```
void
FlashPBInit(unsigned long ulStart,
            unsigned long ulEnd,
            unsigned long ulSize)
```

Parameters:

ulStart is the address of the flash memory to be used for storing flash parameter blocks; this must be the start of an erase block in the flash.

ulEnd is the address of the end of flash memory to be used for storing flash parameter blocks; this must be the start of an erase block in the flash (the first block that is NOT part of the flash memory to be used), or the address of the first word after the flash array if the last block of flash is to be used.

ulSize is the size of the parameter block when stored in flash; this must be a power of two less than or equal to the flash erase block size (typically 1024).

Description:

This function initializes a fault-tolerant, persistent storage mechanism for a parameter block for an application. The last several erase blocks of flash (as specified by *ulStart* and *ulEnd* are used for the storage; more than one erase block is required in order to be fault-tolerant.

A parameter block is an array of bytes that contain the persistent parameters for the application. The only special requirement for the parameter block is that the first byte is a sequence number (explained in FlashPBSave()) and the second byte is a checksum used to validate the correctness of the data (the checksum byte is the byte such that the sum of all bytes in the parameter block is zero).

The portion of flash for parameter block storage is split into N equal-sized regions, where each region is the size of a parameter block (*ulSize*). Each region is scanned to find the most recent valid parameter block. The region that has a valid checksum and has the highest sequence number (with special consideration given to wrapping back to zero) is considered to be the current parameter block.

In order to make this efficient and effective, three conditions must be met. The first is *ulStart* and *ulEnd* must be specified such that at least two erase blocks of flash are dedicated to parameter block storage. If not, fault tolerance can not be guaranteed since an erase of a single block will leave a window where there are no valid parameter blocks in flash. The second condition is that the size (*ulSize*) of the parameter block must be an integral divisor of the size of an erase block of flash. If not, a parameter block will end up spanning between two erase blocks of flash, making it more difficult to manage. The final condition is that the size of the flash dedicated to parameter blocks (*ulEnd* - *ulStart*) divided by the parameter block size (*ulSize*) must be less than or equal to 128. If not, it will not be possible in all cases to determine which parameter block is the most recent (specifically when dealing with the sequence number wrapping back to zero).

When the microcontroller is initially programmed, the flash blocks used for parameter block storage are left in an erased state.

This function must be called before any other flash parameter block functions are called.

Returns:

None.

19.2.1.3 FlashPBIsValid [static]

Determines if the parameter block at the given address is valid.

Prototype:

```
static unsigned long
FlashPBIsValid(unsigned char *pucOffset)
```

Parameters:

pucOffset is the address of the parameter block to check.

Description:

This function will compute the checksum of a parameter block in flash to determine if it is valid.

Returns:

Returns one if the parameter block is valid and zero if it is not.

19.2.1.4 FlashPBSave

Writes a new parameter block to flash.

Prototype:

```
void
FlashPBSave(unsigned char *pucBuffer)
```

Parameters:

pucBuffer is the address of the parameter block to be written to flash.

Description:

This function will write a parameter block to flash. Saving the new parameter blocks involves three steps:

- Setting the sequence number such that it is one greater than the sequence number of the latest parameter block in flash.
- Computing the checksum of the parameter block.
- Writing the parameter block into the storage immediately following the latest parameter block in flash; if that storage is at the start of an erase block, that block is erased first.

By this process, there is always a valid parameter block in flash. If power is lost while writing a new parameter block, the checksum will not match and the partially written parameter block will be ignored. This is what makes this fault-tolerant.

Another benefit of this scheme is that it provides wear leveling on the flash. Since multiple parameter blocks fit into each erase block of flash, and multiple erase blocks are used for parameter block storage, it takes quite a few parameter block saves before flash is re-written.

Returns:

None.

19.3 Programming Example

The following example shows how to use the flash parameter block module to read the contents of a flash parameter block.

```
unsigned char pucBuffer[16], *pucPB;

//
// Initialize the flash parameter block module, using the last two pages of
// a 64 KB device as the parameter block.
//
FlashPBInit(0xf800, 0x10000, 16);

//
// Read the current parameter block.
//
pucPB = FlashPBGet();
if(pucPB)
{
    memcpy(pucBuffer, pucPB);
}
```

20 Sine Calculation Module

20.1 Introduction

This module provides a fixed-point sine function. The input angle is a 0.32 fixed-point value that is the percentage of 360 degrees. This has two benefits; the sine function does not have to handle angles that are outside the range of 0 degrees through 360 degrees (in fact, 360 degrees can not be represented since it would wrap to 0 degrees), and the computation of the angle can be simplified since it does not have to deal with wrapping at values that are not natural for binary arithmetic (such as 360 degrees or 2π radians).

A sine table is used to find the approximate value for a given input angle. The table contains 128 entries that range from 0 degrees through 90 degrees and the symmetry of the sine function is used to determine the value between 90 degrees and 360 degrees. The maximum error caused by this table-based approach is 0.00618, which occurs near 0 and 180 degrees.

This module is contained in `utils/sine.c`, with `utils/sine.h` containing the API definitions for use by applications.

20.2 API Functions

Defines

- cosine(ulAngle)

Functions

- long sine (unsigned long ulAngle)

20.2.1 Define Documentation

20.2.1.1 cosine

Computes an approximation of the cosine of the input angle.

Definition:

```
#define cosine(ulAngle)
```

Parameters:

ulAngle is an angle expressed as a 0.32 fixed-point value that is the percentage of the way around a circle.

Description:
This function computes the cosine for the given input angle. The angle is specified in 0.32 fixed point format, and is therefore always between 0 and 360 degrees, inclusive of 0 and exclusive of 360.

Returns:
Returns the cosine of the angle, in 16.16 fixed point format.

20.2.2 Function Documentation

20.2.2.1 sine

Computes an approximation of the sine of the input angle.

Prototype:
```
long
sine(unsigned long ulAngle)
```

Parameters:
ulAngle is an angle expressed as a 0.32 fixed-point value that is the percentage of the way around a circle.

Description:
This function computes the sine for the given input angle. The angle is specified in 0.32 fixed point format, and is therefore always between 0 and 360 degrees, inclusive of 0 and exclusive of 360.

Returns:
Returns the sine of the angle, in 16.16 fixed point format.

20.3 Programming Example

The following example shows how to produce a sine wave with 7 degrees between successive values.

```
unsigned long ulValue;

//
// Produce a sine wave with each step being 7 degrees advanced from the
// previous.
//
for(ulValue = 0; ; ulValue += 0x04FA4FA4)
{
    //
    // Compute the sine at this angle and do something with the result.
    //
    sine(ulValue);
}
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
TI E2E Community Home Page	e2e.ti.com		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265