RDK-BLDC Firmware Development Package

USER'S GUIDE

SW-RDK-BLDC-UG-6075

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 6075 of this document, last updated on June 04, 2010.

Table of Contents

1 Introduction

1.1 Overview

This document describes the functions, variables, structures, and symbolic constants in the software for the Brushless DC motor RDK. The firmware runs on a Stellaris® LM3S8971 microcontroller, utilizing the provided six-channel motion control PWM module, eight-channel ADC, Ethernet, and quadrature encoder module. The Stellaris Peripheral Driver Library is used to configure and operate these peripherals.

The Brushless DC motor application has the following features:

- Operation from 0 to 60,000 RPM with 1 RPM steps
- Acceleration and deceleration from 1 to 50000 RPM/sec
- PWM frequency of 8 KHz, 12.5 KHz, 16 KHz, 20 KHz, 25 KHz, 40 KHz and 50 KHz.
- Closed-loop speed control
- Dynamic braking
- Real-time data monitoring
- Fault monitoring and handling
- Trapezoid, Sensorless, and Sine Wave Modulation

See the *Stellaris Brushless DC 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 55 KB of FLASH and 21 KB of SRAM. Of this, approximately 35 KB of flash and 17 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 approximately 20 KB of flash and 4 KB of SRAM for the actual motor drive application, parts of which are also more complicated than required due to the run-time reconfigurability.

A variation of the BLDC application has also been provided that removes the GUI support and uses compile-time options to control the motor drive configuration. The configuration that is provided

supports the sensored BLDC motor that is provided in the RDK. This application uses about 12K of FLASH and about 5K of SRAM.

1.3 Processor Usage

The factors that have the largest impact on the processor usage are the PWM frequency and Modulation type. 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). These numbers were measured while running a single pole-pair motor at 5000 RPM.

Modulation	PWM Frequency						
	8 KHz	12.5 KHz	16 KHz	20 KHz	25 KHz	40 KHz	50 KHz
Trapezoid	13%	18%	21%	25%	30%	45%	54%
Sensorless	15%	20%	24%	28%	34%	51%	NA
Sinusoid	22%	33%	41%	50%	NA	NA	NA

1.4 Memory Layout

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

In addition to the boot loader and the Brushless DC 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 256 KB of flash on the LM3S8971 is organized as follows:

0x0000.0000 0x0000.0fff	Boot Loader
0x0000.1000 0x0003.efff	BLDC Firmware
0x0003.f000 0x0003.ffff	Parameter Storage

1.5 Debugging

The Brushless DC 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 difference between rotor and target speed will have increased (possibly significantly). This likely results in a motor over current fault, resulting in the motor drive being shut off, meaning that the debug event has caused catastrophic results for 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; to abuse the real-time data stream, 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. While this is not an ideal debug, it is better than nothing.

With the BLDC RDK using ethernet, an additional debugging option is available. The BLDC RDK software defines an array of 8 unsigned longs as debug information (g_ulDebugInfo[]). This array is included at the end of the real-time data stream. With an ethernet packet monitor (such as Wireshark), the packets containing the real-time data can be monitored for debug information content.

1.6 Porting

The BLDC software has been designed to allow easy porting to a different target environment. In a typical development cycle, the BLDK RDK would be used initially with the provided motor to gain familiarity with the operation of the BLDC motor drive software and the affect that changing parameters has on the operation of the motor. After that, the customer’s motor can be connected to the RDK (assuming that power load is not exceeded) and tested. Fine-tuning the motor drive parameters for a specific motor is an iterative process, and can be quite time consuming (especially in Sensorless mode). Once the parameter values for the motor drive have been determined, these values can be hard-coded into the basic-bldc application and compiled. This application can then be programmed into the BLDC RDK and used to control the motor in a more optimized environment.

Note:

This application is configured to be a stand-alone application. It will not function with the BLDC GUI, and the BLDC GUI cannot be used to program the application into the board. This application will replace both the boot loader firmware (located at address 0) and the quick start firmware (located at address 0x1000).

This application provides minimal user interface support. Currently, only the push-button switch is used to start/stop the motor. The analog input is availble to be used for control (e.g. speed) but has not been programmed for this operation.

Once the basic-bldc firmware is running successfully on the BLDC RDK board, it can be ported to a custom board by modifying the appropriate hardware definitions. Many of these are contained in the file, "pins.h". However, additional hardware defintions may be found in "ui.c", "main.c", and "hall_ctrl.c".

2 Applications

The boot loader (boot_serial) and quickstart (qs-bldc) are programmed onto the MDL-BLDC. The basic application (basic-bldc) can be used in place of the quickstart as a less feature-rich alternative. The boot loader can be used to update the application using the Ethernet port, eliminating the need for a JTAG debugger.

There is an IAR workspace file (`rdk-bldc.eww`) that contains the peripheral driver library project, along with the Brushless DC 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-bldc.mpw`) that contains the peripheral driver library project, along with the Brushless DC Motor Controller software project, in a single, easy to use workspace for use with uVision.

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

2.0.1 Detailed Description

2.1 Basic Brushless DC Motor Drive Application (basic-bldc)

This application is a motor drive for Brushless DC (BLDC) motors. The following features are supported:

- Trapezoid, Sensorless, or Sinusoid modulation
- Closed loop speed control
- DC bus voltage monitoring and control
- Regenerative braking control
- Simple on-board user interface (via a push button)
- Over 25 configurable drive parameters

2.2 Boot Loader (boot_eth)

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 Ethernet to load an application.

2.3 Brushless DC Motor Drive Application (qs-bldc)

This application is a motor drive for Brushless DC (BLDC) motors. The following features are supported:

- Trapezoid, Sensorless, and Sinusoid modulation
- Closed loop speed control
- DC bus voltage monitoring and control
- Regenerative braking control
- Simple on-board user interface (via a push button)
- Comprehensive serial user interface via ethernet port
- Over 25 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.

Ethernet Flash Downloader

Usage:

```
eflash [OPTION]... [INPUT FILE]
```

Description:

Downloads a firmware image to a Stellaris board using an Ethernet connection to the Stellaris Boot Loader. This has the same capabilities as the Ethernet download portion of the Stellaris Flash Programmer.

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

Arguments:

--help displays usage information.

-h is an alias for `--help`.

--ip=IP specifies the IP address to be provided by the BOOTP server.

-i IP is an alias for `--ip`.

--mac=MAC specifies the MAC address

-m MAC is an alias for `--mac`.

--quiet specifies that only error information should be output.

--silent is an alias for `--quiet`.

--verbose specifies that verbose output should be output.

--version displays the version of the utility and exits.

INPUT FILE specifies the name of the firmware image file.

Example:

The following will download a firmware image to the board over Ethernet, where the target board has a MAC address of 00:11:22:33:44:55 and is given an IP address of 169.254.19.70:

```
eflash -m 00:11:22:33:44:55 -i 169.254.19.70 image.bin
```

4 ADC Control

4.1 Introduction

Depending on the mode of operation, the ADC is used to monitor the motor phase current, motor phase back EMF voltage, linear hall sensor voltage, DC bus voltage, analog input voltage, and ambient temperature of the microcontroller. Each of these values is sampled every PWM period based on a trigger from the PWM module. Multiple ADC sequences are used to allow optimization of CPU usage.

Readings from the ADC may be 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. For example, a coefficient of 0.75 is sometimes used to simplify the integer math (requiring only a multiplication by three, an addition, and a division by four).

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

Defines

- FLAG_BEMF_EDGE_BIT
- FLAG_EDGE_BIT
- FLAG_SKIP_BIT
- FLAG_SKIP_LINEAR_BIT

Functions

- void ADC0IntHandler (void)
- static void ADC0IntIdle (void)
- static void ADC0IntSine (void)
- static void ADC0IntSineLinear (void)
- static void ADC0IntTrap (void)
- static void ADC0IntTrapLinear (void)
- void ADCConfigure (void)
- void ADCInit (void)
- unsigned long ADCReadAnalog (void)
- void ADCTickHandler (void)

Variables

- static void(* g_pfnADC0Handler)(void)
- short g_psPhaseCurrent[3]
- static unsigned short g_pusADC0DataRaw[8]
- static unsigned short g_pusBEMFVoltageCount[3]
- static unsigned short g_pusLinearHallMax[3]
- static unsigned short g_pusLinearHallMin[3]
- static unsigned short g_pusLinearHallSensor[3]
- static unsigned short g_pusPhaseCurrentCount[3]
- static unsigned short g_pusPhaseMax[3]
- short g_sAmbientTemp
- short g_sMotorCurrent
- static unsigned char g_ucBEMFState
- static unsigned char g_ucPhaseCurrentIndex
- static unsigned char g_ucPreviousPhaseCurrentIndex
- static unsigned long g_ulADC0Time
- static unsigned long g_ulADCFlags
- static unsigned long g_ulBEMFEdgePrevious
- unsigned long g_ulBEMFHallValue
- unsigned long g_ulBEMFNextHall
- static unsigned long g_ulBEMFPeriod
- unsigned long g_ulBEMFRotorSpeed
- static unsigned long g_ulBEMFSpeedPrevious
- unsigned long g_ulBusVoltage
- unsigned long g_ulLinearHallValue
- static unsigned long g_ulLinearLastHall
- unsigned long g_ulLinearRotorSpeed
- static unsigned long g_ulLinearSpeedPrevious
- unsigned long g_ulMotorPower
- static unsigned long g_ulPhaseBEMFCountMax
- static unsigned long g_ulPhaseBEMFCountMin
- unsigned long g_ulPhaseBEMFVoltage
- static unsigned long g_ulPrevAngle
- static unsigned short g_usAmbientTempCount
- static unsigned short g_usBusVoltageCount

4.2.1 Define Documentation

4.2.1.1 FLAG_BEMF_EDGE_BIT

Definition:

```
#define FLAG_BEMF_EDGE_BIT
```

Description:

The bit number of the flag in g_ulADCFlags that indicates that a zero-crossing has been detected in the Back EMF processing.

4.2.1.2 FLAG_EDGE_BIT

Definition:

```
#define FLAG_EDGE_BIT
```

Description:

The bit number of the flag in g_ulADCFlags that indicates that an edge has been seen (in the speed processing code). This will prevent the ADCTickHandler from resetting the rotor speed.

4.2.1.3 FLAG_SKIP_BIT

Definition:

```
#define FLAG_SKIP_BIT
```

Description:

The bit number of the flag in g_ulADCFlags that indicates that the next edge should be ignored by the Back EMF speed calculation code. This is used at startup since there is no previous edge time to be used to calculate the time between edges.

4.2.1.4 FLAG_SKIP_LINEAR_BIT

Definition:

```
#define FLAG_SKIP_LINEAR_BIT
```

Description:

The bit number of the flag in g_ulADCFlags that indicates that the next edge should be ignored by the Linear Hall speed calculation code. This is used at startup since there is no previous edge time to be used to calculate the time between edges.

4.2.2 Function Documentation

4.2.2.1 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 any sequence overflow conditions. Then, depending on the modulation scheme that is active, the appropriate sub-handler is called.

Returns:

None.

4.2.2.2 ADC0IntIdle [static]

Handles the ADC sample sequence for idle (default) mode.

Prototype:

```
static void
ADC0IntIdle(void)
```

Description:

This functions processes the ADC sequence zero for the idle mode. The sequence has been programmed to read a bus voltage and ambient temperature.

Returns:

None.

4.2.2.3 ADC0IntSine [static]

Handles the ADC sample sequence for sinusoid mode.

Prototype:

```
static void
ADC0IntSine(void)
```

Description:

This functions processes the ADC sequence zero for trapezoid mode. The sequence has been programmed to read a single value of Phase current, along with bus voltage and ambient temperature.

Returns:

None.

4.2.2.4 ADC0IntSineLinear [static]

Handles the ADC sample sequence for sinusoid mode.

Prototype:

```
static void
ADC0IntSineLinear(void)
```

Description:

This functions processes the ADC sequence zero for trapezoid mode. The sequence has been programmed to read a single value of Phase current, along with bus voltage and ambient temperature.

Returns:

None.

4.2.2.5 ADC0IntTrap [static]

Handles the ADC sample sequence for trapezoid mode.

Prototype:

```
static void
ADC0IntTrap(void)
```

Description:

This functions processes the ADC sequence zero for trapezoid mode. The sequence has been programmed to read a single value of Phase current, along with bus voltage and ambient temperature.

Returns:

None.

4.2.2.6 ADC0IntTrapLinear [static]

Handles the ADC sample sequence for trapezoid mode, with linear hall sensors.

Prototype:

```
static void
ADC0IntTrapLinear(void)
```

Description:

This functions processes the ADC sequence zero for trapezoid mode. The sequence has been programmed to read a single value of Phase current, the linear hall sensor inputs, the bus voltage and ambient temperature.

Returns:

None.

4.2.2.7 ADCConfigure

Configure the ADC sequence based on the ADC mode of operation.

Prototype:

```
void
ADCConfigure(void)
```

Description:

This will parse set the ADC mode of operation (based on motor drive parameters) and will reconfigure the ADC sequences accordingly.

Returns:

None.

4.2.2.8 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.

4.2.2.9 ADCReadAnalog

Read the Analog input ADC value.

Prototype:

```
unsigned long
ADCReadAnalog(void)
```

Description:

This function will read the analog input value from sequence three, and retrigger the sequence for the next read.

Returns:

The ADC count value for the analog input if one is available, otherwise, all ones.

4.2.2.10 ADCTickHandler

Handles the ADC System Tick.

Prototype:

```
void
ADCTickHandler(void)
```

Description:

This function is called by the system tick handler. It's primary purpose is to reset the motor speed to 0 if no "Hall" edges have been detected for some period of time.

Returns:

None.

4.2.3 Variable Documentation

4.2.3.1 g_pfnADC0Handler [static]

Definition:

```
static void( *g_pfnADC0Handler )(void)
```

Description:
The interrupt handler for the ADC0 mode.

4.2.3.2 g_psPhaseCurrent

Definition:
```
short g_psPhaseCurrent[3]
```

Description:
The current passing through the three phases of the motor, specified in milliamperes as a signed value.

4.2.3.3 g_pusADC0DataRaw [static]

Definition:
```
static unsigned short g_pusADC0DataRaw[8]
```

Description:
This array contains the raw data read from the ADC sequence (prior to any filtering that may be applied).

4.2.3.4 g_pusBEMFVoltageCount [static]

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

Description:
The Back EMF Voltage ADC count value.

4.2.3.5 g_pusLinearHallMax [static]

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

Description:
The Linear Hall Sensor ADC Maximum Value.

4.2.3.6 g_pusLinearHallMin [static]

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

Description:
The Linear Hall Sensor ADC Minimum Value.

4.2.3.7 g_pusLinearHallSensor [static]

Definition:

```
static unsigned short g_pusLinearHallSensor[3]
```

Description:

The Linear Hall Sensor ADC values (scaled).

4.2.3.8 g_pusPhaseCurrentCount [static]

Definition:

```
static unsigned short g_pusPhaseCurrentCount[3]
```

Description:

The Phase Current ADC count value.

4.2.3.9 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.

4.2.3.10 g_sAmbientTemp

Definition:

```
short g_sAmbientTemp
```

Description:

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

4.2.3.11 g_sMotorCurrent

Definition:

```
short g_sMotorCurrent
```

Description:

The total current passing through the motor, specified in milliamperes as a signed value.

4.2.3.12 g_ucBEMFState [static]

Definition:

```
static unsigned char g_ucBEMFState
```

Description:
The state of the Back EMF processing state machine.

4.2.3.13 g_ucPhaseCurrentIndex [static]

Definition:
```
static unsigned char g_ucPhaseCurrentIndex
```

Description:
The index for the phase current being processed in the ADC sequence handler.

4.2.3.14 g_ucPreviousPhaseCurrentIndex [static]

Definition:
```
static unsigned char g_ucPreviousPhaseCurrentIndex
```

Description:
The index for the phase current previously processed in the ADC sequence handler.

4.2.3.15 g_ulADC0Time [static]

Definition:
```
static unsigned long g_ulADC0Time
```

Description:
The time of the ADC Sequence 0 Interrupt.

4.2.3.16 g_ulADCFlags [static]

Definition:
```
static unsigned long g_ulADCFlags
```

Description:
A set of flags that provide status and control of the ADC Control module.

4.2.3.17 g_ulBEMFEdgePrevious [static]

Definition:
```
static unsigned long g_ulBEMFEdgePrevious
```

Description:
The time at which the last Back EMF edge occurred.

4.2.3.18 g_ulBEMFHallValue

Definition:

```
unsigned long g_ulBEMFHallValue
```

Description:

The Hall state value as determined by the Back EMF processing code.

4.2.3.19 g_ulBEMFNextHall

Definition:

```
unsigned long g_ulBEMFNextHall
```

Description:

The next Back EMF Hall state value.

4.2.3.20 g_ulBEMFPeriod [static]

Definition:

```
static unsigned long g_ulBEMFPeriod
```

Description:

The average commuation period for Sensorless operation.

4.2.3.21 g_ulBEMFRotorSpeed

Definition:

```
unsigned long g_ulBEMFRotorSpeed
```

Description:

The rotor speed as measured by the BEMF processing code.

4.2.3.22 g_ulBEMFSpeedPrevious [static]

Definition:

```
static unsigned long g_ulBEMFSpeedPrevious
```

Description:

The time at which the last Back EMF speed edge occurred.

4.2.3.23 g_ulBusVoltage

Definition:

```
unsigned long g_ulBusVoltage
```

Description:

The DC bus voltage, specified in millivolts.

4.2.3.24 g_ulLinearHallValue

Definition:

```
unsigned long g_ulLinearHallValue
```

Description:

The Hall state value as determined by the linear Hall sensor processing code.

4.2.3.25 g_ulLinearLastHall [static]

Definition:

```
static unsigned long g_ulLinearLastHall
```

Description:

The previous Hall state value as determined by the linear Hall sensor processing code.

4.2.3.26 g_ulLinearRotorSpeed

Definition:

```
unsigned long g_ulLinearRotorSpeed
```

Description:

The rotor speed as measured by the linear Hall sensor processing code.

4.2.3.27 g_ulLinearSpeedPrevious [static]

Definition:

```
static unsigned long g_ulLinearSpeedPrevious
```

Description:

The time at which the last linear Hall sensor speed edge occurred.

4.2.3.28 g_ulMotorPower

Definition:

```
unsigned long g_ulMotorPower
```

Description:

The average motor power, specified in milliwatts.

4.2.3.29 g_ulPhaseBEMFCountMax [static]

Definition:

```
static unsigned long g_ulPhaseBEMFCountMax
```

Description:

The maximum Back EMF ADC reading.

4.2.3.30 g_ulPhaseBEMFCountMin [static]

Definition:

```
static unsigned long g_ulPhaseBEMFCountMin
```

Description:

The minimum Back EMF ADC reading.

4.2.3.31 g_ulPhaseBEMFVoltage

Definition:

```
unsigned long g_ulPhaseBEMFVoltage
```

Description:

The Phase Back EMF voltage, specified in millivolts.

4.2.3.32 g_ulPrevAngle [static]

Definition:

```
static unsigned long g_ulPrevAngle
```

Description:

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

4.2.3.33 g_usAmbientTempCount [static]

Definition:

```
static unsigned short g_usAmbientTempCount
```

Description:

The Ambient Temperature ADC count value. This value is passed through an IIR filter with a coefficient of .875.

4.2.3.34 g_usBusVoltageCount [static]

Definition:

```
static unsigned short g_usBusVoltageCount
```

Description:

The Bus Voltage ADC count value. This value is passed through an IIR filter with a coefficient of .875.

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.

- 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 be measured.

- Motor Stall: This occurs when the motor is running, and the speed is detected as zero for at least 1.5 seconds. This would typically occur due to some type of mechanical interference to the operation of the motor shaft.

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

6.2 Definitions

Defines

- FAULT_CURRENT_HIGH
- FAULT_CURRENT_LOW
- FAULT_EMERGENCY_STOP
- FAULT_STALL
- FAULT_TEMPERATURE_HIGH
- FAULT_VBUS_HIGH
- FAULT_VBUS_LOW
- FAULT_WATCHDOG

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_STALL

Definition:

```
#define FAULT_STALL
```

Description:

The fault flag that indicates that the motor drive has stalled.

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.

6.2.1.8 FAULT_WATCHDOG

Definition:

```
#define FAULT_WATCHDOG
```

Description:

The fault flag that indicates that the watchdog timer expired.

7 Hall Sensor Control

7.1 Introduction

Brushless DC motors may be configured with Hall sensors. These sensors are used to determine motor speed and position.

In this module, the Hall sensor input edges are monitored to determine the current Hall state value (position), and to determine motor speed.

The Hall sensor inputs should be connected to GPIO inputs on the BLDC RDK input connected (Hall A, B, and C). These inputs are configured as GPIO inputs, and configured to generate interrupts on both rising and falling edges.

The Hall state value is stored at each interrupt. The time between the interrupt edges is measured to determine the speed of the motor.

The code for calculating the motor speed and updating the Hall state value is contained in `hall_ctrl.c`, with `hall_ctrl.h` containing the definitions for the variable and functions exported to the remainder of the application.

Note:
If the Hall sensors are configured as Linear Hall sensors, refer to the code in `adc_ctrl.c` for details about the processing of linear Hall sensor input data.

7.2 Definitions

Functions

- void GPIOBIntHandler (void)
- void HallConfigure (void)
- void HallInit (void)
- static void HallSpeedNewValue (unsigned long ulNewSpeed)
- void HallTickHandler (void)

Variables

- static unsigned char g_ucSkipFlag
- unsigned long g_ulHallRotorSpeed
- unsigned long g_ulHallValue
- static unsigned long g_ulOldTime[8]

7.2.1 Function Documentation

7.2.1.1 GPIOBIntHandler

Handles the GPIO port B interrupt.

Prototype:

```
void
GPIOBIntHandler(void)
```

Description:

This function is called when GPIO port B asserts its interrupt. GPIO port B is configured to generate an interrupt on both the rising and falling edges of the Hall sensor input signals.

Returns:

None.

7.2.1.2 HallConfigure

Configure the Hall sensor control routines, based on motor drive parameters.

Prototype:

```
void
HallConfigure(void)
```

Description:

This function will configure the Hall sensor routines, mainly enable/disable the Hall interrupt based on the motor drive configuration.

Returns:

None.

7.2.1.3 HallInit

Initializes the Hall sensor control routines.

Prototype:

```
void
HallInit(void)
```

Description:

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

Returns:

None.

7.2.1.4 HallSpeedNewValue [static]

Updates the current rotor speed.

Prototype:

```
static void
HallSpeedNewValue(unsigned long ulNewSpeed)
```

Parameters:

ulNewSpeed is the newly measured speed.

Description:

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

Returns:

None.

7.2.1.5 HallTickHandler

Handles the Hall System Tick.

Prototype:

```
void
HallTickHandler(void)
```

Description:

This function is called by the system tick handler. It's primary purpose is to reset the motor speed to 0 if no Hall interrupt edges have been detected for some period of time.

Returns:

None.

7.2.2 Variable Documentation

7.2.2.1 g_ucSkipFlag [static]

Definition:

```
static unsigned char g_ucSkipFlag
```

Description:

A bit-mapped flag of Hall edges to skip before starting a speed calculations for a given Hall edge.

7.2.2.2 g_ulHallRotorSpeed

Definition:

```
unsigned long g_ulHallRotorSpeed
```

Description:

The current speed of the motor's rotor.

7.2.2.3 g_ulHallValue

Definition:

```
unsigned long g_ulHallValue
```

Description:

The current Hall Sensor value.

7.2.2.4 g_ulOldTime [static]

Definition:

```
static unsigned long g_ulOldTime[8]
```

Description:

This is the time at which the previous edge was seen and is used to determine the time between edges.

8 Main Application

8.1 Introduction

This is the main Brushless DC 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 speed 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 new waveform values are passed to the PWM module to be supplied to the PWM hardware at the correct time.

The millisecond speed update interrupt handler is responsible for handling the dynamic brake, computing the new drive speed, 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 speed. First, the acceleration or deceleration rate is applied as appropriate to move the drive speed towards the target speed. Also, the amplitude of the PWM outputs is adjusted by a PI controller, moving the rotor speed to the desired speed. In the case of deceleration, the deceleration rate may be reduced based on the DC bus voltage. The result of this speed 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 speed update interrupt handler and the routines that start, stop, and adjust the parameters of the motor drive. Together, they ensure 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_SIZE
- FLASH_PB_START
- PWM_CLOCK
- PWM_CLOCK_WIDTH
- STATE_BACK_PRECHARGE
- STATE_BACK_REV
- STATE_BACK_RUN
- STATE_BACK_STARTUP
- STATE_BACK_STOPPING
- STATE_FLAG_BACKWARD
- STATE_FLAG_FORWARD
- STATE_FLAG_PRECHARGE
- STATE_FLAG_REV
- STATE_FLAG_RUN
- STATE_FLAG_STARTUP
- STATE_FLAG_STOPPING
- STATE_PRECHARGE
- STATE_REV
- STATE_RUN
- STATE_STARTUP
- STATE_STOPPED
- STATE_STOPPING
- SYSTEM_CLOCK
- SYSTEM_CLOCK_WIDTH
- WATCHDOG_RELOAD_VALUE

Functions

- int main (void)
- static void MainCheckFaults (void)
- void MainClearFaults (void)
- void MainEmergencyStop (void)
- unsigned long MainIsFaulted (void)
- unsigned long MainIsReverse (void)
- unsigned long MainIsRunning (void)
- unsigned long MainIsStartup (void)
- static long MainLongMul (long lX, long lY)
- void MainMillisecondTick (void)
- unsigned long MainPowerController (void)

- static void MainPowerHandler (unsigned long ulTarget)
- static void MainPrechargeHandler (void)
- void MainPunchWatchdog (void)
- void MainRun (void)
- void MainSetDirection (tBoolean bForward)
- void MainSetFault (unsigned long ulFaultFlag)
- void MainSetPower (void)
- void MainSetPWMFrequency (void)
- void MainSetSpeed (void)
- unsigned long MainSpeedController (void)
- static void MainSpeedHandler (unsigned long ulTarget)
- static void MainStartupHandler (void)
- void MainStop (void)
- void MainUpdateFAdjI (long lNewFAdjI)
- void MainUpdatePAdjI (long lNewPAdjI)
- void MainUpgrade (void)
- void MainWaveformTick (void)
- void Timer0AIntHandler (void)

Variables

- static volatile tBoolean g_bStartBootloader
- static long g_lPowerIntegrator
- static long g_lPowerIntegratorMax
- static long g_lSpeedIntegrator
- static long g_lSpeedIntegratorMax
- static unsigned char g_ucLocalDecayMode
- unsigned char g_ucMotorStatus
- static unsigned char g_ucStartupHallIndex
- static unsigned long g_ulAccelRate
- unsigned long g_ulAngle
- static unsigned long g_ulAngleDelta
- static unsigned long g_ulDecelRate
- static unsigned long g_ulDutyCycle
- unsigned long g_ulFaultFlags
- static unsigned long g_ulHallPrevious
- unsigned long g_ulMeasuredSpeed
- static unsigned long g_ulPower
- static unsigned long g_ulPowerFract
- static unsigned long g_ulPowerWhole
- static unsigned long g_ulSineTarget
- static unsigned long g_ulSpeed
- static unsigned long g_ulSpeedFract
- static unsigned long g_ulSpeedWhole
- static unsigned long g_ulStartupDutyCycle

- static unsigned long g_ulStartupDutyCycleRamp
- static unsigned long g_ulStartupPeriod
- static unsigned long g_ulStartupState
- static unsigned long g_ulState
- static unsigned long g_ulStateCount

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_SIZE

Definition:

```
#define FLASH_PB_SIZE
```

Description:

The size of the parameter block to save. This must be a power of 2, and should be large enough to contain the tDriveParameters structure (see the ui.h file).

8.2.1.4 FLASH_PB_START

Definition:

```
#define FLASH_PB_START
```

Description:

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

8.2.1.5 PWM_CLOCK

Definition:

```
#define PWM_CLOCK
```

Description:

The frequency of the clock that drives the PWM generators.

8.2.1.6 PWM_CLOCK_WIDTH

Definition:

```
#define PWM_CLOCK_WIDTH
```

Description:

The width of a single PWM clock, in nanoseconds.

8.2.1.7 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.

8.2.1.8 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.

8.2.1.9 STATE_BACK_RUN

Definition:

```
#define STATE_BACK_RUN
```

Description:

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

8.2.1.10 STATE_BACK_STARTUP

Definition:

```
#define STATE_BACK_STARTUP
```

Description:

The motor drive is starting. The motor will be spun in the backward direction until a minimum speed is reached. At that point, the motor state will be transitioned to STATE_BACK_RUN.

8.2.1.11 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.

8.2.1.12 STATE_FLAG_BACKWARD

Definition:

```
#define STATE_FLAG_BACKWARD
```

Description:

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

8.2.1.13 STATE_FLAG_FORWARD

Definition:

```
#define STATE_FLAG_FORWARD
```

Description:

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

8.2.1.14 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.

8.2.1.15 STATE_FLAG_REV

Definition:

```
#define STATE_FLAG_REV
```

Description:

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

8.2.1.16 STATE_FLAG_RUN

Definition:

```
#define STATE_FLAG_RUN
```

Description:

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

8.2.1.17 STATE_FLAG_STARTUP

Definition:

```
#define STATE_FLAG_STARTUP
```

Description:

A state flag that indicates that the motor drive is in the startup condition, getting the motor spinning for sensorless opertation.

8.2.1.18 STATE_FLAG_STOPPING

Definition:

```
#define STATE_FLAG_STOPPING
```

Description:

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

8.2.1.19 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.

8.2.1.20 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.

8.2.1.21 STATE_RUN

Definition:

```
#define STATE_RUN
```

Description:

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

8.2.1.22 STATE_STARTUP

Definition:

```
#define STATE_STARTUP
```

Description:

The motor drive is starting. The motor will be spun in the forward direction until a minimum speed is reached. At that point, the motor state will be transitioned to STATE_RUN.

8.2.1.23 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.

8.2.1.24 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.

8.2.1.25 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().

8.2.1.26 SYSTEM_CLOCK_WIDTH

Definition:

```
#define SYSTEM_CLOCK_WIDTH
```

Description:

The width of a single system clock, in ns.

8.2.1.27 WATCHDOG_RELOAD_VALUE

Definition:

```
#define WATCHDOG_RELOAD_VALUE
```

Description:

The watchdog timer reload value.

8.2.2 Function Documentation

8.2.2.1 main

Handles setup of the application on the Brushless DC motor drive.

Prototype:

```
int
main(void)
```

Description:

This is the main application entry point for the Brushless DC 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 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 speed at zero, and the emergency stop fault condition will be asserted.

Returns:

None.

8.2.2.5 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.

8.2.2.6 MainIsReverse

Determines if the motor drive is currently in reverse mode.

Prototype:
```
unsigned long
MainIsReverse(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 not in reverse and 1 if it is in reverse.

8.2.2.7 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.

8.2.2.8 MainIsStartup

Determines if the motor drive is in precharge/startup.

Prototype:
```
unsigned long
MainIsStartup(void)
```

Description:
This function will determine if the motor drive is currently in startup. (or precharge) mode.

Returns:
Returns 0 if the motor drive is not in startup and 1 if it is running in startup (or precharge).

8.2.2.9 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.

8.2.2.10 MainMillisecondTick

Handles the millisecond speed update software interrupt.

Prototype:

```
void
MainMillisecondTick(void)
```

Description:

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

The speed 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.

8.2.2.11 MainPowerController

Adjusts the motor drive duty cycle based on the rotor speed.

Prototype:

```
unsigned long
MainPowerController(void)
```

Description:
This function uses a PI controller to adjust the motor drive duty cycle in order to get the rotor speed to match the target speed.

Returns:
Returns the new motor drive duty cycle.

8.2.2.12 MainPowerHandler [static]

Adjusts the motor drive power based on the target power.

Prototype:
```
static void
MainPowerHandler(unsigned long ulTarget)
```

Parameters:
ulTarget is the target power of the motor drive, specified as mW.

Description:
This function adjusts the motor drive power toward a given target power. 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.

8.2.2.13 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.

8.2.2.14 MainPunchWatchdog

Reset the watchdog timer

Prototype:
```
void
MainPunchWatchdog(void)
```

Description:
This function will reset the watchdog timer based on the current speed of the motor.

Returns:
None.

8.2.2.15 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 speed.

Returns:
None.

8.2.2.16 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.

8.2.2.17 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.

8.2.2.18 MainSetPower

Changes the target power of the motor drive.

Prototype:
```
void
MainSetPower(void)
```

Description:
This function changes the target power 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 power.

Note:
This function is not yet implemented/used.

Returns:
None.

8.2.2.19 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.

8.2.2.20 MainSetSpeed

Changes the target speed of the motor drive.

Prototype:
```
void
MainSetSpeed(void)
```

Description:
This function changes the target speed 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 speed.

Returns:
None.

8.2.2.21 MainSpeedController

Adjusts the motor drive duty cycle based on the rotor speed.

Prototype:
```
unsigned long
MainSpeedController(void)
```

Description:
This function uses a PI controller to adjust the motor drive duty cycle in order to get the rotor speed to match the target speed.

Returns:
Returns the new motor drive duty cycle.

8.2.2.22 MainSpeedHandler [static]

Adjusts the motor drive speed based on the target speed.

Prototype:
```
static void
MainSpeedHandler(unsigned long ulTarget)
```

Parameters:
ulTarget is the target speed of the motor drive, specified as RPM.

Description:
This function adjusts the motor drive speed towards a given target speed. 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.

8.2.2.23 MainStartupHandler [static]

Handles the startup mode of the motor drive.

Prototype:
```
static void
MainStartupHandler(void)
```

Description:
This function performs the processing and state transitions associated with the startup mode of the motor drive for sensorless operation.

Returns:
None.

8.2.2.24 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.

8.2.2.25 MainUpdateFAdjI

Updates the I coefficient of the speed 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 duty cycle 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.

8.2.2.26 MainUpdatePAdjI

Updates the I coefficient of the power PI controller.

Prototype:
```
void
MainUpdatePAdjI(long lNewPAdjI)
```

Parameters:

lNewPAdjI is the new value of the I coefficient.

Description:

This function updates the value of the I coefficient of the duty cycle 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.

8.2.2.27 MainUpgrade

Setup the main control loop for a firmware upgrade.

Prototype:

```
void
MainUpgrade(void)
```

Description:

This function will start a timer for initiating the firmware update.

Returns:

None.

8.2.2.28 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.

8.2.2.29 Timer0AIntHandler

Handles the Back EMF Timer Interrupt.

Prototype:

```
void
Timer0AIntHandler(void)
```

Description:

This function is called when the Back EMF timer expires. This code will set the Back EMF Hall state value to the next value, as determined by the Back EMF processing code. If the motor is running in the startup state, then open-loop commutation of the motor is performed by indexing through a predefined hall sequence. The timer is then restarted based on the period calculated in the startup state machine. If the motor is running in the normal run state, then the motor is commutated based on the values calculated in the Back EMF processing code. The timer will be restarted when the next zero-crossing event has been detected.

Returns:

None.

8.2.3 Variable Documentation

8.2.3.1 g_bStartBootloader [static]

Definition:

```
static volatile tBoolean g_bStartBootloader
```

Description:

The flag used to initiate the boot loader code.

8.2.3.2 g_lPowerIntegrator [static]

Definition:

```
static long g_lPowerIntegrator
```

Description:

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

8.2.3.3 g_lPowerIntegratorMax [static]

Definition:

```
static long g_lPowerIntegratorMax
```

Description:

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

8.2.3.4 g_lSpeedIntegrator [static]

Definition:

```
static long g_lSpeedIntegrator
```

Description:

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

8.2.3.5 g_lSpeedIntegratorMax [static]

Definition:

```
static long g_lSpeedIntegratorMax
```

Description:

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

8.2.3.6 g_ucLocalDecayMode [static]

Definition:

```
static unsigned char g_ucLocalDecayMode
```

Description:

A local variable used to save the state of the PWM decay mode during sensorless startup.

8.2.3.7 g_ucMotorStatus

Definition:

```
unsigned char g_ucMotorStatus
```

Description:

The current operation state of the motor drive.

8.2.3.8 g_ucStartupHallIndex [static]

Definition:

```
static unsigned char g_ucStartupHallIndex
```

Description:

The current index for the Startup hall commutation sequence.

8.2.3.9 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.

8.2.3.10 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.

8.2.3.11 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.

8.2.3.12 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.

8.2.3.13 g_ulDutyCycle [static]

Definition:

```
static unsigned long g_ulDutyCycle
```

Description:

The current duty cycle for the motor drive, expressed as a 16.16 fixed-point value in the range from 0.0 to 1.0.

8.2.3.14 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, and FAULT_TEMPERATURE_HIGH.

8.2.3.15 g_ulHallPrevious [static]

Definition:

```
static unsigned long g_ulHallPrevious
```

Description:

The previous value of the "Hall" state for trapezoid modulation.

8.2.3.16 g_ulMeasuredSpeed

Definition:

```
unsigned long g_ulMeasuredSpeed
```

Description:

The current speed of the motor. This value is updated based on whether the encoder or Hall sensors are being used.

8.2.3.17 g_ulPower [static]

Definition:

```
static unsigned long g_ulPower
```

Description:

The current motor drive power in watts, expressed as a 18.14 fixed-point value.

8.2.3.18 g_ulPowerFract [static]

Definition:

```
static unsigned long g_ulPowerFract
```

Description:

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

8.2.3.19 g_ulPowerWhole [static]

Definition:

```
static unsigned long g_ulPowerWhole
```

Description:

The whole part of the current motor drive power. This is used in conjunction with g_ulPowerFract to compute the value for g_ulPower.

8.2.3.20 g_ulSineTarget [static]

Definition:

```
static unsigned long g_ulSineTarget
```

Description:

The Sine modulation target speed.

8.2.3.21 g_ulSpeed [static]

Definition:

```
static unsigned long g_ulSpeed
```

Description:

The current motor drive speed in RPM, expressed as a 18.14 fixed-point value.

8.2.3.22 g_ulSpeedFract [static]

Definition:

```
static unsigned long g_ulSpeedFract
```

Description:

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

8.2.3.23 g_ulSpeedWhole [static]

Definition:

```
static unsigned long g_ulSpeedWhole
```

Description:

The whole part of the current motor drive speed. This is used in conjunction with g_ulSpeedFract to compute the value for g_ulSpeed.

8.2.3.24 g_ulStartupDutyCycle [static]

Definition:

```
static unsigned long g_ulStartupDutyCycle
```

Description:

The current Duty Cycle for startup mode.

8.2.3.25 g_ulStartupDutyCycleRamp [static]

Definition:

```
static unsigned long g_ulStartupDutyCycleRamp
```

Description:

The Startup Duty Cycle acceleration ramp value.

8.2.3.26 g_ulStartupPeriod [static]

Definition:

```
static unsigned long g_ulStartupPeriod
```

Description:

The period, in ticks, for startup commutation timer.

8.2.3.27 g_ulStartupState [static]

Definition:

```
static unsigned long g_ulStartupState
```

Description:

The current state for the Startup state machine.

8.2.3.28 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.

8.2.3.29 g_ulStateCount [static]

Definition:

```
static unsigned long g_ulStateCount
```

Description:

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

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.

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

9.2.1.2 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_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.

9.2.3.3 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.

9.2.3.4 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.

9.2.3.5 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.

9.2.3.6 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.

9.2.3.7 g_ulUIOnboardPotCount [static]

Definition:

```
static unsigned long g_ulUIOnboardPotCount
```

Description:

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

9.2.3.8 g_ulUIOnboardPotCount [static]

Definition:

```
static unsigned long g_ulUIOnboardPotCount
```

Description:

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

9.2.3.9 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.

9.2.3.10 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.

9.2.3.11 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.

9.2.3.12 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.

9.2.3.13 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).

9.2.3.14 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).

9.2.3.15 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.

9.2.3.16 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.

9.2.3.17 g_ulUIOnboardSwitches [static]

Definition:

```
static unsigned long g_ulUIOnboardSwitches
```

Description:

The debounced state of the switches.

9.2.3.18 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_CAN0RX_PIN
- PIN_CAN0RX_PORT
- PIN_CAN0TX_PIN
- PIN_CAN0TX_PORT
- PIN_CFG0_PIN
- PIN_CFG0_PORT
- PIN_CFG1_PIN
- PIN_CFG1_PORT
- PIN_CFG2_PIN
- PIN_CFG2_PORT
- PIN_ENCA_PIN
- PIN_ENCA_PORT
- PIN_ENCB_PIN
- PIN_ENCB_PORT
- PIN_HALLA_PIN
- PIN_HALLA_PORT
- PIN_HALLB_PIN
- PIN_HALLB_PORT
- PIN_HALLC_PIN
- PIN_HALLC_PORT
- PIN_INDEX_PIN
- PIN_INDEX_PORT
- PIN_IPHASEA

- PIN_IPHASEB
- PIN_IPHASEC
- PIN_LEDFAULT_PIN
- PIN_LEDFAULT_PORT
- PIN_LEDRUN_PIN
- PIN_LEDRUN_PORT
- PIN_LEDSTATUS0_PIN
- PIN_LEDSTATUS0_PORT
- PIN_LEDSTATUS1_PIN
- PIN_LEDSTATUS1_PORT
- PIN_PHASEA_HIGH_PIN
- PIN_PHASEA_HIGH_PORT
- PIN_PHASEA_LOW_PIN
- PIN_PHASEA_LOW_PORT
- PIN_PHASEB_HIGH_PIN
- PIN_PHASEB_HIGH_PORT
- PIN_PHASEB_LOW_PIN
- PIN_PHASEB_LOW_PORT
- PIN_PHASEC_HIGH_PIN
- PIN_PHASEC_HIGH_PORT
- PIN_PHASEC_LOW_PIN
- PIN_PHASEC_LOW_PORT
- PIN_SWITCH_PIN
- PIN_SWITCH_PIN_BIT
- PIN_SWITCH_PORT
- PIN_UART0RX_PIN
- PIN_UART0RX_PORT
- PIN_UART0TX_PIN
- PIN_UART0TX_PORT
- PIN_VANALOG
- PIN_VBEMFA
- PIN_VBEMFB
- PIN_VBEMFC
- PIN_VSENSE
- PWM_PHASEA_HIGH
- PWM_PHASEA_LOW
- PWM_PHASEB_HIGH
- PWM_PHASEB_LOW
- PWM_PHASEC_HIGH
- PWM_PHASEC_LOW

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_CAN0RX_PIN

Definition:
```
#define PIN_CAN0RX_PIN
```

Description:
The GPIO pin on which the CAN0 Rx pin resides.

10.2.1.4 PIN_CAN0RX_PORT

Definition:
```
#define PIN_CAN0RX_PORT
```

Description:
The GPIO port on which the CAN0 Rx pin resides.

10.2.1.5 PIN_CAN0TX_PIN

Definition:
```
#define PIN_CAN0TX_PIN
```

Description:
The GPIO pin on which the CAN0 Tx pin resides.

10.2.1.6 PIN_CAN0TX_PORT

Definition:
```
#define PIN_CAN0TX_PORT
```

Description:
The GPIO port on which the CAN0 Tx pin resides.

10.2.1.7 PIN_CFG0_PIN

Definition:

```
#define PIN_CFG0_PIN
```

Description:

The GPIO pin on which the CFG0 pin resides.

10.2.1.8 PIN_CFG0_PORT

Definition:

```
#define PIN_CFG0_PORT
```

Description:

The GPIO port on which the CFG0 pin resides.

10.2.1.9 PIN_CFG1_PIN

Definition:

```
#define PIN_CFG1_PIN
```

Description:

The GPIO pin on which the CFG1 pin resides.

10.2.1.10 PIN_CFG1_PORT

Definition:

```
#define PIN_CFG1_PORT
```

Description:

The GPIO port on which the CFG1 pin resides.

10.2.1.11 PIN_CFG2_PIN

Definition:

```
#define PIN_CFG2_PIN
```

Description:

The GPIO pin on which the CFG2 pin resides.

10.2.1.12 PIN_CFG2_PORT

Definition:

```
#define PIN_CFG2_PORT
```

Description:

The GPIO port on which the CFG2 pin resides.

10.2.1.13 PIN_ENCA_PIN

Definition:

```
#define PIN_ENCA_PIN
```

Description:

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

10.2.1.14 PIN_ENCA_PORT

Definition:

```
#define PIN_ENCA_PORT
```

Description:

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

10.2.1.15 PIN_ENCB_PIN

Definition:

```
#define PIN_ENCB_PIN
```

Description:

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

10.2.1.16 PIN_ENCB_PORT

Definition:

```
#define PIN_ENCB_PORT
```

Description:

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

10.2.1.17 PIN_HALLA_PIN

Definition:

```
#define PIN_HALLA_PIN
```

Description:

The GPIO pin on which the HALL A pin resides.

10.2.1.18 PIN_HALLA_PORT

Definition:

```
#define PIN_HALLA_PORT
```

Description:

The GPIO port on which the HALL A pin resides.

10.2.1.19 PIN_HALLB_PIN

Definition:

```
#define PIN_HALLB_PIN
```

Description:

The GPIO pin on which the HALL B pin resides.

10.2.1.20 PIN_HALLB_PORT

Definition:

```
#define PIN_HALLB_PORT
```

Description:

The GPIO port on which the HALL B pin resides.

10.2.1.21 PIN_HALLC_PIN

Definition:

```
#define PIN_HALLC_PIN
```

Description:

The GPIO pin on which the HALL C pin resides.

10.2.1.22 PIN_HALLC_PORT

Definition:

```
#define PIN_HALLC_PORT
```

Description:

The GPIO port on which the HALL C pin resides.

10.2.1.23 PIN_INDEX_PIN

Definition:

```
#define PIN_INDEX_PIN
```

Description:

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

10.2.1.24 PIN_INDEX_PORT

Definition:

```
#define PIN_INDEX_PORT
```

Description:

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

10.2.1.25 PIN_IPHASEA

Definition:

```
#define PIN_IPHASEA
```

Description:

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

10.2.1.26 PIN_IPHASEB

Definition:

```
#define PIN_IPHASEB
```

Description:

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

10.2.1.27 PIN_IPHASEC

Definition:

```
#define PIN_IPHASEC
```

Description:

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

10.2.1.28 PIN_LEDFAULT_PIN

Definition:

```
#define PIN_LEDFAULT_PIN
```

Description:

The GPIO pin on which the fault LED resides.

10.2.1.29 PIN_LEDFAULT_PORT

Definition:

```
#define PIN_LEDFAULT_PORT
```

Description:

The GPIO port on which the fault LED resides.

10.2.1.30 PIN_LEDRUN_PIN

Definition:

```
#define PIN_LEDRUN_PIN
```

Description:

The GPIO pin on which the run LED resides.

10.2.1.31 PIN_LEDRUN_PORT

Definition:

```
#define PIN_LEDRUN_PORT
```

Description:

The GPIO port on which the run LED resides.

10.2.1.32 PIN_LEDSTATUS0_PIN

Definition:

```
#define PIN_LEDSTATUS0_PIN
```

Description:

The GPIO pin on which the Ethernet status zero LED resides.

10.2.1.33 PIN_LEDSTATUS0_PORT

Definition:

```
#define PIN_LEDSTATUS0_PORT
```

Description:

The GPIO port on which the Ethernet status zero LED resides.

10.2.1.34 PIN_LEDSTATUS1_PIN

Definition:

```
#define PIN_LEDSTATUS1_PIN
```

Description:

The GPIO pin on which the Ethernet status one LED resides.

10.2.1.35 PIN_LEDSTATUS1_PORT

Definition:

```
#define PIN_LEDSTATUS1_PORT
```

Description:

The GPIO port on which the Ethernet status one LED resides.

10.2.1.36 PIN_PHASEA_HIGH_PIN

Definition:

```
#define PIN_PHASEA_HIGH_PIN
```

Description:

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

10.2.1.37 PIN_PHASEA_HIGH_PORT

Definition:

```
#define PIN_PHASEA_HIGH_PORT
```

Description:

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

10.2.1.38 PIN_PHASEA_LOW_PIN

Definition:

```
#define PIN_PHASEA_LOW_PIN
```

Description:

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

10.2.1.39 PIN_PHASEA_LOW_PORT

Definition:

```
#define PIN_PHASEA_LOW_PORT
```

Description:

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

10.2.1.40 PIN_PHASEB_HIGH_PIN

Definition:

```
#define PIN_PHASEB_HIGH_PIN
```

Description:

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

10.2.1.41 PIN_PHASEB_HIGH_PORT

Definition:

```
#define PIN_PHASEB_HIGH_PORT
```

Description:

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

10.2.1.42 PIN_PHASEB_LOW_PIN

Definition:

```
#define PIN_PHASEB_LOW_PIN
```

Description:

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

10.2.1.43 PIN_PHASEB_LOW_PORT

Definition:

```
#define PIN_PHASEB_LOW_PORT
```

Description:

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

10.2.1.44 PIN_PHASEC_HIGH_PIN

Definition:

```
#define PIN_PHASEC_HIGH_PIN
```

Description:

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

10.2.1.45 PIN_PHASEC_HIGH_PORT

Definition:

```
#define PIN_PHASEC_HIGH_PORT
```

Description:

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

10.2.1.46 PIN_PHASEC_LOW_PIN

Definition:

```
#define PIN_PHASEC_LOW_PIN
```

Description:

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

10.2.1.47 PIN_PHASEC_LOW_PORT

Definition:

```
#define PIN_PHASEC_LOW_PORT
```

Description:

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

10.2.1.48 PIN_SWITCH_PIN

Definition:

```
#define PIN_SWITCH_PIN
```

Description:

The GPIO pin on which the user push button resides.

10.2.1.49 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.

10.2.1.50 PIN_SWITCH_PORT

Definition:

```
#define PIN_SWITCH_PORT
```

Description:

The GPIO port on which the user push button resides.

10.2.1.51 PIN_UART0RX_PIN

Definition:

```
#define PIN_UART0RX_PIN
```

Description:

The GPIO pin on which the UART0 Rx pin resides.

10.2.1.52 PIN_UART0RX_PORT

Definition:

```
#define PIN_UART0RX_PORT
```

Description:

The GPIO port on which the UART0 Rx pin resides.

10.2.1.53 PIN_UART0TX_PIN

Definition:

```
#define PIN_UART0TX_PIN
```

Description:

The GPIO pin on which the UART0 Tx pin resides.

10.2.1.54 PIN_UART0TX_PORT

Definition:

```
#define PIN_UART0TX_PORT
```

Description:

The GPIO port on which the UART0 Tx pin resides.

10.2.1.55 PIN_VANALOG

Definition:

```
#define PIN_VANALOG
```

Description:

The ADC channel on which the Analog Input voltage sense resides.

10.2.1.56 PIN_VBEMFA

Definition:

```
#define PIN_VBEMFA
```

Description:

The ADC channel on which the DC Back EMF (Phase A) voltage sense resides.

10.2.1.57 PIN_VBEMFB

Definition:

```
#define PIN_VBEMFB
```

Description:

The ADC channel on which the DC Back EMF (Phase B) voltage sense resides.

10.2.1.58 PIN_VBEMFC

Definition:

```
#define PIN_VBEMFC
```

Description:

The ADC channel on which the DC Back EMF (Phase C) voltage sense resides.

10.2.1.59 PIN_VSENSE

Definition:

```
#define PIN_VSENSE
```

Description:

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

10.2.1.60 PWM_PHASEA_HIGH

Definition:

```
#define PWM_PHASEA_HIGH
```

Description:

The PWM channel on which the phase A high side resides.

10.2.1.61 PWM_PHASEA_LOW

Definition:

```
#define PWM_PHASEA_LOW
```

Description:

The PWM channel on which the phase A low side resides.

10.2.1.62 PWM_PHASEB_HIGH

Definition:

```
#define PWM_PHASEB_HIGH
```

Description:

The PWM channel on which the phase B high side resides.

10.2.1.63 PWM_PHASEB_LOW

Definition:

```
#define PWM_PHASEB_LOW
```

Description:

The PWM channel on which the phase B low side resides.

10.2.1.64 PWM_PHASEC_HIGH

Definition:

```
#define PWM_PHASEC_HIGH
```

Description:

The PWM channel on which the phase C high side resides.

10.2.1.65 PWM_PHASEC_LOW

Definition:

```
#define PWM_PHASEC_LOW
```

Description:

The PWM channel on which the phase C low side resides.

11 PWM Control

11.1 Introduction

The generated motor drive waveforms are driven to the inverter bridge with the Pulse Width Modulator (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. This mode 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 target speed 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 in 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, 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_FLAG_NEW_DUTY_CYCLE
- PWM_FLAG_NEW_FREQUENCY
- PWM_FLAG_NEW_PRECHARGE

Functions

- void PWM0IntHandler (void)
- void PWMClearDeadBand (void)
- unsigned long PWMGetPeriodCount (void)
- void PWMInit (void)
- void PWMOutputOff (void)
- void PWMOutputOn (void)
- void PWMOutputPrecharge (void)
- void PWMOutputTrapezoid (unsigned long ulEnable)
- void PWMReducePeriodCount (unsigned long ulCount)
- void PWMSetDeadBand (void)
- void PWMSetDutyCycle (unsigned long ulDutyCycleA, unsigned long ulDutyCycleB, unsigned long ulDutyCycleC)
- 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_ulPWMDutyCycleA
- static unsigned long g_ulPWMDutyCycleB
- static unsigned long g_ulPWMDutyCycleC
- static unsigned long g_ulPWMFlags
- unsigned long g_ulPWMFrequency
- static unsigned long g_ulPWMMillisecondCount
- static unsigned long g_ulPWMPeriodCount
- unsigned long g_ulPWMWidth
- unsigned long g_ulTrapDutyCycle

11.2.1 Define Documentation

11.2.1.1 PWM_FLAG_NEW_DUTY_CYCLE

Definition:

```
#define PWM_FLAG_NEW_DUTY_CYCLE
```

Description:

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

11.2.1.2 PWM_FLAG_NEW_FREQUENCY

Definition:

```
#define PWM_FLAG_NEW_FREQUENCY
```

Description:

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

11.2.1.3 PWM_FLAG_NEW_PRECHARGE

Definition:

```
#define PWM_FLAG_NEW_PRECHARGE
```

Description:

The bit number of the flag in g_ulPWMFlags which indicates that a Pre- charge process has been started.

11.2.2 Function Documentation

11.2.2.1 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.

11.2.2.2 PWMClearDeadBand

Disables the dead timers for the PWM generators.

Prototype:

```
void
PWMClearDeadBand(void)
```

Description:

This function disables the dead timers for all three PWM generators.

Returns:

None.

11.2.2.3 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.

11.2.2.4 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.

11.2.2.5 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.

11.2.2.6 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.

11.2.2.7 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.

11.2.2.8 PWMOutputTrapezoid

Enable/Disable PWM outputs as needed for Trapezoid modulation.

Prototype:

```
void
PWMOutputTrapezoid(unsigned long ulEnable)
```

Parameters:

ulEnable is the bit-mapped value representing which phase(s) of the motor drive should be active.

Description:

This function turns off non-selected outputs and turns on selected outputs.

Returns:

None.

11.2.2.9 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.

11.2.2.10 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.

11.2.2.11 PWMSetDutyCycle

Sets the duty cycle of the generated PWM waveforms.

Prototype:

```
void
PWMSetDutyCycle(unsigned long ulDutyCycleA,
                unsigned long ulDutyCycleB,
                unsigned long ulDutyCycleC)
```

Parameters:

ulDutyCycleA 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.

ulDutyCycleB 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.

ulDutyCycleC 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.

11.2.2.12 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.

11.2.2.13 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.

11.2.2.14 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.

11.2.2.15 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_ulPWMDutyCycleA [static]

Definition:

```
static unsigned long g_ulPWMDutyCycleA
```

Description:

The duty cycle of the waveform output to the A phase of the bridge.

11.2.3.4 g_ulPWMDutyCycleB [static]

Definition:

```
static unsigned long g_ulPWMDutyCycleB
```

Description:

The duty cycle of the waveform output to the B phase of the bridge.

11.2.3.5 g_ulPWMDutyCycleC [static]

Definition:

```
static unsigned long g_ulPWMDutyCycleC
```

Description:

The duty cycle of the waveform output to the C 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.

11.2.3.10 g_ulPWMWidth

Definition:

```
unsigned long g_ulPWMWidth
```

Description:

The number of PWM clocks in a single PWM duty cycle.

11.2.3.11 g_ulTrapDutyCycle

Definition:

```
unsigned long g_ulTrapDutyCycle
```

Description:

The duty cycle (0 to 10000) used for trapezoid current calculations.

12 Sine Wave Modulation

12.1 Introduction

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.

12.2 Definitions

Functions

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

12.2.1 Function Documentation

12.2.1.1 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.

13 Speed Sensing

13.1 Introduction

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 while using the encoder, 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.

13.2 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 ulNewSpeed)
- void SpeedSenseInit (void)

Variables

- unsigned long g_ulRotorSpeed
- static unsigned long g_ulSpeedFlags
- static unsigned long g_ulSpeedPrevious
- static unsigned long g_ulSpeedTime

13.2.1 Define Documentation

13.2.1.1 EDGE_DELTA

Definition:

```
#define EDGE_DELTA
```

Description:

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

13.2.1.2 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.

13.2.1.3 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.

13.2.1.4 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.

13.2.1.5 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).

13.2.1.6 QEI_INT_RATE

Definition:

```
#define QEI_INT_RATE
```

Description:

The rate at which the QEI velocity interrupt occurs.

13.2.2 Function Documentation

13.2.2.1 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 speed.

Returns:

None.

13.2.2.2 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 speed determination, the number of edges counted during the last velocity period is used as a measure of the rotor speed.

Returns:
None.

13.2.2.3 SpeedNewValue [static]

Updates the current rotor speed.

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

Parameters:
ulNewSpeed is the newly measured speed.

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

Returns:
None.

13.2.2.4 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.

13.2.3 Variable Documentation

13.2.3.1 g_ulRotorSpeed

Definition:
```
unsigned long g_ulRotorSpeed
```

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

13.2.3.2 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.

13.2.3.3 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.

13.2.3.4 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.

14 Trapezoid Modulation

14.1 Introduction

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

14.2 Definitions

Defines

- PHASE_A
- PHASE_B
- PHASE_C

Functions

- void TrapModulate (unsigned long ulHall)

Variables

- static const unsigned long g_ulHallToPhase120[8]
- static const unsigned long g_ulHallToPhase60[8]

14.2.1 Define Documentation

14.2.1.1 PHASE_A

Definition:

```
#define PHASE_A
```

Description:

PWM Phase A (High + Low).

14.2.1.2 PHASE_B

Definition:

```
#define PHASE_B
```

Description:

PWM Phase B (High + Low).

14.2.1.3 PHASE_C

Definition:

```
#define PHASE_C
```

Description:

PWM Phase C (High + Low).

14.2.2 Function Documentation

14.2.2.1 TrapModulate

Controls trapezoid modulated waveforms.

Prototype:

```
void
TrapModulate(unsigned long ulHall)
```

Parameters:

ulHall is the current Hall state value for the motor. This value may be read directly from the Hall sensors, if installed, or derived from the Back EMF or Linear Hall sensor readings.

Description:

This function will control the PWM generator channels based on the changes in the Hall Effect sensor value.

Returns:

None.

14.2.3 Variable Documentation

14.2.3.1 g_ulHallToPhase120 [static]

Definition:

```
static const unsigned long g_ulHallToPhase120[8]
```

Description:

Mapping from Hall States to Phase Drive states (120 degree spacing).

This array maps the hall state value to the set of PWM signals that should be driving at that time.

Phase A	-	-	Z	+	+	Z
Phase B	+	Z	-	-	Z	+
Phase C	Z	+	+	Z	-	-
Hall A	1	1	1	0	0	0
Hall B	0	0	1	1	1	0
Hall C	1	0	0	0	1	1

14.2.3.2 g_ulHallToPhase60 [static]

Definition:

```
static const unsigned long g_ulHallToPhase60[8]
```

Description:

Mapping from Hall States to Phase Drive states (60 degree spacing).

This array maps the hall state value to the set of PWM signals that should be driving at that time.

Phase A	+	Z	-	-	Z	+
Phase B	-	-	Z	+	+	Z
Phase C	Z	+	+	Z	-	-
Hall A	0	1	1	1	0	0
Hall B	0	0	1	1	1	0
Hall C	0	0	0	1	1	1

15 User Interface

15.1 Introduction

There are two user interfaces for the the Brushless DC motor application. One uses an push button for basic control of the motor and two LEDs for basic status feedback, and the other uses the Ethernet 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 push button and two LEDs. The push button cycles between run forward, stop, run backward, stop.

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.

A periodic interrupt is used to poll the state of the push button and perform debouncing.

The Ethernet user interface is entirely handled by the Ethernet 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 Ethernet 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.

15.2 Definitions

Data Structures

- tDriveParameters

Defines

- CONTROL_TYPE_POWER
- CONTROL_TYPE_SPEED
- FLAG_BRAKE_BIT
- FLAG_BRAKE_OFF
- FLAG_BRAKE_ON
- FLAG_DECAY_BIT

- FLAG_DECAY_FAST
- FLAG_DECAY_SLOW
- FLAG_DIR_BACKWARD
- FLAG_DIR_BIT
- FLAG_DIR_FORWARD
- FLAG_ENCODER_ABSENT
- FLAG_ENCODER_BIT
- FLAG_ENCODER_PRESENT
- FLAG_PWM_FREQUENCY_12K
- FLAG_PWM_FREQUENCY_16K
- FLAG_PWM_FREQUENCY_20K
- FLAG_PWM_FREQUENCY_25K
- FLAG_PWM_FREQUENCY_40K
- FLAG_PWM_FREQUENCY_50K
- FLAG_PWM_FREQUENCY_80K
- FLAG_PWM_FREQUENCY_8K
- FLAG_PWM_FREQUENCY_MASK
- FLAG_SENSOR_POLARITY_BIT
- FLAG_SENSOR_POLARITY_HIGH
- FLAG_SENSOR_POLARITY_LOW
- FLAG_SENSOR_SPACE_120
- FLAG_SENSOR_SPACE_60
- FLAG_SENSOR_SPACE_BIT
- FLAG_SENSOR_TYPE_BIT
- FLAG_SENSOR_TYPE_GPIO
- FLAG_SENSOR_TYPE_LINEAR
- MOD_TYPE_SENSORLESS
- MOD_TYPE_SINE
- MOD_TYPE_TRAPEZOID
- NUM_SWITCHES
- TIMER1A_INT_RATE
- UI_INT_RATE

Functions

- void SysTickIntHandler (void)
- void Timer1AIntHandler (void)
- static void UIButtonHold (void)
- void UIButtonPress (void)
- static void UIConnectionTimeout (void)
- static void UIControlType (void)
- static void UIDecayMode (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)
- unsigned long UIGetTicks (void)
- void UIInit (void)
- static void UILEDBlink (unsigned long ulIdx, unsigned short usRate, unsigned short usPeriod)
- static void UIModulationType (void)
- static void UIPAdjI (void)
- void UIParamLoad (void)
- void UIParamSave (void)
- static void UIPWMFrequencySet (void)
- void UIRun (void)
- void UIRunLEDBlink (unsigned short usRate, unsigned short usPeriod)
- static void UISensorPolarity (void)
- static void UISensorType (void)
- void UIStop (void)
- static void UIUpdateRate (void)
- void UIUpgrade (void)

Variables

- static long g_lFAdjI
- static long g_lPAdjI
- static const unsigned char g_pucLEDPin[2]
- static const unsigned long g_pulLEDBase[2]
- unsigned long g_pulUIHoldCount[NUM_SWITCHES]
- static unsigned short g_pusBlinkPeriod[2]
- static unsigned short g_pusBlinkRate[2]
- tDriveParameters g_sParameters
- const tUIParameter g_sUIParameters[ ]
- const tUIRealTimeData g_sUIRealTimeData[ ]
- const tUIOnboardSwitch g_sUISwitches[ ]
- unsigned char g_ucBoardID
- static unsigned char g_ucControlType
- unsigned char g_ucCPUUsage
- static unsigned char g_ucDecayMode
- static unsigned char g_ucDirection
- static unsigned char g_ucDynamicBrake
- static unsigned char g_ucEncoder
- static unsigned char g_ucFrequency
- static unsigned char g_ucModulationType
- static unsigned char g_ucSensorPolarity
- static unsigned char g_ucSensorType
- static unsigned char g_ucUpdateRate
- static unsigned long g_ulBlinkCount

- unsigned long g_ulDebugInfo[8]
- unsigned long g_ulGPIOData
- const unsigned long g_ulUINumButtons
- const unsigned long g_ulUINumParameters
- const unsigned long g_ulUINumRealTimeData
- const unsigned long g_ulUITargetType
- static unsigned long g_ulUITickCount
- static unsigned long g_ulUIUseOnboard
- static unsigned short g_usAnalogInputVoltage
- const unsigned short g_usFirmwareVersion

15.2.1 Data Structure Documentation

15.2.1.1 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 ucModulationType;
    unsigned short usAccel;
    unsigned short usDecel;
    short sMinCurrent;
    short sMaxCurrent;
    unsigned char ucPrechargeTime;
    unsigned char ucMaxTemperature;
    unsigned short usFlags;
    unsigned short usNumEncoderLines;
    unsigned short usAccelPower;
    unsigned long ulMinSpeed;
    unsigned long ulMaxSpeed;
    unsigned long ulMinVBus;
    unsigned long ulMaxVBus;
    unsigned long ulBrakeOnV;
    unsigned long ulBrakeOffV;
    unsigned long ulDecelV;
    long lFAdjP;
    long lFAdjI;
    long lPAdjP;
    unsigned long ulBrakeMax;
    unsigned long ulBrakeCool;
    short sAccelCurrent;
    unsigned short usDecelPower;
    unsigned long ulConnectionTimeout;
```

```
    unsigned char ucBEMFSkipCount;
    unsigned char ucControlType;
    unsigned short usSensorlessBEMFThresh;
    unsigned short usStartupCount;
    unsigned short usSensorlessRampTime;
    short sTargetCurrent;
    unsigned char ucPad2[2];
    unsigned long ulSensorlessStartVoltage;
    unsigned long ulSensorlessEndVoltage;
    unsigned long ulSensorlessStartSpeed;
    unsigned long ulSensorlessEndSpeed;
    unsigned long ulMinPower;
    unsigned long ulMaxPower;
    unsigned long ulTargetPower;
    unsigned long ulTargetSpeed;
    long lPAdjI;
}
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.

ucModulationType The motor drive (modulation) scheme in use.

usAccel The rate of acceleration, specified in RPM per second.

usDecel The rate of deceleration, specified in RPM per second.

sMinCurrent The minimum current through the motor drive during operation, specified in milliamperes.

sMaxCurrent The maximum current through the motor drive during operation, specified in milli-amperes.

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_DECAY_BIT, FLAG_DIR_BIT, FLAG_ENCODER_BIT, FLAG_BRAKE_BIT, FLAG_SENSOR_TYPE_BIT, and FLAG_SENSOR_POLARITY_BIT.

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

usAccelPower The rate of acceleration, specified in milliwatts per second.

ulMinSpeed The minimum speed of the motor drive, specified in RPM.

ulMaxSpeed The maximum speed of the motor drive, specified in RPM.

ulMinVBus The minimum bus voltage during operation, specified in millivolts.

ulMaxVBus The maximum bus voltage during operation, specified in millivolts.

ulBrakeOnV The bus voltage at which the braking circuit is engaged, specified in millivolts.

ulBrakeOffV The bus voltage at which the braking circuit is disengaged, specified in millivolts.

ulDeceIV The DC bus voltage at which the deceleration rate is reduced, specified in millivolts.

lFAdjP The P coefficient of the frequency adjust PID controller.

lFAdjI The I coefficient of the frequency adjust PID controller.

lPAdjP The P coefficient of the power adjust PID 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.

sAccelCurrent The motor current at which the acceleration rate is reduced, specified in milliamperes.

usDecelPower The rate of deceleration, specified in milliwatts per second.

ulConnectionTimeout The Ethernet Connection Timeout, specified in seconds.

ucBEMFSkipCount The number of PWM periods to skip in a commutation before looking for the Back EMF zero-crossing event.

ucControlType The control mode for the motor drive algorithm.

usSensorlessBEMFThresh The Back EMF Threshold Voltage for sensorless startup.

usStartupCount The number of counts (commutations) for startup in sensorless mode.

usSensorlessRampTime The open-loop sensorless ramp time, specified in milliseconds.

sTargetCurrent The motor current limit for motor operation.

ucPad2 Padding to ensure consistent parameter block alignment.

ulSensorlessStartVoltage The starting voltage for sensorless startup in millivolts.

ulSensorlessEndVoltage The ending voltage for sensorless startup in millivolts.

ulSensorlessStartSpeed The starting speed for sensorless startup in RPM.

ulSensorlessEndSpeed The ending speed for sensorless startup in RPM.

ulMinPower The minimum power setting in milliwatts.

ulMaxPower The maximum power setting in milliwatts.

ulTargetPower The target power setting in milliwatts.

ulTargetSpeed The target speed setting in RPM.

lPAdjI The I coefficient of the power adjust PID controller.

Description:

This structure contains the Brushless DC 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.

15.2.2 Define Documentation

15.2.2.1 CONTROL_TYPE_POWER

Definition:

```
#define CONTROL_TYPE_POWER
```

Description:

The value for ucControlType that indicates that the motor is being driven using power as the closed loop control target.

15.2.2.2 CONTROL_TYPE_SPEED

Definition:

```
#define CONTROL_TYPE_SPEED
```

Description:

The value for ucControlType that indicates that the motor is being driven using speed as the closed loop control target.

15.2.2.3 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.

15.2.2.4 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.

15.2.2.5 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.

15.2.2.6 FLAG_DECAY_BIT

Definition:

```
#define FLAG_DECAY_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the decay mode for the trapezoid motor drive. This field will be one of FLAG_DECAY_FAST or FLAG_DECAY_SLOW

15.2.2.7 FLAG_DECAY_FAST

Definition:

```
#define FLAG_DECAY_FAST
```

Description:

The value of the FLAG_DECAY_BIT flag that indicates that the motor is to be driven with fast decay in trapezoid mode.

15.2.2.8 FLAG_DECAY_SLOW

Definition:

```
#define FLAG_DECAY_SLOW
```

Description:

The value of the FLAG_DECAY_BIT flag that indicates that the motor is to be driven with slow decay in trapezoid mode.

15.2.2.9 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.

15.2.2.10 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.

15.2.2.11 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.

15.2.2.12 FLAG_ENCODER_ABSENT

Definition:

```
#define FLAG_ENCODER_ABSENT
```

Description:

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

15.2.2.13 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.

15.2.2.14 FLAG_ENCODER_PRESENT

Definition:

```
#define FLAG_ENCODER_PRESENT
```

Description:

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

15.2.2.15 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.

15.2.2.16 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.

15.2.2.17 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.

15.2.2.18 FLAG_PWM_FREQUENCY_25K

Definition:

```
#define FLAG_PWM_FREQUENCY_25K
```

Description:

The value of the FLAG_PWM_FREQUENCY_MASK bit field that indicates that the PWM frequency is 25 KHz.

15.2.2.19 FLAG_PWM_FREQUENCY_40K

Definition:

```
#define FLAG_PWM_FREQUENCY_40K
```

Description:

The value of the FLAG_PWM_FREQUENCY_MASK bit field that indicates that the PWM frequency is 40 KHz.

15.2.2.20 FLAG_PWM_FREQUENCY_50K

Definition:

```
#define FLAG_PWM_FREQUENCY_50K
```

Description:

The value of the FLAG_PWM_FREQUENCY_MASK bit field that indicates that the PWM frequency is 50 KHz.

15.2.2.21 FLAG_PWM_FREQUENCY_80K

Definition:

```
#define FLAG_PWM_FREQUENCY_80K
```

Description:

The value of the FLAG_PWM_FREQUENCY_MASK bit field that indicates that the PWM frequency is 80 KHz.

15.2.2.22 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.

15.2.2.23 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.

15.2.2.24 FLAG_SENSOR_POLARITY_BIT

Definition:

```
#define FLAG_SENSOR_POLARITY_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the polarity of the Hall Effect Sensor(s) inputs. This field will be one of FLAG_SENSOR_POLARITY_HIGH or FLAG_SENSOR_POLARITY_LOW.

15.2.2.25 FLAG_SENSOR_POLARITY_HIGH

Definition:

```
#define FLAG_SENSOR_POLARITY_HIGH
```

Description:

The value of the FLAG_SENSOR_POLARITY_BIT flag that indicates that the Hall Effect sensor(s) are configured as active high.

15.2.2.26 FLAG_SENSOR_POLARITY_LOW

Definition:

```
#define FLAG_SENSOR_POLARITY_LOW
```

Description:

The value of the FLAG_SENSOR_POLARITY_BIT flag that indicates that the Hall Effect sensor(s) are configured as active low.

15.2.2.27 FLAG_SENSOR_SPACE_120

Definition:

```
#define FLAG_SENSOR_SPACE_120
```

Description:

The value of the FLAG_SENSOR_SPACE_BIT flag that indicates that the Hall Effect sensor(s) are spaced at 120 degrees.

15.2.2.28 FLAG_SENSOR_SPACE_60

Definition:

```
#define FLAG_SENSOR_SPACE_60
```

Description:

The value of the FLAG_SENSOR_POLARITY_BIT flag that indicates that the Hall Effect sensor(s) are spaced at 60 degrees.

15.2.2.29 FLAG_SENSOR_SPACE_BIT

Definition:

```
#define FLAG_SENSOR_SPACE_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the spacing of the hall sensors. This field will be one of FLAG_SENSOR_SPACE_120 or FLAG_SENSOR_SPACE_60.

15.2.2.30 FLAG_SENSOR_TYPE_BIT

Definition:

```
#define FLAG_SENSOR_TYPE_BIT
```

Description:

The bit number of the flag in the usFlags member of tDriveParameters that defines the type of Hall Effect Sensor(s) for position/speed feedback. This field will be one of FLAG_SENSOR_TYPE_GPIO or FLAG_SENSOR_TYPE_LINEAR.

15.2.2.31 FLAG_SENSOR_TYPE_GPIO

Definition:

```
#define FLAG_SENSOR_TYPE_GPIO
```

Description:

The value of the FLAG_SENSOR_TYPE_BIT flag that indicates that the Hall Effect sensor(s) are digital GPIO inputs.

15.2.2.32 FLAG_SENSOR_TYPE_LINEAR

Definition:

```
#define FLAG_SENSOR_TYPE_LINEAR
```

Description:

The value of the FLAG_SENSOR_TYPE_BIT flag that indicates that the Hall Effect sensor(s) are Analog/Linear ADC inputs.

15.2.2.33 MOD_TYPE_SENSORLESS

Definition:

```
#define MOD_TYPE_SENSORLESS
```

Description:

The value for ucModulationType that indicates that the motor is being driven with trapezoid modulation, in sensorless mode.

15.2.2.34 MOD_TYPE_SINE

Definition:

```
#define MOD_TYPE_SINE
```

Description:

The value for ucModulationType that indicates that the motor is being driven with sinusoidal modulation, using hall sensors for position sensing.

15.2.2.35 MOD_TYPE_TRAPEZOID

Definition:

```
#define MOD_TYPE_TRAPEZOID
```

Description:

The value for ucModulationType that indicates that the motor is being driven with trapezoid modulation, using hall sensors.

15.2.2.36 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.

15.2.2.37 TIMER1A_INT_RATE

Definition:

```
#define TIMER1A_INT_RATE
```

Description:

The rate at which the timer interrupt occurs.

15.2.2.38 UI_INT_RATE

Definition:

```
#define UI_INT_RATE
```

Description:

The rate at which the user interface interrupt occurs.

15.2.3 Function Documentation

15.2.3.1 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.

15.2.3.2 Timer1AIntHandler

Handles the Timer1A interrupt.

Prototype:

```
void
Timer1AIntHandler(void)
```

Description:

This function is called when Timer1A asserts its interrupt. It is responsible for keeping track of system time. This should be the highest priority interrupt.

Returns:

None.

15.2.3.3 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.

15.2.3.4 UIButtonPress

Handles button presses.

Prototype:

```
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.

15.2.3.5 UIConnectionTimeout [static]

Updates the Ethernet TCP connection timeout.

Prototype:

```
static void
UIConnectionTimeout(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.

15.2.3.6 UIControlType [static]

Updates the control mode bit for the motor dive.

Prototype:
```
static void
UIControlType(void)
```

Description:
This function is called when the variable controlling the motor control variable (speed/power) is updated. The value is then reflected into the usFlags member of g_sParameters.

Returns:
None.

15.2.3.7 UIDecayMode [static]

Updates the decay mode bit of the motor drive.

Prototype:
```
static void
UIDecayMode(void)
```

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

Returns:
None.

15.2.3.8 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.

15.2.3.9 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.

15.2.3.10 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.

Returns:

None.

15.2.3.11 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.

15.2.3.12 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.

15.2.3.13 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.

15.2.3.14 UIGetTicks

This function returns the current number of system ticks.

Prototype:

```
unsigned long
UIGetTicks(void)
```

Returns:

The number of system timer ticks.

15.2.3.15 void UIInit (void)

Initializes the user interface.

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

Returns:
None.

15.2.3.16 static void UILEDBlink (unsigned long *ulIdx*, unsigned short *usRate*, unsigned short *usPeriod*) [static]

Sets the blink rate for an LED.

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.

15.2.3.17 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.

15.2.3.18 UIPAdjI [static]

Updates the I coefficient of the power PI controller.

Prototype:
```
static void
UIPAdjI(void)
```

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

Returns:
None.

15.2.3.19 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 drive from sine to trapezoid). 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.

15.2.3.20 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.

15.2.3.21 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.

15.2.3.22 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.

15.2.3.23 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.

15.2.3.24 UISensorPolarity [static]

Updates the sensor polarity bit of the motor drive.

Prototype:

```
static void
UISensorPolarity(void)
```

Description:

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

Returns:

None.

15.2.3.25 UISensorType [static]

Updates the sensor type bit of the motor drive.

Prototype:

```
static void
UISensorType(void)
```

Description:

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

Returns:

None.

15.2.3.26 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.

15.2.3.27 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.

15.2.3.28 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:

None.

15.2.4 Variable Documentation

15.2.4.1 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().

15.2.4.2 g_lPAdjI [static]

Definition:

```
static long g_lPAdjI
```

Description:

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

15.2.4.3 g_pucLEDPin [static]

Definition:

```
static const unsigned char g_pucLEDPin[2]
```

Description:

This array contains the pin numbers of the two LEDs on the board.

15.2.4.4 g_pulLEDBase [static]

Definition:

```
static const unsigned long g_pulLEDBase[2]
```

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

15.2.4.5 g_pulUIHoldCount

Definition:
```
unsigned long g_pulUIHoldCount[NUM_SWITCHES]
```

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.

15.2.4.6 g_pusBlinkPeriod [static]

Definition:
```
static unsigned short g_pusBlinkPeriod[2]
```

Description:
The blink period of the two 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 and the fault LED is the second entry of the array.

15.2.4.7 g_pusBlinkRate [static]

Definition:
```
static unsigned short g_pusBlinkRate[2]
```

Description:
The blink rate of the two 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 and the fault LED is the second entry of the array.

15.2.4.8 g_sParameters

Definition:
```
tDriveParameters g_sParameters
```

Description:
This structure instance contains the configuration values for the Brushless DC motor drive.

15.2.4.9 g_sUIParameters

Definition:
```
const tUIParameter g_sUIParameters[]
```

Description:
An array of structures describing the Brushless DC motor drive parameters to the Ethernet user interface module.

15.2.4.10 g_sUIRealTimeData

Definition:
```
const tUIRealTimeData g_sUIRealTimeData[]
```

Description:
An array of structures describing the Brushless DC motor drive real-time data items to the serial user interface module.

15.2.4.11 g_sUISwitches

Definition:
```
const tUIOnboardSwitch g_sUISwitches[]
```

Description:
An array of structures describing the on-board switches.

15.2.4.12 g_ucBoardID

Definition:
```
unsigned char g_ucBoardID
```

Description:
This is the board id, read once from the configuration switches at startup.

15.2.4.13 g_ucControlType [static]

Definition:
```
static unsigned char g_ucControlType
```

Description:
The specification of the control variable 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 UIControlType().

15.2.4.14 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.

15.2.4.15 g_ucDecayMode [static]

Definition:

```
static unsigned char g_ucDecayMode
```

Description:

A boolean that is true when slow decay mode 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 UIDecayMode().

15.2.4.16 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 UIDirection-Set().

15.2.4.17 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().

15.2.4.18 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().

15.2.4.19 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().

15.2.4.20 g_ucModulationType [static]

Definition:

```
static unsigned char g_ucModulationType
```

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().

15.2.4.21 g_ucSensorPolarity [static]

Definition:

```
static unsigned char g_ucSensorPolarity
```

Description:

The specification of the polarity of sensor 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 UISensorPolarity().

15.2.4.22 g_ucSensorType [static]

Definition:

```
static unsigned char g_ucSensorType
```

Description:

The specification of the type of sensor 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 UISensorType().

15.2.4.23 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().

15.2.4.24 g_ulBlinkCount [static]

Definition:

```
static unsigned long g_ulBlinkCount
```

Description:

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

15.2.4.25 g_ulDebugInfo

Definition:

```
unsigned long g_ulDebugInfo[8]
```

Description:

Debug Information.

15.2.4.26 g_ulGPIOData

Definition:

```
unsigned long g_ulGPIOData
```

Description:

A 32-bit unsigned value that represents the value of various GPIO signals on the board. Bit 0 corresponds to CFG0; Bit 1 corresponds to CFG1; Bit 2 correpsonds to CFG2; Bit 8 corresponds to the Encoder A input; Bit 9 corresponds to the Encode B input; Bit 10 corresponds to the Encoder Index input.

15.2.4.27 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.

15.2.4.28 g_ulUINumParameters

Definition:

```
const unsigned long g_ulUINumParameters
```

Description:

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

15.2.4.29 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.

15.2.4.30 g_ulUITargetType

Definition:

```
const unsigned long g_ulUITargetType
```

Description:

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

15.2.4.31 g_ulUITickCount [static]

Definition:

```
static unsigned long g_ulUITickCount
```

Description:

The running count of system clock ticks.

15.2.4.32 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.

15.2.4.33 g_usAnalogInputVoltage [static]

Definition:

```
static unsigned short g_usAnalogInputVoltage
```

Description:

The Analog Input voltage, specified in millivolts.

15.2.4.34 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 Ethernet Interface

16.1 Introduction

A generic, TCP packet-based 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 same 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 ethernet protocol is contained in `ui_ethernet.c`, with `ui_ethernet.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 ethernet protocol. The file `ui_common.h` contains the definitions that are common to all of the communications API modules that are supported by the motor RDKs.

The ethernet module builds on the serial module in that the same message format is used. This message is then transmitted/received using a TCP/IP connection between the RDK board and the host application. The following message formats are defined:

16.1.1 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
```

16.1.2 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}
```

16.1.3 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.

16.1.4 Interface To The Application

The ethernet protocol handler takes care of all the ethernet 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 ethernet communication (such as starting the motor drive). The structures are used to handle the parameters and real-time data items of the motor drive.

16.2 Definitions

Defines

- DEFAULT_GATEWAY_ADDR
- DEFAULT_IPADDR
- DEFAULT_NET_MASK
- UI_PROTO_PORT
- UI_QUERY_PORT
- UIETHERNET_MAX_RECV
- UIETHERNET_MAX_XMIT

Functions

- void lwIPHostTimerHandler (void)
- static err_t UIEthernetAccept (void ∗arg, struct tcp_pcb ∗pcb, err_t err)
- static void UIEthernetClose (struct tcp_pcb ∗pcb)
- static void UIEthernetError (void ∗arg, err_t err)
- static unsigned long UIEthernetFindParameter (unsigned char ucID)
- unsigned long UIEthernetGetIPAddress (void)
- void UIEthernetInit (tBoolean bUseDHCP)
- static err_t UIEthernetPoll (void ∗arg, struct tcp_pcb ∗pcb)
- static void UIEthernetRangeCheck (unsigned long ulIdx)
- static err_t UIEthernetReceive (void ∗arg, struct tcp_pcb ∗pcb, struct pbuf ∗p, err_t err)
- static void UIEthernetReceiveUDP (void ∗arg, struct udp_pcb ∗pcb, struct pbuf ∗p, struct ip_-addr ∗addr, u16_t port)
- static void UIEthernetScanReceive (void)
- void UIEthernetSendRealTimeData (void)
- static err_t UIEthernetSent (void ∗arg, struct tcp_pcb ∗pcb, u16_t len)
- void UIEthernetTick (unsigned long ulTickMS)

Variables

- static tBoolean g_bEnableRealTimeData
- static tBoolean g_bSendRealTimeData
- static struct tcp_pcb ∗ g_psTelnetPCB
- static unsigned char g_pucUIEthernetData[UIETHERNET_MAX_XMIT]
- static unsigned char g_pucUIEthernetReceive[UIETHERNET_MAX_RECV]
- static unsigned char g_pucUIEthernetResponse[UIETHERNET_MAX_XMIT]

- static unsigned long g_pulUIRealTimeData[(DATA_NUM_ITEMS+31)/32]
- static unsigned long g_ulConnectionTimeout
- volatile unsigned long g_ulConnectionTimeoutParameter
- volatile unsigned long g_ulEthernetRXCount
- volatile unsigned long g_ulEthernetTimer
- volatile unsigned long g_ulEthernetTXCount
- unsigned long g_ulTxOutError
- static unsigned long g_ulUIEthernetReceiveRead
- static unsigned long g_ulUIEthernetReceiveWrite

16.2.1 Define Documentation

16.2.1.1 DEFAULT_GATEWAY_ADDR

Definition:

```
#define DEFAULT_GATEWAY_ADDR
```

Description:

The Default Gateway address to be used.

16.2.1.2 DEFAULT_IPADDR

Definition:

```
#define DEFAULT_IPADDR
```

Description:

The Default IP address to be used.

Default TCP/IP Address Configuration. Static IP Configuration is used if DHCP times out.

16.2.1.3 DEFAULT_NET_MASK

Definition:

```
#define DEFAULT_NET_MASK
```

Description:

The Default Network mask to be used.

16.2.1.4 UI_PROTO_PORT

Definition:

```
#define UI_PROTO_PORT
```

Description:

The port to use for TCP connections for the Motor Drive UI protocol.

16.2.1.5 UI_QUERY_PORT

Definition:

```
#define UI_QUERY_PORT
```

Description:

The port to use for UDP connections for the Motor Drive UI protocol. (This port is used for query purposes only).

16.2.1.6 UIETHERNET_MAX_RECV

Definition:

```
#define UIETHERNET_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).

16.2.1.7 UIETHERNET_MAX_XMIT

Definition:

```
#define UIETHERNET_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).

16.2.2 Function Documentation

16.2.2.1 lwIPHostTimerHandler

Handles the Ethernet interrupt hooks for the client software.

Prototype:

```
void
lwIPHostTimerHandler(void)
```

Description:

This function will run any handlers that are required to run inthe Ethernet interrupt context.

Returns:

None.

16.2.2.2 UIEthernetAccept [static]

Accept a TCP connection for motor control processing.

Prototype:

```
static err_t
UIEthernetAccept(void *arg,
                 struct tcp_pcb *pcb,
                 err_t err)
```

Parameters:

arg is not used in this implementation.

pcb is the pointer to the TCP control structure.

err is not used in this implementation.

Description:

This function is called when the lwIP TCP/IP stack has an incoming connection request on the telnet port.

Returns:

This function will return an lwIP defined error code.

16.2.2.3 UIEthernetClose [static]

Close an existing Ethernet connection.

Prototype:

```
static void
UIEthernetClose(struct tcp_pcb *pcb)
```

Parameters:

pcb is the pointer to the TCP control structure.

Description:

This function is called when the the TCP connection should be closed.

Returns:

None.

16.2.2.4 UIEthernetError [static]

lwIP TCP/IP error handling.

Prototype:

```
static void
UIEthernetError(void *arg,
                err_t err)
```

Parameters:

arg is not used in this implementation.

err is not used in this implementation.

Description:
This function is called when the lwIP TCP/IP stack has detected an error. The connection is no longer valid.

Returns:
This function will return an lwIP defined error code.

16.2.2.5 UIEthernetFindParameter [static]

Finds a parameter by ID.

Prototype:
```
static unsigned long
UIEthernetFindParameter(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.

16.2.2.6 UIEthernetGetIPAddress

Return the current IPV4 TCP/IP address.

Prototype:
```
unsigned long
UIEthernetGetIPAddress(void)
```

Description:
Read the IP address from the lwIP network interface structure, and return it in unsigned long format.

Returns:
IP address.

16.2.2.7 UIEthernetInit

Initialize the Ethernet controller and lwIP TCP/IP stack.

Prototype:
```
void
UIEthernetInit(tBoolean bUseDHCP)
```

Parameters:

bUseDHCP indicates whether the DHCP client software should be used (if the build options have enabled it).

Description:

Initialize the Ethernet controller for operation, including the setup of the MAC address and enabling of status LEDs. Also initialize the lwIP TCP/IP stack for operation, including DHCP operation.

Returns:

None.

16.2.2.8 UIEthernetPoll [static]

lwIP TCP/IP Polling/Timeout function.

Prototype:

```
static err_t
UIEthernetPoll(void *arg,
               struct tcp_pcb *pcb)
```

Parameters:

arg is not used in this implementation.

pcb is not used in this implementation.

Description:

This function is called when the lwIP TCP/IP stack has no incoming or outgoing data. It can be used to reset an idle connection.

Returns:

This function will return an lwIP defined error code.

16.2.2.9 UIEthernetRangeCheck [static]

Performs range checking on the value of a parameter.

Prototype:

```
static void
UIEthernetRangeCheck(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.

16.2.2.10 UIEthernetReceive [static]

Receive a TCP packet from lwIP for motor control processing.

Prototype:

```
static err_t
UIEthernetReceive(void *arg,
                  struct tcp_pcb *pcb,
                  struct pbuf *p,
                  err_t err)
```

Parameters:

arg is not used in this implementation.

pcb is the pointer to the TCP control structure.

p is the pointer to the PBUF structure containing the packet data.

err is used to indicate if any errors are associated with the incoming packet.

Description:

This function is called when the lwIP TCP/IP stack has an incoming packet to be processed.

Returns:

This function will return an lwIP defined error code.

16.2.2.11 UIEthernetReceiveUDP [static]

Receive a UDP packet from lwIP for motor control processing.

Prototype:

```
static void
UIEthernetReceiveUDP(void *arg,
                     struct udp_pcb *pcb,
                     struct pbuf *p,
                     struct ip_addr *addr,
                     u16_t port)
```

Parameters:

arg is not used in this implementation.

pcb is the pointer to the UDB control structure.

p is the pointer to the PBUF structure containing the packet data.

addr is the source (remote) IP address for this packet.

port is the source (remote) port for this packet.

Description:

This function is called when the lwIP TCP/IP stack has an incoming UDP packet to be processed.

Returns:

This function will return an lwIP defined error code.

16.2.2.12 UIEthernetScanReceive [static]

Scans for packets in the receive buffer.

Prototype:

```
static void
UIEthernetScanReceive(void)
```

Description:

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

Returns:

None.

16.2.2.13 UIEthernetSendRealTimeData

Sends a real-time data packet.

Prototype:

```
void
UIEthernetSendRealTimeData(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.

16.2.2.14 UIEthernetSent [static]

Callback for Ethernet transmit.

Prototype:

```
static err_t
UIEthernetSent(void *arg,
               struct tcp_pcb *pcb,
               u16_t len)
```

Parameters:

arg is not used in this implementation.

pcb is not used in this implementation.

len is not used in this implementation.

Description:

This function is called when the lwIP TCP/IP stack has received an acknowledgement for data that has been transmitted.

Returns:

This function will return an lwIP defined error code.

16.2.2.15 UIEthernetTick

Run the periodic lwIP tasks.

Prototype:

```
void
UIEthernetTick(unsigned long ulTickMS)
```

Parameters:

ulTickMS is the number of milliseconds that have elapsed since the previous call.

Description:

This code should be called periodically, to allow the lwIP periodic tasks to run (in the lwip library).

Returns:

None.

16.2.3 Variable Documentation

16.2.3.1 g_bEnableRealTimeData [static]

Definition:

```
static tBoolean g_bEnableRealTimeData
```

Description:

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

16.2.3.2 g_bSendRealTimeData [static]

Definition:

```
static tBoolean g_bSendRealTimeData
```

Description:

Flag to indicate that a Real Time Data update is ready for transmission.

16.2.3.3 g_psTelnetPCB [static]

Definition:

```
static struct tcp_pcb *g_psTelnetPCB
```

Description:

Pointer to the telnet session PCB data structure.

16.2.3.4 g_pucUIEthernetData [static]

Definition:

```
static unsigned char g_pucUIEthernetData[UIETHERNET_MAX_XMIT]
```

Description:

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

16.2.3.5 g_pucUIEthernetReceive [static]

Definition:

```
static unsigned char g_pucUIEthernetReceive[UIETHERNET_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.

16.2.3.6 g_pucUIEthernetResponse [static]

Definition:

```
static unsigned char g_pucUIEthernetResponse[UIETHERNET_MAX_XMIT]
```

Description:

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

16.2.3.7 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.

16.2.3.8 g_ulConnectionTimeout [static]

Definition:

```
static unsigned long g_ulConnectionTimeout
```

Description:

Counter for TCP connection timeout.

16.2.3.9 g_ulConnectionTimeoutParameter

Definition:

```
volatile unsigned long g_ulConnectionTimeoutParameter
```

Description:

Timeout value for TCP connection timeout timer.

16.2.3.10 g_ulEthernetRXCount

Definition:

```
volatile unsigned long g_ulEthernetRXCount
```

Description:

This global is used to store the number of Ethernet messages that have been received since power-up.

16.2.3.11 g_ulEthernetTimer

Definition:

```
volatile unsigned long g_ulEthernetTimer
```

Description:

Running count updated by the UI System Tick Handler for milliseconds.

16.2.3.12 g_ulEthernetTXCount

Definition:

```
volatile unsigned long g_ulEthernetTXCount
```

Description:

This global is used to store the number of Ethernet messages that have been transmitted since power-up.

16.2.3.13 g_ulTxOutError

Definition:

```
unsigned long g_ulTxOutError
```

Description:

Transmits a packet to the Ethernet controller.

Parameters:

pucBuffer is a pointer to the packet to be transmitted.

This function will send a packet via TCP/Ethernet. 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.

Returns:
Returns **true** if the entire packet was transmitted and **false** if not.

16.2.3.14 g_ulUIEthernetReceiveRead [static]

Definition:
```
static unsigned long g_ulUIEthernetReceiveRead
```

Description:
The offset of the next byte to be read from g_pucUIEthernetReceive.

16.2.3.15 g_ulUIEthernetReceiveWrite [static]

Definition:
```
static unsigned long g_ulUIEthernetReceiveWrite
```

Description:
The offset of the next byte to be written to g_pucUIEthernetReceive.

17 CPU Usage Module

17.1 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.

17.2 API Functions

Functions

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

17.2.1 Function Documentation

17.2.1.1 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.

17.2.1.2 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.

17.3 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();
    }
}
```

18 Flash Parameter Block Module

18.1 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.

18.2 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)

18.2.1 Function Documentation

18.2.1.1 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.

18.2.1.2 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.

18.2.1.3 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.

18.2.1.4 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.

18.3 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);
}
```

19 lwIP Wrapper Module

19.1 Introduction

The lwIP wrapper module provides a simple abstraction layer for the lwIP version 1.3.2 TCP/IP stack. The configuration of the TCP/IP stack is based on the options defined in the `lwipopts.h` file provided by the application.

The lwIPInit() function is used to initialize the lwIP TCP/IP stack. The lwIPEthernetIntHandler() is the interrupt handler function for use with the lwIP TCP/IP stack. This handler will process transmit and receive packets. If no RTOS is being used, the interrupt handler will also service the lwIP timers. The lwIPTimer() function is to be called periodically to support the TCP, ARP, DHCP and other timers used by the lwIP TCP/IP stack. If no RTOS is being used, this timer function will simply trigger an Ethernet interrupt to allow the interrupt handler to service the timers.

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

19.2 API Functions

Functions

- void lwIPEthernetIntHandler (void)
- void lwIPInit (const unsigned char *pucMAC, unsigned long ulIPAddr, unsigned long ulNetMask, unsigned long ulGWAddr, unsigned long ulIPMode)
- unsigned long lwIPLocalGWAddrGet (void)
- unsigned long lwIPLocalIPAddrGet (void)
- void lwIPLocalMACGet (unsigned char *pucMAC)
- unsigned long lwIPLocalNetMaskGet (void)
- void lwIPNetworkConfigChange (unsigned long ulIPAddr, unsigned long ulNetMask, unsigned long ulGWAddr, unsigned long ulIPMode)
- static void lwIPSoftMDIXTimer (void *pvArg)

19.2.1 Function Documentation

19.2.1.1 lwIPEthernetIntHandler

Handles Ethernet interrupts for the lwIP TCP/IP stack.

Prototype:

```
void
lwIPEthernetIntHandler(void)
```

Description:

This function handles Ethernet interrupts for the lwIP TCP/IP stack. At the lowest level, all receive packets are placed into a packet queue for processing at a higher level. Also, the transmit packet queue is checked and packets are drained and transmitted through the Ethernet MAC as needed. If the system is configured without an RTOS, additional processing is performed at the interrupt level. The packet queues are processed by the lwIP TCP/IP code, and lwIP periodic timers are serviced (as needed).

Returns:

None.

19.2.1.2 lwIPInit

Initializes the lwIP TCP/IP stack.

Prototype:

```
void
lwIPInit(const unsigned char *pucMAC,
         unsigned long ulIPAddr,
         unsigned long ulNetMask,
         unsigned long ulGWAddr,
         unsigned long ulIPMode)
```

Parameters:

pucMAC is a pointer to a six byte array containing the MAC address to be used for the interface.

ulIPAddr is the IP address to be used (static).

ulNetMask is the network mask to be used (static).

ulGWAddr is the Gateway address to be used (static).

ulIPMode is the IP Address Mode. **IPADDR_USE_STATIC** will force static IP addressing to be used, **IPADDR_USE_DHCP** will force DHCP with fallback to Link Local (Auto IP), while **IPADDR_USE_AUTOIP** will force Link Local only.

Description:

This function performs initialization of the lwIP TCP/IP stack for the Stellaris Ethernet MAC, including DHCP and/or AutoIP, as configured.

Returns:

None.

19.2.1.3 lwIPLocalGWAddrGet

Returns the gateway address for this interface.

Prototype:

```
unsigned long
lwIPLocalGWAddrGet(void)
```

Description:

This function will read and return the currently assigned gateway address for the Stellaris Ethernet interface.

Returns:
the assigned gateway address for this interface.

19.2.1.4 lwIPLocalIPAddrGet

Returns the IP address for this interface.

Prototype:
```
unsigned long
lwIPLocalIPAddrGet(void)
```

Description:
This function will read and return the currently assigned IP address for the Stellaris Ethernet interface.

Returns:
Returns the assigned IP address for this interface.

19.2.1.5 lwIPLocalMACGet

Returns the local MAC/HW address for this interface.

Prototype:
```
void
lwIPLocalMACGet(unsigned char *pucMAC)
```

Parameters:
pucMAC is a pointer to an array of bytes used to store the MAC address.

Description:
This function will read the currently assigned MAC address into the array passed in *pucMAC*.

Returns:
None.

19.2.1.6 lwIPLocalNetMaskGet

Returns the network mask for this interface.

Prototype:
```
unsigned long
lwIPLocalNetMaskGet(void)
```

Description:
This function will read and return the currently assigned network mask for the Stellaris Ethernet interface.

Returns:
the assigned network mask for this interface.

19.2.1.7 lwIPNetworkConfigChange

Change the configuration of the lwIP network interface.

Prototype:

```
void
lwIPNetworkConfigChange(unsigned long ulIPAddr,
                        unsigned long ulNetMask,
                        unsigned long ulGWAddr,
                        unsigned long ulIPMode)
```

Parameters:

ulIPAddr is the new IP address to be used (static).

ulNetMask is the new network mask to be used (static).

ulGWAddr is the new Gateway address to be used (static).

ulIPMode is the IP Address Mode. **IPADDR_USE_STATIC** 0 will force static IP addressing to be used, **IPADDR_USE_DHCP** will force DHCP with fallback to Link Local (Auto IP), while **IPADDR_USE_AUTOIP** will force Link Local only.

Description:

This function will evaluate the new configuration data. If necessary, the interface will be brought down, reconfigured, and then brought back up with the new configuration.

Returns:

None.

19.2.1.8 lwIPSoftMDIXTimer [static]

Service the lwIP timers.

Prototype:

```
static void
lwIPSoftMDIXTimer(void *pvArg)
```

Description:

This function services all of the lwIP periodic timers, including TCP and Host timers. This should be called from the lwIP context, which may be the Ethernet interrupt (in the case of a non-RTOS system) or the lwIP thread, in the event that an RTOS is used.

Returns:

None.

19.3 Programming Example

The following example shows how to use the lwIP wrapper module to initialize the lwIP stack.

```
unsigned char pucMACArray[6];

//
// Fill in the MAC array and initialize the lwIP library using DHCP.
```

```
//
lwIPInit(pucMACArray, 0, 0, 0, IPADDR_USE_DHCP);

//
// Periodically call the lwIP timer tick.  In a real application, this
// would use a timer interrupt instead of an endless loop.
//
while(1)
{
    SysCtlDelay(1000);
    lwIPTimer(1);
}
```

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

Functions

- long sine (unsigned long ulAngle)

20.2.1 Function Documentation

20.2.1.1 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**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265