RDK-STEPPER Firmware Development Package

USER'S GUIDE

TEXAS INSTRUMENTS

SW-RDK-STEPPER-UG-9453

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 9453 of this document, last updated on September 05, 2012.

Table of Contents

1 Introduction

1.1 Overview

This document provides a Software Reference Manual (SRM) for the Stepper Motor Drive Reference Design Kit (Stepper RDK). The Stepper RDK consists of a board containing a Stellaris® microcontroller and motor drive electronics, a stepper motor, and firmware to run the motor. This SRM refers to the firmware that runs on the Stepper RDK microcontroller.

The firmware runs on a Stellaris LM3S617 microcontroller, utilizing the six channel motion control PWM module, five of the six ADC channels, a comparator and UART. The Stellaris Peripheral Driver Library is used to configure and operate the peripherals.

The Stepper motor application has the following features:

- Drives bipolar stepper motors up to 10,000 steps per second
- Operates at bus voltages up to 80 volts
- Full or half stepping
- Slow and fast current decay modes
- Chopper, open-loop or closed-loop PWM control modes

See the Stellaris Stepper 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 11 KB of flash and 1.3 KB of SRAM. Of this, 3.6 KB of flash and 0.5 KB of SRAM is consumed by the user interface, which represents the application. These numbers are from the Keil/ARM RV-MDK toolchain. The GNU toolchain produces numbers approximately 30% larger.

1.3 Memory Layout

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

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

The 32 KB of flash on the LM3S617 is organized as follows:

`0x0000.0000`
`0x0000.07ff`
	Boot Loader

`0x0000.0800`
`0x0000.6fff`
	Stepper Firmware

`0x0000.7000`
`0x0000.7fff`
	Parameter Storage

1.4 Fixed Point Data Format

The Stepper RDK firmware makes use of a fixed point data format for several variables and function parameters. Throughout the documentation, this is referred to as "24.8" format. This is a way to represent a real number with whole and fractional parts without resorting to using floating point math. All math operations with 24.8 fixed-point numbers are accomplished using only integer instructions. The format "24.8" means that the upper 24 bits are the whole part of the number and the lower 8 bits are the fractional part. Since 8 bits are used as the fraction, the resolution of the data format is 1/256 units.

The following equation shows how to derive the floating point value from a 24.8 fixed-point number:

```
valFloat = (val248 >> 8) + ((val248 & 0xFF) / 256)
```

The following equation shows how to derive the fixed-point value from a floating point number:

```
val248 = valFloat * 256
```

Note that you should try to avoid using floating point numbers in your application if possible because this increases the overall code size and can reduce performance.

1.5 Design

The Stepper RDK firmware is designed to show how stepper motor control can be done using a Stellaris microcontroller and motor drive circuits. All of the stepper sequencing is performed in the microcontroller, no external stepping logic is required. The Stepper RDK firmware provides maximum flexibility by implementing three different methods for controlling the stepper motor winding current. It also provides a number of user adjustable parameters which allows experimentation to find the best settings for use in a particular application.

The firmware is implemented as several layers, with a well defined API to make it easy to integrate with a user's application code. It includes, as the application, a user interface which is used for demonstrating how to use the Stepper firmware. The user interface can be used initially for experimenting with the motor and settings, and can later be replaced by the real application software.

There are two user interfaces: an on-board interface, and an off-board interface. The on-board interface uses a potentiometer knob, a button, and two LEDs to operate the stepper motor. The off-board interface uses a USB-connected serial port and a PC hosted graphical application which provided extensive control of the firmware. Refer to the *Stepper RDK User's Manual* for details of using the on-board and off-board interfaces.

Design/Layer Overview

The following diagram shows the firmware layers:

```
+-----------------------+---+
| Main / User Interface | S |
|-----------------------| t |
|      Stepper API      | e |
|-----------------------| p |
|    Step Sequencer     | c |
|-----------------------| f |
|   Step Controller     | g |
+-----------------------+---+
```

Main / User Interface

This layer represents the application software. It provides an interface to the external world so that the user can control the stepper motor. It makes calls down into the Stepper API in order to carry out the requested motor operations, and to read and report the status of the motor back to the user.

Stepper API

This layer provides a programming interface for the application software. It provides functions for configuring and controlling the stepper motor, and for reading the status of the system. Normally, the functions provided in this layer are all that is needed by the application software.

Step Sequencer

This layer provides the logic and timing necessary to generate the stepping sequence. Whenever a stepper motion is requested, it calculates a speed profile to satisfy the motion request. The speed profile consists of an acceleration ramp up to the running speed, and a deceleration ramp down to zero speed, at the target position. For maximum flexibility, this part is designed to accommodate new motion requests on the fly, which means that the speed profile may be recomputed while the motor is running, and allows the motor to change from one speed to another while moving to the target position. The Step Sequencer uses a timer to time the intervals between steps. At each step time, it determines what current should be applied to each of the two motor windings (A and B), and makes function calls to the Step Controller layer which controls the current in the windings.

Step Controller

This layer provides the logic for controlling the current in each of the two motor windings (A and B). After the Step Sequencer determines what current should be applied to each winding, the Step Controller will set the hardware signals that control the motor drive electronics as needed to control the current in the winding. The Step Controller can use PWM or chopping methods to control the amount of current in the winding, and can use fast or slow current decay modes.

Step Config (Stepcfg)

The Step Config layer provides configuration definitions for the firmware, and is available to all the other layers. It provides definition of items such as which A/D channels to use for current monitoring, and which timers to use for timing intervals needed for stepping. This is really just a header file containing macros, and provides one location for allocating microcontroller hardware resources.

2 Applications

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

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

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

2.1 Boot Loader (boot_serial)

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

2.2 Stepper Motor Drive Application (qs-stepper)

This application is a motor drive for bipolar stepper motors. The following features are available:

- Full and Half Stepping
- Slow and Fast Current Decay
- Chopping or PWM current control modes
- Adjustable Drive and Holding Current
- Stepping rates up to 10,000 steps/second for suitable motors
- DC bus voltage monitoring and CPU temperature monitoring
- Overcurrent fault protection.
- Simple on-board user interface (via a potentiometer and push button)
- Comprehensive serial user interface
- Several configurable drive parameters
- Persistent storage of drive parameters in flash

3 Development System Utilities

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

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

Serial Flash Downloader

Usage:

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

Description:

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

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

Arguments:

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

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

-d disables auto-baud.

-h displays usage information.

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

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

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

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

INPUT FILE specifies the name of the firmware image file.

Example:

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

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

4 Stepper API

4.1 Introduction

This is the main API used for controlling the stepper motor. This API can be used for configuring the stepper modes and parameters, and commanding the motor to move. The API is also used to retrieve status such as current motor position and speed.

This API should be used by the application as the main interface to the stepper motor control.

Most of the functions in this API are passed through to lower level modules in order to carry out the action or get status.

The function StepperInit() should be called once during system initialization, to initialize the stepper API module.

The functions StepperEnable() and StepperDisable() are used for enabling and disabling the motor. The stepper motor will not run until it has been enabled. The function StepperEmergencyStop() can be used when the motor needs to be stopped right away.

The function StepperSetMotion() is the main function used for commanding the stepper motor to move. This is used to set the position, speed, and acceleration values for the motor.

The function StepperGetMotorStatus() is used to retrieve status information about the motor, such as the current position and speed.

The following functions are used for configuring various parameters used for the motor operation: StepperSetControlMode(), StepperSetStepMode(), StepperSetDecayMode(), StepperSetPWMFreq(), StepperSetFixedOnTime(), StepperSetBlankingTime(), StepperSetMotorParms(), StepperSetFaultParms().

The code for implementing the stepper API is contained in `stepper.c`, with `stepper.h` containing the definitions for the variables and functions exported to the application.

4.2 Definitions

Data Structures

- tStepperStatus

Functions

- void StepperClearFaults (void)
- void StepperCompIntHandler (void)
- void StepperDisable (void)

- void StepperEmergencyStop (void)
- void StepperEnable (void)
- tStepperStatus ∗ StepperGetMotorStatus (void)
- void StepperInit (void)
- void StepperResetPosition (long lNewPosition)
- void StepperSetBlankingTime (unsigned short usBlankOffTime)
- void StepperSetControlMode (unsigned char ucControlMode)
- void StepperSetDecayMode (unsigned char ucDecayMode)
- void StepperSetFaultParms (unsigned short usFaultCurrent)
- void StepperSetFixedOnTime (unsigned short usFixedOnTime)
- void StepperSetMotion (long lPos, unsigned short usSpeed, unsigned short usAccel, unsigned short usDecel)
- void StepperSetMotorParms (unsigned short usDriveCurrent, unsigned short usHoldCurrent, unsigned short usBusVoltage, unsigned short usDriveResistance)
- void StepperSetPWMFreq (unsigned short usPWMFreq)
- void StepperSetStepMode (unsigned char ucStepMode)

Variables

- tStepperStatus sStepperStatus

4.2.1 Data Structure Documentation

4.2.1.1 tStepperStatus

Definition:

```
typedef struct
{
    long lPosition;
    long lTargetPos;
    unsigned char ucMotorStatus;
    unsigned char ucStepMode;
    unsigned char ucControlMode;
    unsigned short usPWMFrequency;
    unsigned short usSpeed;
    unsigned short usCurrent[2];
    unsigned char ucFaultFlags;
    unsigned char bEnabled;
}
tStepperStatus
```

Members:

lPosition The current position of the motor in fixed-point 24.8 format.

lTargetPos The target position of the motor in fixed-point 24.8 format.

ucMotorStatus The status of the motor represented as one of the following values: MOTOR_STATUS_STOP, MOTOR_STATUS_RUN, MOTOR_STATUS_ACCEL, or MOTOR_STATUS_DECEL

ucStepMode The stepping mode of the motor represented as one of the following values: STEP_MODE_FULL, STEP_MODE_WAVE, STEP_MODE_HALF, or STEP_MODE_MICRO.

ucControlMode The control mode of the motor represented as one of the following values: CONTROL_MODE_OPENPWM, CONTROL_MODE_CLOSEDPWM or CONTROL_MODE_CHOP.

usPWMFrequency The PWM frequency setting in Hz.

usSpeed The speed of the motor in whole steps per second.

usCurrent An array holding the current for each winding in milliamps. Winding A is at index 0, and Winding B is at index 1.

ucFaultFlags The flags indicating a fault. Bit 0 is overcurrent.

bEnabled A flag indicating if the motor is enabled.

Description:
This is a structure used to hold status information about the stepper motor.

4.2.2 Function Documentation

4.2.2.1 StepperClearFaults

Clears the fault flags.

Prototype:
```
void
StepperClearFaults(void)
```

Description:
This function clears the fault flags, which will allow the motor to run again, after a fault occurred.

Returns:
None.

4.2.2.2 StepperCompIntHandler

Interrupt handler for the comparator interrupt.

Prototype:
```
void
StepperCompIntHandler(void)
```

Description:
This interrupt handler is triggered when the comparator trips. The comparator is set to trip when the combined winding current goes above a certain value. When this happens, the fault signal will be asserted, which will automatically place the hardware in a safe state.

In this function, the motor is stopped immediately and placed in a safe state, and the over current fault flag is set.

Returns:
None.

4.2.2.3 StepperDisable

Disable the stepper for running.

Prototype:

```
void
StepperDisable(void)
```

Description:
This function disables the stepper motor for running. If the motor is currently moving, it will be gracefully stopped. After that, all motion commands will be ignored until StepperEnable() is called again.

Returns:
None.

4.2.2.4 StepperEmergencyStop

Immediately stop and place the motor in a safe state.

Prototype:

```
void
StepperEmergencyStop(void)
```

Description:
This function disables all the motor control signals immediately. This is not a graceful stop, and the position information will be lost. The motor will be left in the disabled state.

Returns:
None.

4.2.2.5 StepperEnable

Enable the stepper for running.

Prototype:

```
void
StepperEnable(void)
```

Description:
This function enables the stepper motor for running. All motion commands are ignored if the stepper is not enabled. The motor cannot be enabled if there are any pending faults. The function StepperClearFaults() must be called first.

Returns:
None.

4.2.2.6 StepperGetMotorStatus

Get the status of the motor.

Prototype:
```
tStepperStatus *
StepperGetMotorStatus(void)
```

Description:
This function returns a pointer to a structure that holds the motor status. Status items include position, speed, etc. This function must be called in order to update the data items in the structure.

Returns:
A pointer to the structure containing the status information.

4.2.2.7 StepperInit

Initializes the stepper control module.

Prototype:
```
void
StepperInit(void)
```

Description:
This function sets up the stepper software, and initializes the hardware necessary for control of the stepper. This should be called just once when the system is initialized. It will call the Init functions for all lower modules.

Returns:
None.

4.2.2.8 StepperResetPosition

Sets the value of the current position.

Prototype:
```
void
StepperResetPosition(long lNewPosition)
```

Parameters:
lNewPosition is the new position in 24.8 format.

Description:
This function can be used to initialize or reset the current known position. Whatever value is passed, the current position will be updated to match, without moving the motor. This can be used when "homing" the motor. For example, if the position is known due to a limit switch, then the position could be reset to that known value.

The value of the new position is restricted to whole steps, and any fractional portion will be truncated.

This function can only be used when the motor is not moving. If the motor is moving then the new setting is ignored.

The parameter *lNewPosition* is a signed number representing the motor position in fixed-point 24.8 format. The upper 24 bits are the (signed) whole step position, while the lower 8 bits are the fractional step position. While this allows for a theoretical resolution of 1/256 step size, the motor does not actually support micro-steps that small. The value of the lower 8 bits (the fractional step) is the fractional value multiplied by 256. For example, the lower 8 bits of a half-step is 0x80 ($0.5 * 256$). Likewise, a quarter step is 0x40 ($0.25 * 256$). In order to use half-steps you must be using half- or micro-stepping mode. In order to use fractional steps smaller than half, you must be using micro-stepping mode.

Returns:
None.

4.2.2.9 StepperSetBlankingTime

Set the off blanking interval for chopper control mode.

Prototype:
```
void
StepperSetBlankingTime(unsigned short usBlankOffTime)
```

Parameters:
usBlankOffTime is the off blanking time in microseconds.

Description:
This function is used to set the off blanking time used in the chopper mode. The off blanking time is the amount of time that the winding is kept off after the chopper turns the winding off, before turning it on again. This is actually the minimum time, since the chopper may dynamically lengthen the off blanking time if needed to keep the current from rising too much.

Calls to this function take effect immediately.

Returns:
None.

4.2.2.10 StepperSetControlMode

Set the current control mode for the motor windings.

Prototype:
```
void
StepperSetControlMode(unsigned char ucControlMode)
```

Parameters:
ucControlMode is the current control method, CONTROL_MODE_OPENPWM, CONTROL_-MODE_CLOSEDPWM, or CONTROL_MODE_CHOPPER.

Description:
This function is used to set the method that is used to control the current in the winding.

The open-loop PWM method applies voltage to the winding for a fixed amount of time (fixed on time), and then switches to PWM. This allows the current to rise rapidly in the winding before the PWM starts. The fixed on time can be minimal which is the same as using PWM without a fixed rise time.

The chopper method monitors the winding current and turns the H-bridge on and off to maintain the desired current.

The closed-loop PWM method measures the current during the PWM pulse and adjusts the duty cycle to maintain the current at the desired level.

Calls to this function only take effect if the motor is not running. Otherwise, the setting is ignored.

Returns:
None.

4.2.2.11 StepperSetDecayMode

Sets the current decay mode.

Prototype:
```
void
StepperSetDecayMode(unsigned char ucDecayMode)
```

Parameters:
ucDecayMode is the current decay mode, DECAY_MODE_FAST or DECAY_MODE_SLOW.

Description:
This function is used to set the current decay mode used when the motor winding is switched off during chopping or PWM control. Slow decay mode closes both low-side switches on the H-bridge, which allows the current in the winding to recirculate and decay slowly. Fast decay mode opens all the switches on the winding so that the current cannot recirculate and decays rapidly.

Calls to this function take effect immediately.

Returns:
None.

4.2.2.12 StepperSetFaultParms

Sets the fault current level that is used for hardware fault control.

Prototype:
```
void
StepperSetFaultParms(unsigned short usFaultCurrent)
```

Parameters:
usFaultCurrent is the fault current in milliamps.

Description:

This function is used to set the comparator that will be triggered if the current rises above a certain level. If this happens, then the comparator will trigger a fault condition, independent of software, and will shut off all the control signals to the motor.

Returns:

None.

4.2.2.13 StepperSetFixedOnTime

Sets the fixed-on interval when using open-loop PWM control mode.

Prototype:

```
void
StepperSetFixedOnTime(unsigned short usFixedOnTime)
```

Parameters:

usFixedOnTime is the time the winding will remain on before PWM in microseconds.

Description:

The fixed-on time is the amount of time that the winding will be left turned on at the beginning of the step, in order to let the current rise as fast as possible. At the end of the fixed-on time, the control signal switches to PWM.

Calls to this function will take effect immediately.

Returns:

None.

4.2.2.14 StepperSetMotion

Sets new position and motion parameters for the stepper, and initiates motion (if enabled).

Prototype:

```
void
StepperSetMotion(long lPos,
                 unsigned short usSpeed,
                 unsigned short usAccel,
                 unsigned short usDecel)
```

Parameters:

lPos is the target position for the stepper in 24.8 format

usSpeed is the running speed in steps/second, that should be used when the motor is moving

usAccel is the acceleration in steps/second∗∗2, that is used when the motor is accelerated to the running speed

usDecel is the deceleration in steps/second∗∗2, that is used when decelerating the motor from the running speed to a stop

Description:

This function will start the stepper motor moving to the target position specified by lPos. The acceleration and deceleration parameters will be used to create a speed profile that will be used as the motor runs.

If the stepper has been enabled by a prior call to StepperEnable(), then this function will take effect immediately. If the stepper is not enabled, or has been stopped by StepperDisable(), then this function will have no effect. If the motor is already moving, then the speed profile is recalculated, and the motor speed adjusted if necessary.

The parameter *lPos* is a signed number representing the motor position in fixed-point 24.8 format. The upper 24 bits are the (signed) whole step position, while the lower 8 bits are the fractional step position. While this allows for a theoretical resolution of 1/256 step size, the motor does not actually support micro-steps that small. The value of the lower 8 bits (the fractional step) is the fractional value multiplied by 256. For example, the lower 8 bits of a half-step is 0x80 (0.5 $*$ 256). Likewise, a quarter step is 0x40 (0.25 $*$ 256). In order to use half-steps you must be using half- or micro-stepping mode. In order to use fractional steps smaller than half, you must be using micro-stepping mode.

Returns:
None.

4.2.2.15 StepperSetMotorParms

Sets the motor winding parameters used for controlling winding current.

Prototype:
```
void
StepperSetMotorParms(unsigned short usDriveCurrent,
                     unsigned short usHoldCurrent,
                     unsigned short usBusVoltage,
                     unsigned short usDriveResistance)
```

Parameters:

usDriveCurrent is the drive current in milliamps. The control method will attempt to drive the winding current to this value when stepping.

usHoldCurrent is the holding current in milliamps. The control method will attempt to maintain this current in the winding when the motor is stopped.

usBusVoltage is the bus voltage used for driving the motor in millivolts.

usDriveResistance is the stepper winding resistance in milliohms.

Description:
This function will set the parameters associated with the current used to drive the motor windings. The drive current is applied to the windings when the motor is stepping. The hold current is applied to the active windings when the motor is stopped. The holding current can be zero. The resistance is the winding resistance from the motor specification. The resistance and bus voltage is used for calculating duty cycle when using a PWM control method.

Calls to this function will take effect immediately. It is up to the caller to ensure that the current values specified are within safe limits for the motor, and that the actual bus voltage does not exceed the value specified by usBusVoltage.

Returns:
None.

4.2.2.16 StepperSetPWMFreq

Sets the PWM frequency used for PWM control modes.

Prototype:

```
void
StepperSetPWMFreq(unsigned short usPWMFreq)
```

Parameters:

usPWMFreq is the PWM frequency in Hz.

Description:

This function will set PWM frequency for PWM-based control modes.

This function will only take effect if the motor is stopped. Otherwise, it has no effect.

Returns:

None.

4.2.2.17 StepperSetStepMode

Sets the stepping mode step size.

Prototype:

```
void
StepperSetStepMode(unsigned char ucStepMode)
```

Parameters:

ucStepMode is the stepping size, STEP_MODE_FULL, STEP_MODE_WAVE, STEP_-MODE_HALF, or STEP_MODE_MICRO.

Description:

This function is used to set the step size to full, half or micro steps.

Calls to this function only take effect if the motor is not running. Otherwise, the setting is ignored.

Returns:

None.

4.2.3 Variable Documentation

4.2.3.1 sStepperStatus

Definition:

```
tStepperStatus sStepperStatus
```

Description:

Holds the current status of the motor. The fields are updated when the StepperGetMotor-Status() function is called, and a pointer to this is returned to the caller.

5 Stepper Configuration

5.1 Introduction

The `stepcfg.h` file contains all the configuration macros for assigning the microcontroller resources used by the Stepper RDK firmware.

5.2 Definitions

Defines

- ADC_INT_PRI
- BUSV_ADC_CHAN
- COMP_INT_PRI
- CONTROL_MODE_CHOP
- CONTROL_MODE_CLOSEDPWM
- CONTROL_MODE_OPENPWM
- COUNTS2MILLIAMPS(c)
- DECAY_MODE_FAST
- DECAY_MODE_SLOW
- FAULT_FLAG_CURRENT
- FIXED_TMR_BASE
- FIXED_TMR_INT_PRI
- FLASH_PB_END
- FLASH_PB_SIZE
- FLASH_PB_START
- MILLIAMPS2COUNTS(m)
- MODE_LED_PIN_NUM
- MODE_LED_PORT
- POT_ADC_CHAN
- STATUS_LED_PIN_NUM
- STATUS_LED_PORT
- STEP_MODE_FULL
- STEP_MODE_HALF
- STEP_MODE_MICRO
- STEP_MODE_WAVE
- STEP_TMR_BASE
- STEP_TMR_INT_PRI
- SYSTEM_CLOCK

- SYSTICK_INT_PRI
- UI_ADC_SEQUENCER
- UI_SER_INT_PRI
- USER_BUTTON_GPIO_PERIPH
- USER_BUTTON_PIN_NUM
- WINDING_A_ADC_CHANNEL
- WINDING_A_ADC_SEQUENCER
- WINDING_B_ADC_CHANNEL
- WINDING_B_ADC_SEQUENCER
- WINDING_ID_A
- WINDING_ID_B

5.2.1 Define Documentation

5.2.1.1 ADC_INT_PRI

Definition:
```
#define ADC_INT_PRI
```

Description:
Defines the interrupt priority for the ADC interrupt for current sampling.

5.2.1.2 BUSV_ADC_CHAN

Definition:
```
#define BUSV_ADC_CHAN
```

Description:
Defines the ADC channel to be used for the bus voltage.

5.2.1.3 COMP_INT_PRI

Definition:
```
#define COMP_INT_PRI
```

Description:
Defines the interrupt priority for the current fault comparator.

5.2.1.4 CONTROL_MODE_CHOP

Definition:
```
#define CONTROL_MODE_CHOP
```

Description:
Defines a value indicating that the chopper current control method should be used.

5.2.1.5 CONTROL_MODE_CLOSEDPWM

Definition:

```
#define CONTROL_MODE_CLOSEDPWM
```

Description:

Defines a value meaning that the Closed-loop PWM current control method should be used.

5.2.1.6 CONTROL_MODE_OPENPWM

Definition:

```
#define CONTROL_MODE_OPENPWM
```

Description:

Defines a value meaning that the Open-loop PWM current control method should be used.

5.2.1.7 COUNTS2MILLIAMPS

Computes the current in milliamps from raw ADC counts.

Definition:

```
#define COUNTS2MILLIAMPS(c)
```

Parameters:

c is the current in raw ADC counts.

Description:

This macro will convert a value of current in raw ADC counts as read from the A/D converter, to the value in milliamps.

Returns:

Returns the value of the current in units of milliamps.

5.2.1.8 DECAY_MODE_FAST

Definition:

```
#define DECAY_MODE_FAST
```

Description:

Defines a value indicating that the fast decay mode should be used.

5.2.1.9 DECAY_MODE_SLOW

Definition:

```
#define DECAY_MODE_SLOW
```

Description:

Defines a value indicating that the slow decay mode should be used.

5.2.1.10 FAULT_FLAG_CURRENT

Definition:

```
#define FAULT_FLAG_CURRENT
```

Description:

Defines the fault flag indicating an over-current fault occurred.

5.2.1.11 FIXED_TMR_BASE

Definition:

```
#define FIXED_TMR_BASE
```

Description:

Defines the timer used for the fixed-interval timer.

5.2.1.12 FIXED_TMR_INT_PRI

Definition:

```
#define FIXED_TMR_INT_PRI
```

Description:

Defines the interrupt priority for the fixed-interval timers.

5.2.1.13 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.

5.2.1.14 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 uiparms.h file).

5.2.1.15 FLASH_PB_START

Definition:

```
#define FLASH_PB_START
```

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

5.2.1.16 MILLIAMPS2COUNTS

Computes the raw ADC counts that represent a current value.

Definition:

```
#define MILLIAMPS2COUNTS(m)
```

Parameters:
m is the current in milliamps.

Description:
This macro will convert a value of current in milliamps to the value in ADC counts. This is used to get the ADC value that represents the threshold current for chopper mode.

Returns:
Returns the value of the current in units of ADC counts.

5.2.1.17 MODE_LED_PIN_NUM

Definition:

```
#define MODE_LED_PIN_NUM
```

Description:
Defines the GPIO pin for the status LED.

5.2.1.18 MODE_LED_PORT

Definition:

```
#define MODE_LED_PORT
```

Description:
Defines the GPIO port for the mode LED.

5.2.1.19 POT_ADC_CHAN

Definition:

```
#define POT_ADC_CHAN
```

Description:
Defines the ADC channel to be used for the potentiometer.

5.2.1.20 STATUS_LED_PIN_NUM

Definition:

```
#define STATUS_LED_PIN_NUM
```

Description:
Defines the GPIO pin for the status LED.

5.2.1.21 STATUS_LED_PORT

Definition:

```
#define STATUS_LED_PORT
```

Description:
Defines the GPIO port for the status LED.

5.2.1.22 STEP_MODE_FULL

Definition:

```
#define STEP_MODE_FULL
```

Description:
Defines a value meaning that full (normal) stepping should be used.

5.2.1.23 STEP_MODE_HALF

Definition:

```
#define STEP_MODE_HALF
```

Description:
Defines a value meaning that half stepping should be used.

5.2.1.24 STEP_MODE_MICRO

Definition:

```
#define STEP_MODE_MICRO
```

Description:
Defines a value meaning that micro stepping should be used.

5.2.1.25 STEP_MODE_WAVE

Definition:

```
#define STEP_MODE_WAVE
```

Description:
Defines a value meaning that wave drive stepping should be used. Wave drive is whole stepping, with one winding on at a time.

5.2.1.26 STEP_TMR_BASE

Definition:

```
#define STEP_TMR_BASE
```

Description:
Defines the timer used for the step timer.

5.2.1.27 STEP_TMR_INT_PRI

Definition:

```
#define STEP_TMR_INT_PRI
```

Description:
Defines the interrupt priority for the step timer.

5.2.1.28 SYSTEM_CLOCK

Definition:

```
#define SYSTEM_CLOCK
```

Description:
Defines the processor clock frequency. In order to change the processor clock, the initialization code in main() must be changed, and this macro must be changed to match.

5.2.1.29 SYSTICK_INT_PRI

Definition:

```
#define SYSTICK_INT_PRI
```

Description:
Defines the interrupt priority for the system tick timer.

5.2.1.30 UI_ADC_SEQUENCER

Definition:

```
#define UI_ADC_SEQUENCER
```

Description:
Defines the ADC sequencer to be used for measurements in the user interface.

5.2.1.31 UI_SER_INT_PRI

Definition:

```
#define UI_SER_INT_PRI
```

Description:
Defines the interrupt priority for the serial user interface and UART.

5.2.1.32 USER_BUTTON_GPIO_PERIPH

Definition:

```
#define USER_BUTTON_GPIO_PERIPH
```

Description:

Defines the GPIO port for the user button.

5.2.1.33 USER_BUTTON_PIN_NUM

Definition:

```
#define USER_BUTTON_PIN_NUM
```

Description:

Defines the GPIO pin for the user button.

5.2.1.34 WINDING_A_ADC_CHANNEL

Definition:

```
#define WINDING_A_ADC_CHANNEL
```

Description:

Defines the ADC channel for winding A current sense.

5.2.1.35 WINDING_A_ADC_SEQUENCER

Definition:

```
#define WINDING_A_ADC_SEQUENCER
```

Description:

Defines the ADC sequencer to be used for chopping winding A.

5.2.1.36 WINDING_B_ADC_CHANNEL

Definition:

```
#define WINDING_B_ADC_CHANNEL
```

Description:

Defines the ADC channel for winding B current sense.

5.2.1.37 WINDING_B_ADC_SEQUENCER

Definition:

```
#define WINDING_B_ADC_SEQUENCER
```

Description:

Defines the ADC sequencer to be used for chopping winding B.

5.2.1.38 WINDING_ID_A

Definition:

```
#define WINDING_ID_A
```

Description:

Defines the index value for winding A. This is used in places as an index for looking up values needed for winding A.

5.2.1.39 WINDING_ID_B

Definition:

```
#define WINDING_ID_B
```

Description:

Defines the index value for winding B. This is used in places as an index for looking up values needed for winding B.

6 Step Sequencer

6.1 Introduction

The Step Sequencer module is used for generating the steps in the correct sequence, and with the correct timing to cause the motor to run in the desired speed and direction. Whenever any of the driving parameters (position, speed, accel, decel) is changed, a speed profile is computed which will cause the motor to accelerate to the running speed, and then decelerate to a stop at the target position. As the motor is running, the time between steps is calculated in real time in order to achieve the acceleration and deceleration ramps.

Normally, there is no reason that an application needs to call any of these functions directly, nor make direct access to any of the module global variables. The following explains how this module is used by the Stepper API module.

First, the module is initialized by calling StepSeqInit(). Then, the following functions are called to configure the operation of the stepper motor: StepSeqControlMode(), StepSeqStepMode(), StepSeqDecayMode(), and StepSeqCurrent(). These are used to set up the winding current control method, the stepping mode, the decay mode, and the driving current.

When it is time to make the stepper motor move, the function StepSeqMove() is called. This function will compute the speed profile based on the input parameters, and the current motor status. If the motor is already running, then a new speed profile is calculated to satisfy the new motion request, which can include real-time speed changes.

If the motor is running, it can be stopped gracefully by calling StepSeqStop(). This will decelerate the motor from whatever speed it is running to a full stop using the last specified deceleration value. When the motor is stopped this way, position knowledge is retained. If the motor must be stopped immediately, then StepSeqShutdown() can be used. This function will immediately disable the motor control signals and stop the stepping sequence. When this happens, the known position information may no longer be accurate.

The StepSeqHandler() function is the interrupt handler for the step timer. It is called at each step or half step time to generate the next step in the sequence.

Optimizations

The StepSeqMove() function is somewhat complicated in order to handle all of the different cases of the new motion requested and the current status of the motor. The simple case is when the motor is stopped. Calculating a new speed profile is easy in this case. But if the motor is already moving, then it becomes more complicated. For example, the motor may need to speed up or slow down from one running speed to a new running speed, as well as change the target position. Or, the new target position may be in the opposite direction from the present direction, which means the motor needs to be decelerated to a stop, and then run in the other direction.

In a real application, it may be reasonable to put some constraints on the need to dynamically change the motion parameters of a running motor. For example, it may be completely reasonable to require that the motor be stopped before a new motion command is issued. In this case, a large amount of the code in StepSeqMove() could be removed to save code space and complexity. This is a potential optimization that could be made by the user.

An optimization that was made in the code is the the use of direct access to peripheral registers instead of making calls to the DriverLib peripheral library. This is done in the step timer interrupt handler in order to make the interrupt code execute just a bit faster. Whenever a register access is made like this, there is a comment in the code showing the equivalent DriverLib call.

It is likely that an actual stepper application will use only one of the three current control methods (Open-loop PWM, closed-loop PWM, or chopper mode). In this case, the user could modify this module to remove the code that is related to the unused control method. One of the functions such as StepSeqUsingOpenPwm() or StepSeqUsingChop() could be removed along with conditional code that refers to these functions. These changes would make the resulting code somewhat smaller and more efficient to execute.

Step Sequencing

The Step Sequencer determines how the current must be set in each of the motor windings (A and B) in order to cause the motor to step in the correct direction, and must make each step at a certain time in order to make the motor run at the correct speed.

A step sequence table like that shown below is used to determine how much current should be applied to each winding at each point in the stepping sequence. There are 4 full steps for a complete stepping cycle. In an ideal environment, the current waveform to drive the motor through one full cycle (of 4 full steps) is a sine wave in each winding, 90 degrees out of phase. This waveform is represented as a series of "micro-steps" in the step sequence table. The table covers 4 full steps, each further divided into 8 micro-steps, for a total of 32 micro-steps in a full stepping cycle.

```
           Stepping Sequence
        +----------------------+
        |Step |    A       B   |
        |----------------------|
  F-->  | 0-4 |  46341   46341 |
        | 0-5 |  54491   36410 |
        | 0-6 |  60547   25080 |
        | 0-7 |  64277   12785 |
  H-->  | 1-0 |  65536       0 |
        | 1-1 |  64277  -12785 |
        | 1-2 |  60547  -25080 |
        | 1-3 |  54491  -36410 |
  F-->  | 1-4 |  46341  -46341 |
        | 1-5 |  36410  -54491 |
        | 1-6 |  25080  -60547 |
        | 1-7 |  12785  -64277 |
  H-->  | 2-0 |      0  -65536 |
        | 2-1 | -12785  -64277 |
        | 2-2 | -25080  -60547 |
        | 2-3 | -36410  -54491 |
  F-->  | 2-4 | -46341  -46341 |
        | 2-5 | -54491  -36410 |
        | 2-6 | -60547  -25080 |
        | 2-7 | -64277  -12785 |
  H-->  | 3-0 | -65536       0 |
        | 3-1 | -64277   12785 |
        | 3-2 | -60547   25080 |
        | 3-3 | -54491   36410 |
  F-->  | 3-4 | -46341   46341 |
        | 3-5 | -36410   54491 |
        | 3-6 | -25080   60547 |
        | 3-7 | -12785   64277 |
  H-->  | 0-0 |      0   65536 |
        | 0-1 |  12785   64277 |
        | 0-2 |  25080   60547 |
        | 0-3 |  36410   54491 |
        +----------------------+
```

The current for the winding is computed by multiplying the drive current (which is the maximum current to use) by the value from the table (a signed value), and then dividing by 65536. The result is a waveform that varies sinusoidally as the motor steps through a full cycle, from 0 to the positive and negative values of the drive current.

If the full stepping mode is used, then only those entries marked with "F" are used. If half stepping mode is used, then those entries marked with "F" and "H" are used. If micro-stepping mode is used, then all the entries are used.

Looking at the entries marked with "F", you can see that both windings are on at each whole step point. This is normal whole stepping. However, whole stepping could also be performed by using just the entries marked with "H", in which case only one winding is on at each step point. This is called "wave" drive and is also an option for driving the stepper.

Micro-stepping is appropriate for low step rates and allows the motor to move more smoothly between full step points. Micro-stepping places a greater burden on the CPU because the current in the windings must be recomputed and changed more often. As the step rate increases the benefit of using micro-stepping is reduced and half or full stepping should be used instead.

The speed of the motor is determined by the time between steps (the "step time"). The step time is the time between full, half, or micro-steps, depending on which mode is used. The shorter the step time, the faster the motor runs. In the simplest case of constant speed, the step time is just the inverse of the motor speed expressed as a step rate (in steps/sec). For acceleration and deceleration, the instantaneous speed changes at every step time, so the step time for the next step must be computed at every step. The method for computing speed profiles is taken from the article, *"Generate stepper-motor speed profiles in real time"* by David Austin, Embedded Systems Design, Jan 2005. The article is available at `http://www.embedded.com/`.

Refer to the figure below for a typical speed profile. In this case the motor is stopped. The number of steps to accelerate to the running speed and decelerate back to 0 are computed (accel and decel can be different rates). Once the number of steps for accel and decel are known, then the absolute position of the profile transition points can be computed. The transition points are used at each step time to determine if the method for computing step time needs to change. For example, as the motor is stepping to each position toward the stop position, when it reaches the "Decel Pos", it will change the step time calculation from a constant to a gradually increasing step time at each step (thus causing the motor to slow).

```
                         Speed Profile
         Speed
           ^
           |
           |
Run Speed -|- - - - - - ***********************
           |          * |                     |*
           |        *   |                     | *
           |      *     |                     |  *
           |    *       |                     |   *
Stopped(0)-+----|-------|---------------------|---|-----> Position
                |       |                     |   |
                |     Run Pos                 |  Stop Pos
                |                             |
            Accel Pos                     Decel Pos
```

If the motor is already moving, then computing the speed profile becomes more complex. The algorithm must take into account whether the motor is in accel, run, or decel phases, whether the new speed is higher or lower than the current speed, whether the accel and decel rates have changed, and whether the new position can be reached given the current position, speed, and deceleration rate.

In the Step Sequencer module, the main work of computing the speed profile is done by the StepSeqMove() function. The output of this function is the calculation of the speed profile transition points. The function StepSeqHandler() performs the actual stepping of the motor. This function is an interrupt handler that is triggered when the hardware timer that is used as a step timer, times out. Upon entry, StepSeqHandler() advances by one full, half, or micro step through the step sequence table, in the appropriate direction, to determine the winding settings for the next step. It then calls into the Step Control module to cause the winding current to change. Then the new position is compared against the speed profile transition points to determine the phase of the speed profile (the "motor status", which is stop, accel, decel, or run). This is used to compute the step time of the next step, and the step timer is started up again. Thus, the step sequencing continues until the stop position is reached.

The code for implementing the Step Sequencer is contained in `stepseq.c`, with `stepseq.h` containing the definitions for the variables and functions exported to the application.

6.2 Definitions

Defines

- DRIVE_CURRENT
- GET_C0(w)
- GET_CMIN(s)
- GET_NUM_STEPS(s, w)
- HOLD_CURRENT

Functions

- static unsigned long LongDiv256 (unsigned long ulNum, unsigned long ulDenom)
- static unsigned long MulDiv (unsigned long ulValue, unsigned long ulNum, unsigned long ulDenom)
- void StepSeqControlMode (unsigned char ucMode)
- void StepSeqCurrent (unsigned short usDrive, unsigned short usHold, unsigned long ulMax)
- void StepSeqDecayMode (unsigned char ucMode)
- void StepSeqHandler (void)
- void StepSeqInit (void)
- void StepSeqMove (long lNewPosition, unsigned short usSpeed, unsigned short usAccel, unsigned short usDecel)
- void StepSeqShutdown (void)
- void StepSeqStepMode (unsigned char ucMode)
- void StepSeqStop (void)
- static void StepSeqUsingChop (unsigned long ulWinding, unsigned int uTableIdx)
- static void StepSeqUsingClosedPwm (unsigned long ulWinding, unsigned int uTableIdx)
- static void StepSeqUsingOpenPwm (unsigned long ulWinding, unsigned int uTableIdx)

Variables

- static unsigned char bDeferredMove
- static unsigned char bStopping
- volatile long g_lCurrentPos
- unsigned char g_ucMotorStatus
- unsigned long g_ulStepTime
- unsigned short g_usPwmFreq
- static long lChopSetting[2]
- static long lDeferredPosition
- static long lPosAccel
- static long lPosDecel
- static long lPosRun
- static long lPosStop
- static long lPwmSetting[2]
- static long lStepDelta
- int nMicroStepTable[ ][2]
- static int nPrevStepLevel[2]
- static unsigned char ucControlMode
- static unsigned char ucDecayMode
- static unsigned char ucStepMode
- static unsigned long ulAccelDenom
- static unsigned long ulDecelDenom
- static unsigned long ulDenom
- static unsigned short ulMaxI
- static unsigned long ulMinStepTime
- static unsigned long ulStep0Time
- static unsigned long ulStep1Time
- static unsigned short usDeferredAccel
- static unsigned short usDeferredDecel
- static unsigned short usDeferredSpeed
- static unsigned short usDriveI
- static unsigned int uSettingIdx
- static unsigned short usHoldI
- static unsigned short usLastDecel

6.2.1 Define Documentation

6.2.1.1 DRIVE_CURRENT

Definition:

```
#define DRIVE_CURRENT
```

Description:

An index which will select the drive current setting for applying current to the windings.

6.2.1.2 GET_C0

Computes the initial value of C which is the step time.

Definition:

```
#define GET_C0(w)
```

Parameters:

w is the rate of acceleration, specified in steps per second^2.

Description:

From David Austin's article, the starting inter-step delay is based on the acceleration.

Returns:

Returns the initial value of C in 24.8 fixed-point notation.

6.2.1.3 GET_CMIN

Computes the minimum inter-step delay.

Definition:

```
#define GET_CMIN(s)
```

Parameters:

s is the target speed specified in steps per second.

Description:

From David Austin's article, the minimum inter-step delay based on the maximum speed. This is the constant step time used when the motor is running at speed.

Returns:

Returns the minimum value of C in 24.8 fixed-point notation.

6.2.1.4 GET_NUM_STEPS

Computes the number of steps to accelerate to speed.

Definition:

```
#define GET_NUM_STEPS(s,
                      w)
```

Parameters:

s is the target speed specified in steps per second.

w is the rate of acceleration specified in steps per second^2.

Description:

From David Austin's article, the number of steps to accelerate at the given rate up to the given speed. This is used for computing the number of steps for both acceleration and deceleration.

Returns:

Returns the number of steps required to reach the given speed at the given constant rate of acceleration.

6.2.1.5 HOLD_CURRENT

Definition:

```
#define HOLD_CURRENT
```

Description:

An index which will select the holding current setting for applying current to the windings.

6.2.2 Function Documentation

6.2.2.1 LongDiv256 [static]

Divides two numbers, returning the result in 24.8 fixed-point notation.

Prototype:

```
static unsigned long
LongDiv256(unsigned long ulNum,
           unsigned long ulDenom)
```

Parameters:

ulNum is the numerator for the division.

ulDenom is the denominator for the division.

Description:

This function takes two integers and divides them, returning the results in 24.8 fixed-point notation (with full accuracy). Assuming infinite precision of the operands, this would be "(256 $*$ ulNum) / ulDenom", though with the fixed precision of the processor the multiply would overflow if performed as is.

Returns:

The result of the division.

6.2.2.2 MulDiv [static]

Multiplies a number by a fraction specified as a numerator and denominator.

Prototype:

```
static unsigned long
MulDiv(unsigned long ulValue,
       unsigned long ulNum,
       unsigned long ulDenom)
```

Parameters:

ulValue is the value to be multiplied.

ulNum is the numerator of the fraction.

ulDenom is the denominator of the fraction.

Description:

This function take an integer and multiplies it by a fraction specified by a numerator and denominator. Assuming infinite precision of the operands, this would be "(ulValue $*$ ulNum) /

ulDenom", though with the fixed precision of the processor the multiply would overflow if performed as is.

Returns:
The result of the multiplication.

6.2.2.3 StepSeqControlMode

Sets the current control method to chopper, open-loop PWM or closed-loop PWM.

Prototype:
```
void
StepSeqControlMode(unsigned char ucMode)
```

Parameters:
ucMode is the current control mode, CONTROL_MODE_OPENPWM, CONTROL_MODE_-CLOSEDPWM or CONTROL_MODE_CHOPPER.

Description:
This function sets the method used for current control. The motor must be stopped (MOTOR_-STATUS_STOP), or else this function does nothing. If the mode is PWM, then the PWM period is computed from g_ulPwmFreq, which should have been set to the desired PWM frequency prior to calling this function.

Also, StepSeqCurrent() should have been called prior to calling this function in order to set the drive and holding current.

Returns:
None.

6.2.2.4 StepSeqCurrent

Sets the drive and holding current to be used by the motor.

Prototype:
```
void
StepSeqCurrent(unsigned short usDrive,
               unsigned short usHold,
               unsigned long ulMax)
```

Parameters:
usDrive is the drive current in milliamps.
usHold is the holding current in milliamps.
ulMax is the maximum current that would flow in the winding given the winding resistance and bus voltage.

Description:
This function takes the specified drive and hold currents in milliamps and converts it and stores it in units that are used by the chopping and PWM control methods.

For the chopper and closed-loop PWM methods, the specified current is converted into the equivalent value in ADC counts, which is what is measured by ADC when making current measurements.

For open-loop PWM, the duty cycle is calculated for the drive and hold currents by using the ratio to the ulMax, maximum current specified.

Note that the ulMax current specified is not the maximum safe current that should flow in the winding. It is the maximum current that would flow if the full bus voltage were applied to the winding with no modulation. This is usually a value that is much larger than the maximum safe winding current.

This function takes effect immediately, which means the current values can be changed while the motor is running.

Returns:
None.

6.2.2.5 StepSeqDecayMode

Sets the current decay mode to fast or slow.

Prototype:
```
void
StepSeqDecayMode(unsigned char ucMode)
```

Parameters:
ucMode is the current decay mode, DECAY_MODE_SLOW or DECAY_MODE_FAST.

Description:
This function sets the decay mode. It takes effect immediately, which means the the value can be changed while the motor is running.

Returns:
None.

6.2.2.6 StepSeqHandler

Processes a single step.

Prototype:
```
void
StepSeqHandler(void)
```

Description:
This function is called when the step timer times out. It is called once for each step (or half/micro step). The motor position is advanced, along with an index that points into the step sequence table. Then functions are called to cause the actual step to take place according to the position in the step sequence table.

Once the step has taken place, then the current position is compared with the different points in the speed profile (accel, run, decel, stop), and actions taken to implement that part of the speed profile. These are summarized as follows:

- ACCEL - the step time for the first step is used. At each step, the new step time is calculated in order to decrease the time between steps and thus causes the motor to accelerate.
- RUN - at the run point, the step timer is set to use the duration that is equivalent to the running speed.
- DECEL - the step time for the first decelerating step is used. At each following step, the new step time is calculated in order to increase the time between steps, and thus causes the motor to decelerate.
- STOP - the motor is set to use the holding current, and the step timer is stopped so no more steps will occur.

Returns:
None.

6.2.2.7 StepSeqInit

Initializes the step sequencer module.

Prototype:
```
void
StepSeqInit(void)
```

Description:
This function will configure the peripherals needed to run the step sequencer.

Returns:
None.

6.2.2.8 StepSeqMove

Initiates a move of the motor by calculating a speed profile and starting up the step sequencing.

Prototype:
```
void
StepSeqMove(long lNewPosition,
            unsigned short usSpeed,
            unsigned short usAccel,
            unsigned short usDecel)
```

Parameters:
lNewPosition is the new target motor position in steps in fixed-point 24.8 format.
usSpeed is the target running speed in steps/sec.
usAccel is the acceleration in steps/sec^2.
usDecel is the deceleration in steps/sec^2.

Description:
This function calculates a speed profile for the specified motion. It looks at the current position and the number of steps needed to accelerate to, and decelerate from, in order to end up at the specified target position. The acceleration, running, deceleration, and stopping points are calculated, along with parameters needed for calculating the acceleration and deceleration step times, done by the step handler StepSeqHandler().

The current motion is also considered. If the motor is already moving, then the current speed and direction are taken into consideration, and a new speed profile is computed. If the new speed is different from the current speed, then this may involve accelerating or decelerating from the current speed to get to the new speed. It may also be the case that given the current speed and direction that it is not possible to reach the new target position without first stopping the motor and then moving it again. In this case, a deferred move is set up, and the motor is commanded to decelerate to a stop. Once it stops, then the deferred move is made to ensure the motor will end up at the correct position.

The parameter *lNewPosition* is a signed number representing the motor position in fixed-point 24.8 format. The upper 24 bits are the (signed) whole step position, while the lower 8 bits are the fractional step position. While this allows for a theoretical resolution of 1/256 step size, the motor does not actually support micro-steps that small. The value of the lower 8 bits (the fractional step) is the fractional value multiplied by 256. For example, the lower 8 bits of a half-step is 0x80 ($0.5 * 256$). Likewise, a quarter step is 0x40 ($0.25 * 256$). In order to use half-steps you must be using half- or micro-stepping mode. In order to use fractional steps smaller than half, you must be using micro-stepping mode.

Returns:
None.

6.2.2.9 StepSeqShutdown

Stops the motor immediately with no deceleration, and turns off all control signals.

Prototype:
```
void
StepSeqShutdown(void)
```

Description:
This function will disable all control methods, stop the step sequencing, and set all the control signals to a safe level. The H-bridges will be disabled. Position knowledge will be lost.

Returns:
None.

6.2.2.10 StepSeqStepMode

Sets the step size to whole-steps, half-steps, or micro-steps.

Prototype:
```
void
StepSeqStepMode(unsigned char ucMode)
```

Parameters:
ucMode is the step size, STEP_MODE_FULL, STEP_MODE_WAVE, STEP_MODE_HALF, or STEP_MODE_MICRO.

Description:
This function sets the step size. The size can be either full (normal or wave), half- or micro-steps.

The motor must be stopped (MOTOR_STATUS_STOP) or else this function does nothing.

Returns:
None.

6.2.2.11 StepSeqStop

Initiates a stop of the motor as quickly as possible, without loss of control.

Prototype:

```
void
StepSeqStop(void)
```

Description:
This function will immediately decelerate the motor to a stop using the last specified deceleration rate. This is a graceful way to stop the motor quickly, and does not cause loss of position information.

If the holding current has a non-zero value, then when the motor is stopped, the holding current will be applied.

Returns:
None.

6.2.2.12 StepSeqUsingChop [static]

Performs a single step of a sequence for a single winding using chopper mode.

Prototype:

```
static void
StepSeqUsingChop(unsigned long ulWinding,
                 unsigned int uTableIdx)
```

Parameters:
ulWinding is the winding ID, WINDING_ID_A or WINDING_ID_B.
uTableIdx is the index into the step sequence table.

Description:
This function determines the setting for a single winding based on the position in the step sequence table (uTableIdx), and calls the appropriate function to control that winding using the chopper method. It handles both slow and fast current decay modes.

This function should be called once for each of the two windings, at each step.

Returns:
None.

6.2.2.13 StepSeqUsingClosedPwm [static]

Performs a single step of a sequence for a single winding using closed-loop PWM.

Prototype:

```
static void
StepSeqUsingClosedPwm(unsigned long ulWinding,
                      unsigned int uTableIdx)
```

Parameters:

ulWinding is the winding ID, WINDING_ID_A or WINDING_ID_B.

uTableIdx is the index into the step sequence table.

Description:

This function determines the setting for a single winding based on the position in the step sequence table (uTableIdx), and calls the appropriate function to control that winding using PWM. It handles both slow and fast current decay modes.

This function should be called once for each of the two windings at each step.

Returns:

None.

6.2.2.14 StepSeqUsingOpenPwm [static]

Performs a single step of a sequence for a single winding using open-loop PWM.

Prototype:

```
static void
StepSeqUsingOpenPwm(unsigned long ulWinding,
                    unsigned int uTableIdx)
```

Parameters:

ulWinding is the winding ID, WINDING_ID_A or WINDING_ID_B.

uTableIdx is the index into the step sequence table.

Description:

This function determines the setting for a single winding based on the position in the step sequence table (uTableIdx), and calls the appropriate function to control that winding using PWM. It handles both slow and fast current decay modes.

This function should be called once for each of the two windings at each step.

Returns:

None.

6.2.3 Variable Documentation

6.2.3.1 bDeferredMove [static]

Definition:

```
static unsigned char bDeferredMove
```

Description:

A flag that indicates there is a deferred move pending. A deferred move is used when StepSeqMove() is called with a new target position that is not reachable. This is either because the position is "behind" the the current motion direction, or because it is too close ahead of the current position and there are not enough steps left to decelerate to that stopping position. In these cases, the move target is saved as a deferred move that will be made as soon as the motor stops.

6.2.3.2 bStopping [static]

Definition:

```
static unsigned char bStopping
```

Description:

A flag that indicates that the motor is in the process of stopping. This is used to avoid multiple attempts to stop the motor if it is already stopping.

6.2.3.3 g_lCurrentPos

Definition:

```
volatile long g_lCurrentPos
```

Description:

The current position of the motor. This value is updated every step time. The motor position is kept in fixed-point 24.8 format.

6.2.3.4 g_ucMotorStatus

Definition:

```
unsigned char g_ucMotorStatus
```

Description:

The motor status, one of: MOTOR_STATUS_STOP, MOTOR_STATUS_RUN, MOTOR_STATUS_ACCEL, MOTOR_STATUS_DECEL.

6.2.3.5 g_ulStepTime

Definition:

```
unsigned long g_ulStepTime
```

Description:

The time between each step in system clock ticks. This value is recomputed at each step during acceleration and deceleration, and is fixed when the motor is running at a continuous speed. To maintain precision, this step time is kept in fixed-point 24.8 format for all calculations, and converted to integer format only when it is written to the timer.

6.2.3.6 g_usPwmFreq

Definition:

```
unsigned short g_usPwmFreq
```

Description:

The PWM frequency that will be used for PWM mode. This is used to calculate the PWM period when the StepSeqControlMode() function is called. This value should be set to the desired PWM frequency prior to calling that function.

6.2.3.7 lChopSetting [static]

Definition:

```
static long lChopSetting[2]
```

Description:

This variable holds the value that is used for the chopper current comparison. It is in units of raw ADC counts. This is a value that represents the current that should be applied to the winding, and is computed and set by the StepSeqCurrent() function. The first entry is the value representing the drive current, and the second entry is the value representing the holding current.

6.2.3.8 lDeferredPosition [static]

Definition:

```
static long lDeferredPosition
```

Description:

The target position of the deferred move.

6.2.3.9 lPosAccel [static]

Definition:

```
static long lPosAccel
```

Description:

The step position at which an acceleration should begin. This will normally be the first step after the current position.

6.2.3.10 lPosDecel [static]

Definition:

```
static long lPosDecel
```

Description:

The step position at which a deceleration should begin. This is computed when the motion is requested, as some future position along the speed profile.

6.2.3.11 lPosRun [static]

Definition:

```
static long lPosRun
```

Description:

The step position at which the step time should transition to running speed, from a prior acceleration or deceleration. This is computed when the motion is requested by StepSeqMove(), as some future position along the speed profile.

6.2.3.12 lPosStop [static]

Definition:

```
static long lPosStop
```

Description:

The step position at which motion should stop. This is computed when the motion is requested, at some future position along the speed profile, and should always be the end of the deceleration curve.

6.2.3.13 lPwmSetting [static]

Definition:

```
static long lPwmSetting[2]
```

Description:

This variable holds the value that is used for the PWM on time, when PWM mode is used. This is a value that represents the current that should be applied to the winding, and is computed and set by the StepSeqCurrent() function. The first entry is the value representing the drive current, and the second entry is the value representing the holding current.

6.2.3.14 lStepDelta [static]

Definition:

```
static long lStepDelta
```

Description:

The size of each step. This is in fixed-point 24.8 format, which means that a value of 1 whole step is 0x100, and the value of a half-step is 0x080 (one-half is 128/256 or 0x80/0x100). This value is signed to account for direction. At each step time, this value is added to the current position to advance the current position.

6.2.3.15 nMicroStepTable

Definition:

```
int nMicroStepTable[][2]
```

Description:

This is the step sequencing table. It is used to determine, at each step time, the polarity and value that should be applied to the winding. This table has entries for 8 microsteps per step, giving 32 microsteps for one stepping cycle. It can be used for micro, half, or whole stepping. Each entry going down represents a micro step, and the columns are the windings, A and B. For whole stepping, only the entries at 0, 8, 16, and 24 are used. For half stepping 0, 4, 8, 12, 16, 20, 24, and 28 are used.

6.2.3.16 nPrevStepLevel [static]

Definition:

```
static int nPrevStepLevel[2]
```

Description:

This holds the value from the step sequence table from the prior step. It is used to see if the setting for a particular winding has changed since the previous step. This is used to avoid extra calls to the stepping functions if there has been no change to the winding.

6.2.3.17 ucControlMode [static]

Definition:

```
static unsigned char ucControlMode
```

Description:

The current control method, CONTROL_MODE_CHOP, CONTROL_MODE_OPENPWM, or CONTROL_MODE_CLOSEDPWM.

6.2.3.18 ucDecayMode [static]

Definition:

```
static unsigned char ucDecayMode
```

Description:

The current decay mode, DECAY_MODE_SLOW, or DECAY_MODE_FAST.

6.2.3.19 ucStepMode [static]

Definition:

```
static unsigned char ucStepMode
```

Description:

The step size mode, STEP_MODE_FULL, STEP_MODE_WAVE, STEP_MODE_HALF, or STEP_MODE_MICRO.

6.2.3.20 ulAccelDenom [static]

Definition:

```
static unsigned long ulAccelDenom
```

Description:

The denominator of the equation used for calculating step times during acceleration. This is the denominator value used for the first acceleration step.

6.2.3.21 ulDecelDenom [static]

Definition:

```
static unsigned long ulDecelDenom
```

Description:

The denominator of the equation used for calculating step times during deceleration. This is the denominator value used for the first deceleration step.

6.2.3.22 ulDenom [static]

Definition:

```
static unsigned long ulDenom
```

Description:

The denominator of the equation used for calculating step times during acceleration or deceleration. This value is adjusted at every step time.

6.2.3.23 ulMaxI [static]

Definition:

```
static unsigned short ulMaxI
```

Description:

The value of the maximum current in milliamps. This value represents the maximum current that would flow through the winding if it were left on, not the value that should be applied. This is the value that is passed in from the StepSeqCurrent() function. It is saved because it is used for calculating PWM duty cycle.

6.2.3.24 ulMinStepTime [static]

Definition:

```
static unsigned long ulMinStepTime
```

Description:

The minimum stepping time in system clock ticks. This is computed based on target running speed, and represents the step time once the motor is running at full speed. This value is in fixed-point 24.8 format.

6.2.3.25 ulStep0Time [static]

Definition:

```
static unsigned long ulStep0Time
```

Description:

The step time for the zeroth step. This is the time until the motor is first moved. To maintain precision, the step times are kept in fixed-point 24.8 format, until they are actually written to the timer.

6.2.3.26 ulStep1Time [static]

Definition:

```
static unsigned long ulStep1Time
```

Description:

The step time for the first step in the acceleration profile.. This is the time between the first and second movements of the motor. The first step time is a special case and is pre-computed. All following step times during acceleration are computed on the fly. To maintain precision, all step times are kept in fixed-point 24.8 format, until they are used.

6.2.3.27 usDeferredAccel [static]

Definition:

```
static unsigned short usDeferredAccel
```

Description:

The acceleration to be used for the deferred move.

6.2.3.28 usDeferredDecel [static]

Definition:

```
static unsigned short usDeferredDecel
```

Description:

The deceleration to be used for the deferred move.

6.2.3.29 usDeferredSpeed [static]

Definition:

```
static unsigned short usDeferredSpeed
```

Description:

The target speed of the deferred move.

6.2.3.30 usDriveI [static]

Definition:

```
static unsigned short usDriveI
```

Description:

The value of the drive current in milliamps. This is the value that is passed in from the StepSeqCurrent() function. It is saved because it is used for calculating PWM duty cycle.

6.2.3.31 uSettingIdx [static]

Definition:

```
static unsigned int uSettingIdx
```

Description:

An index that is used for looking up the current setting to be applied to the winding. It is either DRIVE_CURRENT or HOLD_CURRENT.

6.2.3.32 usHoldI [static]

Definition:

```
static unsigned short usHoldI
```

Description:

The value of the holding current in milliamps. This is the value that is passed in from the StepSeqCurrent() function. It is saved because it is used for calculating PWM duty cycle.

6.2.3.33 usLastDecel [static]

Definition:

```
static unsigned short usLastDecel
```

Description:

The most recent value used for deceleration, in steps/sec^2. It is saved because it is used by the StepSeqStop() function as the deceleration value to use for immediately decelerating the motor to a stop.

7 Step Control

7.1 Introduction

The Step Control module is used for controlling the drive signals to the stepper motor. The functions in this module are called by the Step Sequencer in order to set the control pins to the specific values needed to attain the correct position in the step sequence.

Normally there is no reason that an application needs to call any of these functions directly, nor make direct access to any of the module global variables. The following explains how this module is used by the Step Sequencer module.

First, the module is initialized by calling StepCtrlInit(). Then, open or closed-loop PWM mode, or Chopper mode is selected by calling StepCtrlOpenPWMMode(), StepCtrlClosedPWMMode() or StepCtrlChopMode(). These functions should be called whenever the control mode is changed, when the motor is stopped. The module is not designed to handle control mode changes when the motor is running.

Functions are provided to set a specific winding to be controlled in open-loop PWM, closed-loop PWM or chopper mode, fast or slow current decay, and using a specific control value. These functions are StepCtrlChopSlow(), StepCtrlChopFast(), StepCtrlOpenPwmSlow(), StepCtrlOpen-PwmFast(), StepCtrlClosedPwmSlow() and StepCtrlClosedPwmFast(). If open-loop PWM mode is used, then the control value is the amount of time the winding signal should be turned on (determining PWM duty cycle). If closed-loop PWM or chopper mode is used, then the control value is the current threshold that should be used for chopping.

Finally, two interrupt handlers are provided. There is a timer interrupt handler, StepCtrlTimer-Handler(). This is used for measuring either the fixed on time for open-loop PWM mode, or the blanking off time for chopper mode. There is also an interrupt handler for ADC conversion: Step-CtrlADCHandler(). This interrupt handler is invoked whenever an ADC conversion is completed. This handler is used in chopper mode to measure the winding current, and decide whether to turn the winding off (chopping). It is also used in closed-loop PWM mode to measure the winding current and adjust the PWM duty cycle to maintain the proper current in the winding.

Optimizations

An optimization that has been made is that direct register accesses are made to the peripherals in the interrupt handlers, instead of making calls to the DriverLib peripheral library. The HWREG macro is used to make the register accesses. This too provides somewhat more efficient code than making a function call, at the possible expense of a slight increase in code size. Wherever a register access is made like this, there is a comment in the code showing the equivalent DriverLib call.

It is likely that an actual stepper application will use either chopper or one of the PWM control modes, but not all three. In this case, if desired, the user could modify this module to remove the code that implements the unused control methods. To do this, the functions that implement the unused method could be deleted, and the places in the code where there is a run-time conditional based on the control method can be changed to eliminate the unused branch. These changes would make the resulting code somewhat smaller, and more efficient to execute.

Motor Control Circuit

The microcontroller can control the current in the motor windings through the use of three control signals for each winding. There is one signal for each side of the H-bridge, designated P and N, and there is an enable signal. (Note that on the board schematic, the two sides of the H-bridge are designated 1 and 2).

The diagram below shows a simplified representation of the H-bridge circuit. Each side has two switches to control whether the voltage applied to that side of the winding is the bus voltage (+V) or ground (GND). The polarity of the motor winding is designated so that the P side of the H-bridge is considered positive. Therefore, when the P high-side switch (Ph) is closed, and the N low-side switch is closed (Nl), the bus voltage will be connected on the positive side of the winding, and ground connected on the negative side of the winding. Thus, positive bus voltage is applied to the winding, and the current will flow in the positive direction. Likewise, if the P low-side switch (Pl), and N high-side (Nh) are closed, then negative voltage is applied to the winding and current will flow in the negative direction.

H-Bridge Switching Table

EN	P	N	Ph	Pl	Nh	Nl	V
0	0	0					0
1	0	0		X		X	0
1	0	1		X	X		-
1	1	0	X			X	+
1	1	1	---N/A---				

```
                         ^ +V
        H-Bridge         |
    Circuit Diagram      |
              +-------+-------+
              |               |
    Ph----\                    /----Nh
              \               /
              |    WINDING    |
              +----/\/\/\/----+
              |    +     -    |
    Pl----\                    /----Nl
              \               /
              |               |
              +-------+-------+
                      |
                      |
                     --- GND
                      -
```

The control circuit is designed so that both the high- and low-side switches cannot be closed at the same time. Otherwise, it would be possible to short out the power supply.

When the enable signal is off, all the switches are open. When the enable signal is on, either the high- or low-side switch will be on, depending on the state of the control signal. The Switching Table above shows the three control signals available for each winding, their possible states, and the resulting connections of the H-bridge switches. An "X" in the table means that switch is closed. The V column shows the voltage applied to the winding: none (0), positive (+), or negative (-).

The following table shows how the pins of the microcontroller are assigned to the motor control signals. Each of the control signals can be controlled by a PWM generator. There are two signals per PWM generator, with each PWM generator having an "A" and a "B" half. The "A" and "B" halves of the PWM generator should not be confused with the fact that the two windings are called A and B. For each winding, the two switching signals are controlled from a single PWM generator. The "A" half of the generator controls the P side of the winding and the "B" half of the generator controls the N side. For the enable signals, there is one PWM generator that controls the two signals, one for each winding. In this case, the A half of the PWM generator controls the enable signal for the A winding, and the B half of the PWM generator controls the B winding. In the table below, the control signals are shown for each winding, associated with the PWM outputs. The values shown in parentheses indicate which PWM generator controls that PWM output.

H-Bridge Control Pin Assignments

	Winding	
Ctrl	A	B
P-side	D0/PWM0(0A)	B0/PWM2(1A)
N-side	D1/PWM1(0B)	B1/PWM3(1B)
Enable	E0/PWM4(2A)	E1/PWM5(2B)

Current Decay Modes

In order to stop the current in a winding, the voltage must be removed. There are two ways to do this. First, if both sides of the H-bridge are set the same way, so that both low-side switches are closed, then both sides of the winding are connected to ground. Since the motor winding is inductive, if there was already current flowing in the winding, it will continue to circulate in the winding and decay slowly. This is called slow decay mode.

On the other hand, if all of the switches are opened (by turning the enable signal off), then there is also no voltage applied to the winding, except that now neither side of the winding is connected to anything. In this case, the current cannot continue to circulate and decays rapidly. This is called fast decay mode.

Using PWM Outputs as GPIOs

The motor control circuits are driven by the microcontroller's PWM outputs. The PWM outputs are used essentially as if they were GPIO outputs when a control signal needs to be set to the on or off value. This is done by setting a very short PWM period, and then programming the PWM generator to drive the output low or high for all events. The reason this is done is because the PWM generator outputs can be programmed to be automatically placed in a safe state if the hardware fault pin is asserted. So, if the hardware fault pin is used, then the control pins to the motor drive circuitry will automatically go into the safe state, which is to disable the gate drivers and open all the H-bridge switches.

The second reason that the PWM generators are used this way is because there are some places in the code where a certain control signal needs to be changed to be high or low. To accomplish this, the interrupt handler code just writes a pre-computed value to the PWM generator control register. It doesn't need to know if the output is being used for actual PWM, or just as a GPIO in chopper mode. Otherwise, more run-time conditionals would be needed to change the pin configurations between PWM and GPIO depending on whether PWM was needed for a pin.

Chopper Operation

The chopper works by using the ADC to measure the winding current when the winding has voltage applied, and turning off the voltage to the winding when the current goes above the target current threshold. The control signal is left off for the off blanking time before being turned on again.

Chopping control of a winding is started when one of the functions StepCtrlChopSlow() or StepCtrlChopFast() is called. These functions will set the gate driver control signals so that the voltage is applied to the winding in the correct direction and start an A/D conversion. This will cause current to begin to flow in the winding, and when the A/D conversion is complete it will cause an interrupt.

The interrupt handler for the ADC looks at the measured current. If the current is below the threshold, it leaves the winding on and starts another conversion. If the current is above the threshold it turns the control signal to the winding off, and starts up a timer with a timeout of the off blanking time.

When the timer expires, a timer interrupt is generated. The interrupt handler for the timer turns the winding back on and starts up an A/D conversion, and the whole chopping cycle repeats. This continues until the next step requires a change of current in the winding.

Open-loop PWM Operation

The current in the winding can also be controlled using PWM instead of by measuring the current with the chopper. The PWM method works by turning the winding on for a fixed amount of time, called the "fixed rise time". This time allows the current to rise rapidly in the winding before PWM is started.

PWM control of a winding is started when one of the StepCtrlOpenPwmSlow() or StepCtrlOpen-PwmFast() functions is called. These functions will set the gate driver control signals so that the voltage is applied to the winding in the correct direction. Then the timer is started with a timeout of the fixed rise time.

When the timer expires, a timer interrupt is generated. The interrupt handler for the timer programs the PWM generator control register to change the control pin from being on, to being controlled by the PWM duty cycle. In this way, the pin starts switching on and off as determined by the PWM frequency and duty cycle. This continues until the next step requires a change of current in the winding.

Closed-loop PWM Operation

There is another PWM mode available called closed-loop PWM. The "closed" part refers to the fact that measured current feedback is used to determine the PWM duty cycle.

This method works by synchronizing the ADC acquisition with the PWM pulse so that the ADC acquisition is started when the pulse is on. When the acquisition is complete the ADC interrupt is triggered. In the ADC handler, the current measurement is taken from the ADC and the PWM duty cycle is recalculated based on the measured current. If the current is too low, then the duty cycle is increased. If the current approaches the target level, then the PWM duty cycle is decreased. If the measured current is too high, then the PWM duty cycle is set to the minimum pulse width. The minimum pulse width is needed in order to make a current measurement.

This method has the advantage of using feedback to regulate the current in the winding (as in chopper mode), yet does not require the intensive processing of chopper mode to take data and switch the output signals. This method relies on the ability of the hardware to synchronize the PWM and ADC modules. This has the effect of isolating the action of taking ADC data and switching the outputs from the processing to determine the pulse width setting, making it much less sensitive to interrupt latency.

The code for implementing the Step Control module is contained in `stepctrl.c`, with `stepctrl.h` containing the definitions for the variables and functions exported to the application.

7.2 Definitions

Data Structures

- tWinding

Defines

- ACQ_DELAY_COUNTS

- ADC_DELAY
- ADC_STATE_CHOP
- ADC_STATE_CLOSEDPWM
- ADC_STATE_IDLE
- CTRL_PIN_OFF_VAL
- CTRL_PIN_ON_VAL
- CTRL_PIN_PWMA_VAL
- CTRL_PIN_PWMB_VAL
- MAX_CURRENT_DELTA
- MIN_PWM_COUNTS
- TIMER_STATE_ADC_DELAY
- TIMER_STATE_BLANK_OFF
- TIMER_STATE_FIXED_ON
- TIMER_STATE_IDLE
- WINDING_A_PWM_GEN_OFFSET
- WINDING_B_PWM_GEN_OFFSET
- WINDING_EN_GEN_BASE
- WINDING_EN_PWM_GEN_OFFSET

Functions

- void StepCtrlADCAIntHandler (void)
- void StepCtrlADCBIntHandler (void)
- static void StepCtrlADCHandler (unsigned long ulWinding)
- void StepCtrlChopFast (unsigned long ulWinding, long lSetting)
- void StepCtrlChopMode (void)
- void StepCtrlChopSlow (unsigned long ulWinding, long lSetting)
- void StepCtrlClosedPwmFast (unsigned long ulWinding, long lSetting)
- void StepCtrlClosedPWMMode (unsigned long ulPeriod)
- void StepCtrlClosedPwmSlow (unsigned long ulWinding, long lSetting)
- void StepCtrlInit (void)
- void StepCtrlOpenPwmFast (unsigned long ulWinding, long lSetting)
- void StepCtrlOpenPWMMode (unsigned long ulPeriod)
- void StepCtrlOpenPwmSlow (unsigned long ulWinding, long lSetting)
- void StepCtrlTimerAIntHandler (void)
- void StepCtrlTimerBIntHandler (void)
- static void StepCtrlTimerHandler (unsigned long ulWinding)

Variables

- unsigned long g_ulCurrentRaw[2][8]
- unsigned long g_ulPeakCurrentRaw[2]
- unsigned long g_ulPwmPeriod
- unsigned short g_usBlankOffTime
- unsigned short g_usFixedOnTime
- tWinding sWinding[ ]

7.2.1 Data Structure Documentation

7.2.1.1 tWinding

Definition:

```
typedef struct
{
    unsigned long ulPwmGenBase;
    unsigned long ulPwmGenBit;
    unsigned long ulTmrLoadAddr;
    unsigned long ulTmrEnaVal;
    unsigned long ulPwmAB;
    unsigned long ulPwmGenCtlReg;
    unsigned long ulPwmGenCtlVal;
    unsigned long ulChopperCurrent;
    unsigned char ucADCSeq;
    unsigned char ucTimerState;
    unsigned char ucADCState;
}
tWinding
```

Members:

ulPwmGenBase The register base address of the PWM generator used to control the H-bridge for the winding.

ulPwmGenBit The bit identifier of the PWM generator used to control the winding.

ulTmrLoadAddr The load register address for the timer used for fixed timing methods.

ulTmrEnaVal The value used to enable the timer used for the winding.

ulPwmAB A value which is used to indicate winding A or B. It is used as an offset to the correct PWM control register for the H-bridge enable signals. The value is 0(A) or 4(B).

ulPwmGenCtlReg The PWM generator control register that is used by the timer and ADC ISRs to start or stop PWM control on the winding.

ulPwmGenCtlVal The value to be applied to the PWM generator control register, by the timer or ADC ISRs, to cause PWM to start or stop running.

ulChopperCurrent The current threshold to use for chopper mode in raw ADC counts.

ucADCSeq The sequencer used for chopper ADC samples for this winding.

ucTimerState The state of the fixed timer handler.

ucADCState The state of the ADC handler.

Description:

A structure that holds register addresses and control values that are used by the winding control code. The values are pre-computed as much as possible to reduce the amount of run-time calculations needed, especially for interrupt service routines.

7.2.2 Define Documentation

7.2.2.1 ACQ_DELAY_COUNTS

Definition:

```
#define ACQ_DELAY_COUNTS
```

Description:
For closed-loop PWM mode, defines the amount of time after the center of a PWM pulse before an ADC acquisition is triggered. This is used to adjust the setup and hold time of the current signal for an ADC measurement.

7.2.2.2 ADC_DELAY

Definition:
```
#define ADC_DELAY
```

Description:
Defines the amount of delay after the winding is turned on, before the ADC acquisition starts, when using chopper mode. The delay is to allow for rise time in the current sense circuit before an ADC acquisition is started. The value is in units of microseconds.

7.2.2.3 ADC_STATE_CHOP

Definition:
```
#define ADC_STATE_CHOP
```

Description:
Defines the CHOP state for the ADC handler.

7.2.2.4 ADC_STATE_CLOSEDPWM

Definition:
```
#define ADC_STATE_CLOSEDPWM
```

Description:
Defines the Closed-loop PWM state for the ADC handler.

7.2.2.5 ADC_STATE_IDLE

Definition:
```
#define ADC_STATE_IDLE
```

Description:
Defines the IDLE state for the ADC handler.

7.2.2.6 CTRL_PIN_OFF_VAL

Definition:
```
#define CTRL_PIN_OFF_VAL
```

Description:
Defines the value that is written to the PWM control register that will cause the output to be turned off.

7.2.2.7 CTRL_PIN_ON_VAL

Definition:

```
#define CTRL_PIN_ON_VAL
```

Description:

Defines the value that is written to the PWM control register that will cause the output to be turned on.

7.2.2.8 CTRL_PIN_PWMA_VAL

Definition:

```
#define CTRL_PIN_PWMA_VAL
```

Description:

Defines the value that is written to the PWM control register that will cause the PWM generator output to start generating PWM based on comparator A. Note that this "A" refers to part of the PWM generator (A and B comparators) and is not necessarily the same thing as the "A" winding.

7.2.2.9 CTRL_PIN_PWMB_VAL

Definition:

```
#define CTRL_PIN_PWMB_VAL
```

Description:

Defines the value that is written to the PWM control register that will cause the PWM generator output to start generating PWM based on comparator B. Note that this "B" refers to part of the PWM generator (A and B comparators) and is not necessarily the same thing as the "B" winding.

7.2.2.10 MAX_CURRENT_DELTA

Definition:

```
#define MAX_CURRENT_DELTA
```

Description:

For closed-loop PWM mode, defines the maximum difference between the measured and desired current before the PWM output will be turned on 100%. For differences less than this amount, the PWM output is adjusted to some value less than 100%.

7.2.2.11 MIN_PWM_COUNTS

Definition:

```
#define MIN_PWM_COUNTS
```

Description:

Defines the minimum PWM pulse width for closed-loop PWM mode. The PWM must have a minimum pulse width because current measurement can only be taken when the output is on.

7.2.2.12 TIMER_STATE_ADC_DELAY

Definition:

```
#define TIMER_STATE_ADC_DELAY
```

Description:
Defines the ADC_DELAY state for the fixed timer.

7.2.2.13 TIMER_STATE_BLANK_OFF

Definition:

```
#define TIMER_STATE_BLANK_OFF
```

Description:
Defines the BLANK_OFF state for the fixed timer.

7.2.2.14 TIMER_STATE_FIXED_ON

Definition:

```
#define TIMER_STATE_FIXED_ON
```

Description:
Defines the FIXED_ON state for the fixed timer.

7.2.2.15 TIMER_STATE_IDLE

Definition:

```
#define TIMER_STATE_IDLE
```

Description:
Defines the IDLE state for the fixed timer.

7.2.2.16 WINDING_A_PWM_GEN_OFFSET

Definition:

```
#define WINDING_A_PWM_GEN_OFFSET
```

Description:
Defines the register offset of the PWM generators used for winding A. This would be changed if the PWM outputs were wired differently.

7.2.2.17 WINDING_B_PWM_GEN_OFFSET

Definition:

```
#define WINDING_B_PWM_GEN_OFFSET
```

Description:
Defines the register offset of the PWM generators used for winding B. This would be changed if the PWM outputs were wired differently.

7.2.2.18 WINDING_EN_GEN_BASE

Definition:
```
#define WINDING_EN_GEN_BASE
```

Description:
Defines the register address of the base of the PWM generator that is used to control the H-bridge enable signals for windings A and B.

7.2.2.19 WINDING_EN_PWM_GEN_OFFSET

Definition:
```
#define WINDING_EN_PWM_GEN_OFFSET
```

Description:
Defines the register offset of the PWM generators used for the H-bridge enable signals (for both windings A and B).

7.2.3 Function Documentation

7.2.3.1 StepCtrlADCAIntHandler

The interrupt handler for the ADC sequencer used for winding A.

Prototype:
```
void
StepCtrlADCAIntHandler(void)
```

Description:
This handler is called when the ADC sequencer used for winding A finishes taking samples.

This interrupt handler calls a common handler for the ADC interrupts for winding A and B.

Returns:
None.

7.2.3.2 StepCtrlADCBIntHandler

The interrupt handler for the ADC sequencer used for winding B.

Prototype:
```
void
StepCtrlADCBIntHandler(void)
```

Description:

This handler is called when the ADC sequencer used for winding B completes taking samples.

This interrupt handler calls a common handler for the ADC interrupts for winding A and B.

Returns:

None.

7.2.3.3 StepCtrlADCHandler [static]

The interrupt handler for the ADC winding current sample.

Prototype:

```
static void
StepCtrlADCHandler(unsigned long ulWinding)
```

Parameters:

ulWinding specifies which winding is being processed. It can be one of WINDING_ID_A or WINDING_ID_B.

Description:

This handler is called from the interrupt handler for the ADC sequencer for winding A or B. The ADC is used for closed-loop PWM and Chopper modes. If chopper mode is used then the sample acquisition was started either from this function, or from the winding fixed timer interrupt handler. If closed-loop PWM mode is used, then the ADC sample acquisition is triggered from the PWM generator at a certain point in the PWM cycle (when the output is turned on).

Chopper operation:

This interrupt handler compares the current sampled from the ADC with the chopping threshold. If the measured current is below the threshold, then a new acquisition is started and the control pin is left in the on state. If the measured current is above the threshold, then the control pin is turned off, and the winding timer is used to start the off blanking time interval.

Closed-loop PWM operation:

The interrupt handler compares the current samples from the ADC with the chopping threshold. If the measured current is above the threshold, then the PWM output is set to the minimum width. If the measured current is below the threshold, then the PWM duty cycle is adjusted so that the duty cycle is related to the difference between the actual current and the desired current. The larger the difference, the higher the duty cycle.

On entry here, the interrupt has already been acknowledged by the specific ADC interrupt handler that called here.

Returns:

None.

7.2.3.4 StepCtrlChopFast

Sets up a step using chopper mode and fast decay.

Prototype:

```
void
StepCtrlChopFast(unsigned long ulWinding,
                 long lSetting)
```

Parameters:

ulWinding is the winding ID (A or B)

lSetting is the chopping current (signed), in raw ADC counts

Description:

This function will configure the chopper to control the pins needed for fast current decay. It drives the winding positive or negative (or off), according to the value and sign of the lSetting parameter. Once the control signals are set to apply voltage to the winding, an ADC acquisition is started. This will start the chopper running for this winding.

For fast current decay, one side of the H-bridge is set high, and the other side is set low, to cause current to flow in the positive or negative direction. The gate driver enable signal is then turned on or off to control the current. When the enable signal is on, bus voltage is applied to the winding and current flows. When the enable signal is off, all the H-bridge switches are open and the current in the winding decays rapidly.

Returns:

None.

7.2.3.5 StepCtrlChopMode

Configures the winding control signals for chopper mode.

Prototype:

```
void
StepCtrlChopMode(void)
```

Description:

This function should be called prior to using chopper mode as the control method. It configures the control signals and the PWM generators to be used in chopper mode.

Returns:

None.

7.2.3.6 StepCtrlChopSlow

Sets up a step using chopper mode and slow decay.

Prototype:

```
void
StepCtrlChopSlow(unsigned long ulWinding,
                 long lSetting)
```

Parameters:

ulWinding is the winding ID (A or B)

lSetting is the chopping current (signed), in raw ADC counts

Description:

This function will configure the chopper to control the pins needed for slow current decay. It drives the winding positive or negative (or off), according to the value and sign of the lSetting parameter. Once the control signals are set to apply voltage to the winding, an ADC acquisition is started. This will start the chopper running for this winding.

For slow current decay, one side of the H-bridge is set high, and the other side is set low, to cause current to flow in the positive or negative direction. The gate drivers are always enabled. To control current, the "high" side of the H-bridge is switched between high and low. When it is high, the bus voltage is applied to the winding and current flows. When it is low, then both sides of the H-bridge will be low, and bus voltage is removed, but current continues to circulate in the winding as it decays slowly.

Returns:

None.

7.2.3.7 StepCtrlClosedPwmFast

Sets up a step using Closed-loop PWM mode and fast decay.

Prototype:

```
void
StepCtrlClosedPwmFast(unsigned long ulWinding,
                      long lSetting)
```

Parameters:

ulWinding is the winding ID (A or B)

lSetting is the target winding current (signed), in raw ADC counts

Description:

This function will configure the PWM to control the pins needed for fast current decay. Its sets the winding gate drivers for positive or negative current depending on the lSetting parameter. Then the enable signal for the winding is set to be switched on and off by a PWM generator. The pulse width will be controlled by the ADC handler which will measure the actual current and adjust the PWM pulse width accordingly.

For fast current decay, one side of the H-bridge is set high, and the other side is set low, to cause current to flow in the positive or negative direction. The gate driver enable signal is then turned on or off to control the current. When the enable signal is on, bus voltage is applied to the winding and current flows. When the enable signal is off, all the H-bridge switches are open and the current in the winding decays rapidly.

Returns:

None.

7.2.3.8 StepCtrlClosedPWMMode

Configures the winding control signals for Closed-Loop PWM mode.

Prototype:

```
void
StepCtrlClosedPWMMode(unsigned long ulPeriod)
```

Parameters:

ulPeriod is the PWM period in system clock ticks

Description:

This function should be called prior to using Closed-loop PWM mode as the control method. It configures the control signals and the PWM generators to be used in PWM mode, and sets up the trigger points for ADC acqusition for current measurement.

Returns:

None.

7.2.3.9 StepCtrlClosedPwmSlow

Sets up a step using Closed-loop PWM mode and slow decay.

Prototype:

```
void
StepCtrlClosedPwmSlow(unsigned long ulWinding,
                      long lSetting)
```

Parameters:

ulWinding is the winding ID (A or B)

lSetting is the target winding current (signed), in raw ADC counts

Description:

This function will configure the PWM to control the pins needed for slow current decay. It sets the winding for positive or negative current, depending on the value of the lSetting parameter. The high side will be set to switch on and off according to a PWM generator. The pulse width will be controlled by the ADC handler which will measure the actual current and adjust the PWM pulse width accordingly.

For slow current decay, one side of the H-bridge is set high, and the other side is set low, to cause current to flow in the positive or negative direction. The gate drivers are always enabled. To control current, the "high" side of the H-bridge is switched between high and low. When it is high, the bus voltage is applied to the winding and current flows. When it is low, then both sides of the H-bridge will be low, and bus voltage is removed, but current continues to circulate in the winding as it decays slowly.

Returns:

None.

7.2.3.10 StepCtrlInit

Initializes the step control module.

Prototype:

```
void
StepCtrlInit(void)
```

Description:

This function initializes all the peripherals used by this module for controlling the stepper motor.

Returns:
None.

7.2.3.11 StepCtrlOpenPwmFast

Sets up a step using Open-loop PWM mode and fast decay.

Prototype:
```
void
StepCtrlOpenPwmFast(unsigned long ulWinding,
                    long lSetting)
```

Parameters:
ulWinding is the winding ID (A or B)
lSetting is the duration that the signal is on, in system clock ticks (signed to indicate polarity)

Description:
This function will configure the PWM to control the pins needed for fast current decay. It drives the winding positive or negative (or off), according to the value and sign of the lSetting parameter. Once the control signals are set to apply voltage to the winding, the fixed timer is started with a timeout value for the fixed rise time. When the fixed timer times out, it will set the PWM generator to start using PWM for the control signal. There is no feedback from the measured current in the winding, which is why this is called open-loop PWM.

The lSetting parameter is the duration in system clock ticks, that the PWM output will be on. This will be a fraction of the PWM period, resulting in the PWM duty cycle. It is a signed value, to indicate the direction the current should flow.

For fast current decay, one side of the H-bridge is set high, and the other side is set low, to cause current to flow in the positive or negative direction. The gate driver enable signal is then turned on or off to control the current. When the enable signal is on, bus voltage is applied to the winding and current flows. When the enable signal is off, all the H-bridge switches are open and the current in the winding decays rapidly.

Returns:
None.

7.2.3.12 StepCtrlOpenPWMMode

Configures the winding control signals for Open-loop PWM mode.

Prototype:
```
void
StepCtrlOpenPWMMode(unsigned long ulPeriod)
```

Parameters:
ulPeriod is the PWM period in system clock ticks

Description:
This function should be called prior to using open-loop PWM mode as the control method. It configures the control signals and the PWM generators to be used in open-loop PWM mode.

Returns:
None.

7.2.3.13 StepCtrlOpenPwmSlow

Sets up a step using Open-loop PWM mode and slow decay.

Prototype:
```
void
StepCtrlOpenPwmSlow(unsigned long ulWinding,
                    long lSetting)
```

Parameters:
ulWinding is the winding ID (A or B)
lSetting is the duration that the signal is on, in system clock ticks (signed to indicate polarity)

Description:
This function will configure the PWM to control the pins needed for slow current decay. It drives the winding positive or negative (or off), according to the value and sign of the lSetting parameter. Once the control signals are set to apply voltage to the winding, the fixed timer is started with a timeout value for the fixed rise time. When the fixed timer times out, it will set the PWM generator to start using PWM for the control signal. There is no feedback from the measured current, which is why this is called open-loop PWM.

The lSetting parameter is the duration in system clock ticks, that the PWM output will be on. This will be a fraction of the PWM period, resulting in the PWM duty cycle. It is a signed value, to indicate the direction the current should flow.

For slow current decay, one side of the H-bridge is set high, and the other side is set low, to cause current to flow in the positive or negative direction. The gate drivers are always enabled. To control current, the "high" side of the H-bridge is switched between high and low. When it is high, the bus voltage is applied to the winding and current flows. When it is low, then both sides of the H-bridge will be low, and bus voltage is removed, but current continues to circulate in the winding as it decays slowly.

Returns:
None.

7.2.3.14 StepCtrlTimerAIntHandler

The interrupt handler for the timer used for winding A fixed timing.

Prototype:
```
void
StepCtrlTimerAIntHandler(void)
```

Description:
This handler is called when the timer for winding A times out. This timer is used to generate a timeout either for the fixed on time if PWM mode is used, or the off blanking time if chopper mode is used.

This interrupt handler calls a common timer handler for both A and B windings.

Returns:
None.

7.2.3.15 StepCtrlTimerBIntHandler

The interrupt handler for the timer used for winding B fixed timing.

Prototype:
```
void
StepCtrlTimerBIntHandler(void)
```

Description:
This handler is called when the timer for winding B times out. This timer is used to generate a timeout either for the fixed on time if PWM mode is used, or the off blanking time if chopper mode is used.

This interrupt handler calls a common timer handler for both A and B windings.

Returns:
None.

7.2.3.16 StepCtrlTimerHandler [static]

The interrupt handler for fixed timing.

Prototype:
```
static void
StepCtrlTimerHandler(unsigned long ulWinding)
```

Parameters:
ulWinding specifies which winding is being processed. It can be one of WINDING_ID_A or WINDING_ID_B.

Description:
This handler is called from the specific interrupt handler for the timer used for winding A or winding B. There are two timers, one for each winding, each with an interrupt handler. This handler is used for the common processing for both. The timers are used to generate a timeout either for the fixed on time if open-loop PWM mode is used, or the off blanking time if chopper mode is used.

If open-loop PWM mode is used, then when this timer times out, it switches the control output to PWM (which was previously on). If chopper mode is used, then when this timer times out, it turns the control signal back on (which was previously off), and starts an ADC conversion.

On entry here, the interrupt has already been acknowledged by the specific timer interrupt handler that called here.

Returns:
None.

7.2.4 Variable Documentation

7.2.4.1 g_ulCurrentRaw

Definition:

```
unsigned long g_ulCurrentRaw[2][8]
```

Description:

This array is used to store the ADC data for each of the A and B windings. Storage is allocated for up to 8 samples on each winding, even though the sequencers are programmed for fewer samples than 8. This amount of space is allowed because it is the theoretical maximum number of samples that could be returned when reading all the data from an ADC sequencer, and ensures that an adjacent variable in memory will never be overwritten.

The data stored in this array is in units of raw ADC counts, and must be converted in order to find the current in engineering units.

7.2.4.2 g_ulPeakCurrentRaw

Definition:

```
unsigned long g_ulPeakCurrentRaw[2]
```

Description:

This is used to store the peak current measured in chopper mode, for each winding. The value stored is the last current reading taken when the chopper detects that the current has gone past the threshold. This should represent the current level when the winding is on. The value is reset to 0 whenever the winding is turned off, because when the winding is off, no current can be measured.

7.2.4.3 g_ulPwmPeriod

Definition:

```
unsigned long g_ulPwmPeriod
```

Description:

The PWM period in units of system clock ticks. This value is set when StepCtrlOpenPWMMode() is called to set up a PWM control mode. The value is used for PWM control methods in order to be able to compute on and off times in the PWM cycle.

7.2.4.4 g_usBlankOffTime

Definition:

```
unsigned short g_usBlankOffTime
```

Description:

When using the chopper control method, this determines the amount of time that the control signal is left turned off, before being turned on again. It is measured in units of microseconds, with a range of 10-65535. Great care should be taken when changing this value because an incorrect value could cause too much current to flow in the motor winding.

7.2.4.5 g_usFixedOnTime

Definition:

```
unsigned short g_usFixedOnTime
```

Description:

When using the open-loop PWM control method, this determines the amount of time that the signal is left turned on, before PWM is applied. It is measured in units of microseconds, with a range of 1-65535. This value should be set to an appropriate value prior to starting the PWM control mode.

7.2.4.6 sWinding

Definition:

```
tWinding sWinding[ ]
```

Description:

The table that holds the register addresses and control values for each winding.

8 User Interface

8.1 Introduction

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

The on-board user interface consists of a potentiometer, push button, and two LEDs. The on-board interface operates in two modes: speed mode and position mode. In speed mode the potentiometer controls the speed at which the motor runs. The button can be used to start and stop the motor, reversing direction each time. In position mode, the potentiometer controls the position of the motor. Motion commands are sent to the stepper every time the potentiometer is moved, so that the motor tracks the potentiometer position.

The initial mode is speed mode. The mode is indicated by blinking on the Mode LED. It blinks one time for speed mode, and twice for position mode. When the RDK board is reset, the Mode LED will blink one time, indicating speed mode. The mode can be changed by holding down the user button for 5 seconds.

In speed mode, the motor is started running by a single press and release of the user button. The speed can be changed while the motor is running by changing the position of the potentiometer knob. If the button is pressed again, the motor will stop. Each time it is pressed it will start or stop, reversing direction each time it starts. The motor speed ranges from 10 steps/second up to about 1000 steps/second at the extremes of the potentiometer range.

In position mode, the motor is enabled for motion. When the potentiometer knob is moved, the motor will move to track the knob position. If the button is pressed, the motor will be disabled and will not move with the knob. If it is pressed again, then the motor will be re-enabled. The potentiometer reading is scaled so that one turn of the potentiometer is about the same as one revolution of the motor. The result is a one-to-one response between the position of the knob and the motor.

As the motor turns, in either mode, the Status LED blinks at a rate corresponding to the motor speed. If a fault occurs, due to a current limit, then the status LED will blink rapidly. If this happens, the fault can be cleared by holding down the user button for 5 seconds, and the Status LED will stop blinking.

A periodic interrupt, the SysTick timer, is used to poll the state of the push button and perform debouncing. It is also used to read the ADC values for the potentiometer position as well as the bus voltage and processor temperature. The SysTick interrupt initiates motion commands when the position of the potentiometer changes.

When the serial interface is used, the on-board interface is (typically) disabled, and the button and potentiometer have no effect. However, the Status LED is still used as a motor speed indicator.

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

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

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

8.2 Definitions

Defines

- ABS(n)
- NUM_SWITCHES
- UI_INT_RATE
- UI_POT_MAX
- UI_POT_MIN

Enumerations

- tUIMode

Functions

- void SysTickIntHandler (void)
- static void UIButtonHold (void)
- static void UIButtonPress (void)
- void UIClearFaults (void)
- void UIEmergencyStop (void)
- void UIInit (void)
- void UIOnBoard (void)
- void UIParamLoad (void)
- void UIParamSave (void)
- void UIRun (void)
- void UISetChopperBlanking (void)
- void UISetControlMode (void)
- void UISetDecayMode (void)
- void UISetFaultParms (void)
- void UISetFixedOnTime (void)
- void UISetMotion (void)
- void UISetMotorParms (void)
- void UISetPWMFreq (void)
- void UISetStepMode (void)
- void UIStop (void)
- void UIUpgrade (void)

Variables

- unsigned char bReverse
- static tUIMode eUIMode
- unsigned long g_pulUIHoldCount[NUM_SWITCHES]
- const tUIOnboardSwitch g_sUISwitches[ ]
- const unsigned long g_ulUINumButtons
- static tStepperStatus ∗ pStepperStatus
- static unsigned long ulPotPosition

8.2.1 Define Documentation

8.2.1.1 ABS

An absolute value macro, provided here to avoid the need to pull in the library.

Definition:

```
#define ABS(n)
```

Parameters:

n a signed value

Description:

This macro returns the absolute value of whatever value is passed to it.

Returns:

Returns the absolute value of the input value.

8.2.1.2 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.

8.2.1.3 UI_INT_RATE

Definition:

```
#define UI_INT_RATE
```

Description:

The rate at which the user interface interrupt occurs (SysTick).

8.2.1.4 UI_POT_MAX

Definition:

```
#define UI_POT_MAX
```

Description:

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

8.2.1.5 UI_POT_MIN

Definition:

```
#define UI_POT_MIN
```

Description:

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

8.2.2 Enumeration Documentation

8.2.2.1 tUIMode

Description:

Definition and mode for the on-board user interface.

Enumerators:

UI_MODE_SPEED Defines speed mode, where the motor runs at a speed determined by the position of the knob.

UI_MODE_POSITION Defines position mode, there the motor moves to a position to match the position of the knob.

NUM_UI_MODES The number of user interface modes.

8.2.3 Function Documentation

8.2.3.1 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.

8.2.3.2 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. This causes the stepper on-board interface to switch modes.

Returns:
None.

8.2.3.3 UIButtonPress [static]

Handles button presses.

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

Description:
This function is called when a press of the on-board push button has been detected.

Returns:
None.

8.2.3.4 UIClearFaults

Clears a fault condition.

Prototype:
```
void
UIClearFaults(void)
```

Description:
Once a fault condition occurs, the motor will not run until the fault is cleared. This function clears any faults.

Returns:
None.

8.2.3.5 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. This will immediately remove all power from the motor. It is not a controlled stop, and the position information will be lost. The motor will be disabled, and must be re-enabled before it can be used again.

Returns:

None.

8.2.3.6 UIInit

Initializes the user interface.

Prototype:

```
void
UIInit(void)
```

Description:

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

Returns:

None.

8.2.3.7 UIOnBoard

Switch User Interface modes between on-board and off-board

Prototype:

```
void
UIOnBoard(void)
```

Description:

This function is called by the serial user interface when the on-board interface enable flag is changed. If the on-board interface is being disabled, that means the user is using the off-board interface. If the on-board interface was being used, then the target and actual positions will have non-zero values, which could cause confusion when the PC based GUI program is started. So the target and actual positions are reset to 0 if the motor is not moving.

Returns:

None.

8.2.3.8 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. 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.

8.2.3.9 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.

8.2.3.10 UIRun

Enables 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 enabled as a result; this is a no operation if the motor drive is already running.

There is no immediate response when the motor is enabled, it must be followed by a motion command in order to make the motor move.

Returns:

None.

8.2.3.11 UISetChopperBlanking

Sets the chopper blanking intervals.

Prototype:

```
void
UISetChopperBlanking(void)
```

Description:
This function is called by the serial user interface whenever the chopper blanking parameters are changed.

Returns:
None.

8.2.3.12 UISetControlMode

Sets the motor control mode (method).

Prototype:

```
void
UISetControlMode(void)
```

Description:
This function is called by the serial user interface whenever the motor control mode is changed.

Returns:
None.

8.2.3.13 UISetDecayMode

Sets the current decay control mode.

Prototype:

```
void
UISetDecayMode(void)
```

Description:
This function is called by the serial user interface whenever the current decay mode is changed.

Returns:
None.

8.2.3.14 UISetFaultParms

Sets up fault conditions whenever a fault triggering parameter is set.

Prototype:

```
void
UISetFaultParms(void)
```

Description:
This function is called by the serial user interface whenever the maximum current parameter is changed. This parameter is used to set up a fault trigger, which will trip if the current exceeds the specified value.

Returns:
None.

8.2.3.15 UISetFixedOnTime

Sets the PWM fixed on time.

Prototype:
```
void
UISetFixedOnTime(void)
```

Description:
This function is called by the serial user interface whenever the fixed on time parameter is changed.

Returns:
None.

8.2.3.16 UISetMotion

Commands the stepper drive to make a motion.

Prototype:
```
void
UISetMotion(void)
```

Description:
This function is called by the serial user interface whenever any of the following parameters are changed: target position, speed and acceleration or deceleration rates. This will immediately change the motor motion to match the new parameters.

If the motor was not first enabled, then this will have no effect.

Returns:
None.

8.2.3.17 UISetMotorParms

Sets the motor parameters related to winding current.

Prototype:
```
void
UISetMotorParms(void)
```

Description:
This function is called by the serial user interface whenever the motor drive or holding current parameters are changed.

Returns:
None.

8.2.3.18 UISetPWMFreq

Sets the PWM frequency.

Prototype:
```
void
UISetPWMFreq(void)
```

Description:
This function is called by the serial user interface whenever the PWM frequency parameter is changed.

Returns:
None.

8.2.3.19 UISetStepMode

Sets the step size mode.

Prototype:
```
void
UISetStepMode(void)
```

Description:
This function is called by the serial user interface whenever the step size mode is changed.

Returns:
None.

8.2.3.20 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. After this, the motor will be disabled, and must be re-enabled before another motion command can be issued.

This will cause a controlled stop of the motor.

Returns:
None.

8.2.3.21 UIUpgrade

Update the firmware using the bootloader.

Prototype:
```
void
UIUpgrade(void)
```

Description:
This function is called by the serial user interface when the firmware update command is received.

Returns:
None.

8.2.4 Variable Documentation

8.2.4.1 bReverse

Definition:
```
unsigned char bReverse
```

Description:
A flag that is used to keep track of motor direction when using the on-board interface in speed mode.

8.2.4.2 eUIMode [static]

Definition:
```
static tUIMode eUIMode
```

Description:
Holds the state of the on-board interface: UI_MODE_SPEED, or UI_MODE_POSITION.

8.2.4.3 g_pulUIHoldCount

Definition:
```
unsigned long g_pulUIHoldCount
```

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

8.2.4.4 g_sUISwitches

Definition:

```
const tUIOnboardSwitch g_sUISwitches[]
```

Description:

An array of structures describing the on-board switches.

8.2.4.5 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.

8.2.4.6 pStepperStatus [static]

Definition:

```
static tStepperStatus *pStepperStatus
```

Description:

A pointer to the motion status information that is returned from the positioner module.

8.2.4.7 ulPotPosition [static]

Definition:

```
static unsigned long ulPotPosition
```

Description:

The value of the last read potentiometer position.

9 Parameters

9.1 Introduction

This module contains the parameters that are maintained and updated by the User Interface Module.

The code for implementing UI Parameters is contained in `uiparms.c`, with `uiparms.h` containing the definitions for the variables and functions exported to the application.

9.2 Definitions

Data Structures

- tDriveParameters

Variables

- unsigned char g_bUIUseOnboard
- long g_lTargetPos
- short g_sAmbientTemp
- tDriveParameters g_sParameters
- const tUIParameter g_sUIParameters[ ]
- const tUIRealTimeData g_sUIRealTimeData[ ]
- unsigned char g_ucCPUUsage
- const unsigned long g_ulUINumParameters
- const unsigned long g_ulUINumRealTimeData
- const unsigned long g_ulUITargetType
- unsigned short g_usBusVoltage
- const unsigned short g_usFirmwareVersion
- unsigned short g_usMotorCurrent

9.2.1 Data Structure Documentation

9.2.1.1 tDriveParameters

Definition:

```
typedef struct
{
    unsigned char ucSequenceNum;
    unsigned char ucCRC;
    unsigned char ucControlMode;
    unsigned char ucDecayMode;
    unsigned char ucStepMode;
    unsigned short usSpeed;
    unsigned short usAccel;
    unsigned short usDecel;
    unsigned short usFixedOnTime;
    unsigned short usPWMFrequency;
    unsigned short usBlankOffTime;
    unsigned short usDriveCurrent;
    unsigned short usHoldCurrent;
    unsigned short usMaxCurrent;
    unsigned short usResistance;
}
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).

ucControlMode The stepper control mode: CONTROL_MODE_PWM or CONTROL_MODE_CHOP.

ucDecayMode The current decay mode: DECAY_MODE_SLOW or DECAY_MODE_FAST.

ucStepMode The stepping mode: STEP_MODE_FULL, STEP_MODE_WAVE, STEP_MODE_HALF or STEP_MODE_MICRO.

usSpeed The running speed in steps/sec.

usAccel The acceleration in steps/sec**2.

usDecel The deceleration in steps/sec**2.

usFixedOnTime The "on" interval for fixed rise time in microseconds.

usPWMFrequency The PWM frequency, in Hz.

usBlankOffTime The "off" blanking interval for chopper mode in microseconds.

usDriveCurrent The driving current in milliamps.

usHoldCurrent The holding current in milliamps.

usMaxCurrent The maximum faulting current in milliamps.

usResistance The motor winding resistance in milliohms.

Description:

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

9.2.2 Variable Documentation

9.2.2.1 g_bUIUseOnboard

Definition:

```
unsigned char g_bUIUseOnboard
```

Description:
A flag to indicate if the on-board user interface should be used. This is normally disabled if the serial interface is used.

9.2.2.2 g_lTargetPos

Definition:

```
long g_lTargetPos
```

Description:
The target position for the motor motion that the UI is requesting.

9.2.2.3 g_sAmbientTemp

Definition:

```
short g_sAmbientTemp
```

Description:
The internal temperature of the microcontroller in deg C.

9.2.2.4 g_sParameters

Definition:

```
tDriveParameters g_sParameters
```

Description:
This structure instance contains the configuration values for the stepper motor drive. This is where the initial default values for all the parameters are set. However, the default values can be changed by saving an update to flash memory.

9.2.2.5 g_sUIParameters

Definition:

```
const tUIParameter g_sUIParameters[]
```

Description:
An array of structures describing the stepper motor drive parameters to the serial user interface module. This table contains all of the parameters including the size, limits, and resolution for each.

9.2.2.6 g_sUIRealTimeData

Definition:

```
const tUIRealTimeData g_sUIRealTimeData[]
```

Description:

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

9.2.2.7 g_ucCPUUsage

Definition:

```
unsigned char g_ucCPUUsage
```

Description:

The CPU usage in percent.

9.2.2.8 g_ulUINumParameters

Definition:

```
const unsigned long g_ulUINumParameters
```

Description:

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

9.2.2.9 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.

9.2.2.10 g_ulUITargetType

Definition:

```
const unsigned long g_ulUITargetType
```

Description:

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

9.2.2.11 g_usBusVoltage

Definition:

```
unsigned short g_usBusVoltage
```

Description:
The bus voltage that was measured by the ADC in millivolts.

9.2.2.12 g_usFirmwareVersion

Definition:

```
const unsigned short g_usFirmwareVersion
```

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

9.2.2.13 g_usMotorCurrent

Definition:

```
unsigned short g_usMotorCurrent
```

Description:
The peak current of the two motor windings averaged together. This is the peak winding current when the chopper mode is used. The units are in milliamps.

10 Main

10.1 Introduction

This is the application's main entry point. It provides default handlers for NMI, Fault, and unhandled interrupts. It also provides the C main() entry point.

This module just provides a bare minimum hardware initialization, and then calls the initialization function for the User Interface. After that it stays in a loop, putting the processor to sleep when no interrupt is being serviced.

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

10.2 Definitions

Functions

- void FaultISR (void)
- void IntDefaultHandler (void)
- int main (void)
- void NmiSR (void)

10.2.1 Function Documentation

10.2.1.1 FaultISR

This is the code that gets called when the processor receives a fault interrupt. This simply enters an infinite loop, preserving the system state for examination by a debugger.

Prototype:

```
void
FaultISR(void)
```

Returns:
None.

10.2.1.2 void IntDefaultHandler (void)

This is the code that gets called when the processor receives an unexpected interrupt. This simply enters an infinite loop, preserving the system state for examination by a debugger.

Returns:
None.

10.2.1.3 int main (void)

The application main entry point.

This is where the program starts running. It initializes the system clock. Then it calls the User Interface module initialization function, UIInit(). This will cause all the other modules to be initialized. Finally, it enters an infinite loop, sleeping while waiting for an interrupt to occur.

Returns:
None.

10.2.1.4 void NmiSR (void)

This is the code that gets called when the processor receives a NMI. This simply enters an infinite loop, preserving the system state for examination by a debugger.

Returns:
None.

11 LED Blinker

11.1 Introduction

The blinker module provides a set of functions and a processing state machine to assist with making LEDs blink in a certain pattern or rate. The LED can be made to blink on for a certain time, then off for a certain time, and can repeat that on-off cycle a number of times.

To use the blinker module, a client should initialize the GPIO port pins that will be used. The blinker module does not perform any initializations of GPIO hardware. Then BlinkerInit() should be called once for each LED that will be controlled by this module.

Finally, BlinkerStart() can be used to start a blinking pattern on a specific LED, and BlinkerUpdate() can be used to change the pattern on an already blinking LED.

The blinker state machine runs by periodic calls to the BlinkHandler() function. The client should call this function on a regular interval to keep the blinkers running.

The code for implementing the LED Blinker API is contained in `blinker.c`, with `blinker.h` containing the definitions for the variables and functions exported to the application.

11.2 Definitions

Data Structures

- tBlinker

Defines

- NUM_BLINKERS

Enumerations

- tBlinkState

Functions

- void BlinkHandler (void)
- void BlinkInit (unsigned long ulIdx, unsigned long ulPort, unsigned long ulPin)
- void BlinkStart (unsigned long ulIdx, unsigned long ulOn, unsigned long ulOff, unsigned long ulRepeat)

- void BlinkUpdate (unsigned long ulIdx, unsigned long ulOn, unsigned long ulOff)

Variables

- tBlinker sBlinker[NUM_BLINKERS]

11.2.1 Data Structure Documentation

11.2.1.1 tBlinker

Definition:

```
typedef struct
{
    tBlinkState eState;
    unsigned long ulPort;
    unsigned long ulPin;
    unsigned long ulOnCount;
    unsigned long ulOffCount;
    unsigned long ulOnLoad;
    unsigned long ulOffLoad;
    unsigned long ulRepeat;
}
tBlinker
```

Members:

eState The state of blinking state machine for this LED.
ulPort The GPIO port base address for this LED.
ulPin The GPIO pin (bit mask) for this LED.
ulOnCount The number of counts remaining for the LED to be on.
ulOffCount The number of counts remaining for the LED to be off.
ulOnLoad The reload value for the on counts.
ulOffLoad The reload value for the off counts.
ulRepeat The number of times to repeat the pattern (should be >= 1).

Description:

This structure contains the parameters associated with making an LED blink according to a certain pattern.

11.2.2 Define Documentation

11.2.2.1 NUM_BLINKERS

Definition:

```
#define NUM_BLINKERS
```

Description:

Defines the number of LED blinker instances that are supported.

11.2.3 Enumeration Documentation

11.2.3.1 tBlinkState

Description:
Defines the states that the LED blinking state machine can be in.

Enumerators:

BLINK_IDLE The blinker is not doing anything.

BLINK_START The blinker has been started and is in the initial state.

BLINK_ON The blinker is turned on and counting down to the end of the on period.

BLINK_OFF The blinker is turned off and counting down to the end of the off period.

11.2.4 Function Documentation

11.2.4.1 BlinkHandler

Runs the blinker state machine through one state. Should be called periodically.

Prototype:
```
void
BlinkHandler(void)
```

Description:
This function runs the blinker state machine through one state cycle. It is intended to be called periodically in order to keep the blinker state machine running. One call to this function represents one on or off count as specified by the BlinkStart() or BlinkUpdate() functions.

Returns:
None.

11.2.4.2 BlinkInit

Initializes a blinker instance.

Prototype:
```
void
BlinkInit(unsigned long ulIdx,
          unsigned long ulPort,
          unsigned long ulPin)
```

Parameters:

ulIdx is the index of the blinker instance. Blinker instances are managed by the caller.

ulPort is the base address of the GPIO port for this LED blinker.

ulPin is the bit mask (one bit set) of the position of the LED on the GPIO port.

Description:
This function is used to set up a blinker instance to control a specific GPIO pin (which presumably is driving an LED). The caller specifies the GPIO port, and the pin on that port. The pin is specified as a bit mask, not as the pin number.

The blinker instances must be managed by the caller. This module does not keep track of which instances are used.

Returns:
None.

11.2.4.3 BlinkStart

Starts a blinker blinking according to the specified pattern.

Prototype:
```
void
BlinkStart(unsigned long ulIdx,
           unsigned long ulOn,
           unsigned long ulOff,
           unsigned long ulRepeat)
```

Parameters:
ulIdx is the index of the blinker instance. Blinker instances are managed by the caller.
ulOn is the number of counts for the LED to be turned on.
ulOff is the number of counts for the LED to be turned off.
ulRepeat is the number of times to repeat the on-off pattern (should be at least 1).

Description:
This function starts the LED blinking according to the specified pattern. The LED will be turned on for the number of counts specified by ulOn, then off for the number of counts specified by ulOff. The on-off cycle will repeat for the number of times specified by ulRepeat. ulRepeat must be 1 to get the pattern to blink once.

If ulOn is 0, then the LED will just be turned off, likewise it will be turned on if ulOff is 0.

The number of counts is the number of times that BlinkHandler() is called by the client.

Returns:
None.

11.2.4.4 BlinkUpdate

Updates the blink on and off counts while the blinker is blinking.

Prototype:
```
void
BlinkUpdate(unsigned long ulIdx,
            unsigned long ulOn,
            unsigned long ulOff)
```

Parameters:
ulIdx is the index of the blinker instance. Blinker instances are managed by the caller.
ulOn is the number of counts for the LED to be turned on.
ulOff is the number of counts for the LED to be turned off.

Description:
This function is used to update the on and off counts of a blinker that is already blinking. This can be used in the case when a high repeat count has been used in order to provide the appearance of continuous blinking. Using the update function can change the blinking rate without a visible glitch, which might happen if the BlinkStart() were used again.

Returns:
None.

11.2.5 Variable Documentation

11.2.5.1 sBlinker

Definition:

```
tBlinker sBlinker[NUM_BLINKERS]
```

Description:
The blinker instances. There should be one for each LED to be managed by blinker.

12 On-board User Interface

12.1 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.

12.2 Definitions

Data Structures

- tUIOnboardSwitch

Functions

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

Variables

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

12.2.1 Data Structure Documentation

12.2.1.1 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.

12.2.2 Function Documentation

12.2.2.1 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.

12.2.2.2 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.

12.2.2.3 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.

12.2.3 Variable Documentation

12.2.3.1 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.

12.2.3.2 g_ulUIOnboardClockB [static]

Definition:

```
static unsigned long g_ulUIOnboardClockB
```

Description:

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

12.2.3.3 g_ulUIOnboardFilteredPotValue [static]

Definition:

```
static unsigned long g_ulUIOnboardFilteredPotValue
```

Description:

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

12.2.3.4 g_ulUIOnboardPotCount [static]

Definition:

```
static unsigned long g_ulUIOnboardPotCount
```

Description:

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

12.2.3.5 g_ulUIOnboardPotMax [static]

Definition:

```
static unsigned long g_ulUIOnboardPotMax
```

Description:

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

12.2.3.6 g_ulUIOnboardPotMin [static]

Definition:

```
static unsigned long g_ulUIOnboardPotMin
```

Description:

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

12.2.3.7 g_ulUIOnboardPotSum [static]

Definition:

```
static unsigned long g_ulUIOnboardPotSum
```

Description:

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

12.2.3.8 g_ulUIOnboardPotValue [static]

Definition:

```
static unsigned long g_ulUIOnboardPotValue
```

Description:

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

12.2.3.9 g_ulUIOnboardSwitches [static]

Definition:

```
static unsigned long g_ulUIOnboardSwitches
```

Description:

The debounced state of the switches.

13 Serial Interface

13.1 Introduction

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

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

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

13.1.1 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
```

13.1.2 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}
```

13.1.3 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.

13.1.4 Interface To The Application

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

13.2 Definitions

Defines

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

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

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

Functions

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

Variables

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

13.2.1 Define Documentation

13.2.1.1 CMD_DISABLE_DATA_ITEM

Definition:

```
#define CMD_DISABLE_DATA_ITEM
```

Description:

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

Command:

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

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

Response:

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

13.2.1.2 CMD_DISCOVER_TARGET

Definition:

```
#define CMD_DISCOVER_TARGET
```

Description:

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

Command:

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

Response:

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

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

13.2.1.3 CMD_EMERGENCY_STOP

Definition:

```
#define CMD_EMERGENCY_STOP
```

Description:

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

Command:

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

Response:

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

13.2.1.4 CMD_ENABLE_DATA_ITEM

Definition:

```
#define CMD_ENABLE_DATA_ITEM
```

Description:

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

Command:

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

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

Response:

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

13.2.1.5 CMD_GET_DATA_ITEMS

Definition:
`#define CMD_GET_DATA_ITEMS`

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

Command:

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

Response:

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

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

13.2.1.6 CMD_GET_PARAM_DESC

Definition:
`#define CMD_GET_PARAM_DESC`

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

Command:

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

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

Response:

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

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

13.2.1.7 CMD_GET_PARAM_VALUE

Definition:

`#define CMD_GET_PARAM_VALUE`

Description:

Gets the value of a parameter.

Command:

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

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

Response:

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

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

13.2.1.8 CMD_GET_PARAMS

Definition:

`#define CMD_GET_PARAMS`

Description:

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

Command:

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

Response:

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

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

13.2.1.9 CMD_ID_TARGET

Definition:

`#define CMD_ID_TARGET`

Description:

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

Command:

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

Response:

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

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

13.2.1.10 CMD_LOAD_PARAMS

Definition:
`#define CMD_LOAD_PARAMS`

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

Command:

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

Response:

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

13.2.1.11 CMD_RUN

Definition:
`#define CMD_RUN`

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

Command:

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

Response:

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

13.2.1.12 CMD_SAVE_PARAMS

Definition:
`#define CMD_SAVE_PARAMS`

Description:

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

Command:

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

Response:

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

13.2.1.13 CMD_SET_PARAM_VALUE

Definition:

`#define CMD_SET_PARAM_VALUE`

Description:

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

Command:

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

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

Response:

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

13.2.1.14 CMD_START_DATA_STREAM

Definition:

`#define CMD_START_DATA_STREAM`

Description:

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

Command:

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

Response:

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

13.2.1.15 CMD_STOP

Definition:

```
#define CMD_STOP
```

Description:

Stops the motor, if it is not already stopped.

Command:

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

Response:

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

13.2.1.16 CMD_STOP_DATA_STREAM

Definition:

```
#define CMD_STOP_DATA_STREAM
```

Description:

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

Command:

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

Response:

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

13.2.1.17 CMD_UPGRADE

Definition:

```
#define CMD_UPGRADE
```

Description:

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

Command:

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

Response:

```
<none>
```

13.2.1.18 DATA_ANALOG_INPUT

Definition:

```
#define DATA_ANALOG_INPUT
```

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

13.2.1.19 DATA_BUS_VOLTAGE

Definition:

```
#define DATA_BUS_VOLTAGE
```

Description:
This real-time data item provides the bus voltage.

13.2.1.20 DATA_DEBUG_INFO

Definition:

```
#define DATA_DEBUG_INFO
```

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

13.2.1.21 DATA_DIRECTION

Definition:

```
#define DATA_DIRECTION
```

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

13.2.1.22 DATA_FAULT_STATUS

Definition:

```
#define DATA_FAULT_STATUS
```

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

13.2.1.23 DATA_MOTOR_CURRENT

Definition:

```
#define DATA_MOTOR_CURRENT
```

Description:

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

13.2.1.24 DATA_MOTOR_POSITION

Definition:

```
#define DATA_MOTOR_POSITION
```

Description:

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

13.2.1.25 DATA_MOTOR_POWER

Definition:

```
#define DATA_MOTOR_POWER
```

Description:

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

13.2.1.26 DATA_MOTOR_STATUS

Definition:

```
#define DATA_MOTOR_STATUS
```

Description:

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

13.2.1.27 DATA_NUM_ITEMS

Definition:

```
#define DATA_NUM_ITEMS
```

Description:

The number of real-time data items.

13.2.1.28 DATA_PHASE_A_CURRENT

Definition:

```
#define DATA_PHASE_A_CURRENT
```

Description:

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

13.2.1.29 DATA_PHASE_B_CURRENT

Definition:

```
#define DATA_PHASE_B_CURRENT
```

Description:

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

13.2.1.30 DATA_PHASE_C_CURRENT

Definition:

```
#define DATA_PHASE_C_CURRENT
```

Description:

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

13.2.1.31 DATA_PROCESSOR_USAGE

Definition:

```
#define DATA_PROCESSOR_USAGE
```

Description:

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

13.2.1.32 DATA_ROTOR_SPEED

Definition:

```
#define DATA_ROTOR_SPEED
```

Description:

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

13.2.1.33 DATA_STATOR_SPEED

Definition:

```
#define DATA_STATOR_SPEED
```

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

13.2.1.34 DATA_TEMPERATURE

Definition:
```
#define DATA_TEMPERATURE
```

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

13.2.1.35 MOTOR_STATUS_ACCEL

Definition:
```
#define MOTOR_STATUS_ACCEL
```

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

13.2.1.36 MOTOR_STATUS_DECEL

Definition:
```
#define MOTOR_STATUS_DECEL
```

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

13.2.1.37 MOTOR_STATUS_RUN

Definition:
```
#define MOTOR_STATUS_RUN
```

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

13.2.1.38 MOTOR_STATUS_STOP

Definition:
```
#define MOTOR_STATUS_STOP
```

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

13.2.1.39 PARAM_ACCEL

Definition:
```
#define PARAM_ACCEL
```

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

13.2.1.40 PARAM_ACCEL_CURRENT

Definition:
```
#define PARAM_ACCEL_CURRENT
```

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

13.2.1.41 PARAM_ACCEL_POWER

Definition:
```
#define PARAM_ACCEL_POWER
```

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

13.2.1.42 PARAM_BEMF_SKIP_COUNT

Definition:
```
#define PARAM_BEMF_SKIP_COUNT
```

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

13.2.1.43 PARAM_BLANK_OFF

Definition:
```
#define PARAM_BLANK_OFF
```

Description:
Specifies the blanking time after the current is removed.

13.2.1.44 PARAM_BRAKE_COOL_TIME

Definition:
```
#define PARAM_BRAKE_COOL_TIME
```

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

13.2.1.45 PARAM_BRAKE_OFF_VOLTAGE

Definition:

```
#define PARAM_BRAKE_OFF_VOLTAGE
```

Description:

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

13.2.1.46 PARAM_BRAKE_ON_VOLTAGE

Definition:

```
#define PARAM_BRAKE_ON_VOLTAGE
```

Description:

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

13.2.1.47 PARAM_CAN_RX_COUNT

Definition:

```
#define PARAM_CAN_RX_COUNT
```

Description:

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

13.2.1.48 PARAM_CAN_TX_COUNT

Definition:

```
#define PARAM_CAN_TX_COUNT
```

Description:

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

13.2.1.49 PARAM_CLOSED_LOOP

Definition:

```
#define PARAM_CLOSED_LOOP
```

Description:

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

13.2.1.50 PARAM_CONTROL_MODE

Definition:

```
#define PARAM_CONTROL_MODE
```

Description:

Specifies the motor control mode.

13.2.1.51 PARAM_CURRENT_POS

Definition:

```
#define PARAM_CURRENT_POS
```

Description:

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

13.2.1.52 PARAM_CURRENT_POWER

Definition:

```
#define PARAM_CURRENT_POWER
```

Description:

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

13.2.1.53 PARAM_CURRENT_SPEED

Definition:

```
#define PARAM_CURRENT_SPEED
```

Description:

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

13.2.1.54 PARAM_DATA_RATE

Definition:

```
#define PARAM_DATA_RATE
```

Description:

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

13.2.1.55 PARAM_DC_BRAKE_TIME

Definition:

```
#define PARAM_DC_BRAKE_TIME
```

Description:

Specifies the amount of time to apply DC injection braking.

13.2.1.56 PARAM_DC_BRAKE_V

Definition:

```
#define PARAM_DC_BRAKE_V
```

Description:

Specifies the voltage to be applied during DC injection braking.

13.2.1.57 PARAM_DECAY_MODE

Definition:

```
#define PARAM_DECAY_MODE
```

Description:

Specifies the motor winding current decay mode.

13.2.1.58 PARAM_DECEL

Definition:

```
#define PARAM_DECEL
```

Description:

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

13.2.1.59 PARAM_DECEL_POWER

Definition:

```
#define PARAM_DECEL_POWER
```

Description:

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

13.2.1.60 PARAM_DECEL_VOLTAGE

Definition:

```
#define PARAM_DECEL_VOLTAGE
```

Description:

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

13.2.1.61 PARAM_DIRECTION

Definition:

```
#define PARAM_DIRECTION
```

Description:

Specifies the direction of rotation for the motor.

13.2.1.62 PARAM_ENCODER_PRESENT

Definition:

```
#define PARAM_ENCODER_PRESENT
```

Description:

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

13.2.1.63 PARAM_ETH_RX_COUNT

Definition:

```
#define PARAM_ETH_RX_COUNT
```

Description:

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

13.2.1.64 PARAM_ETH_TCP_TIMEOUT

Definition:

```
#define PARAM_ETH_TCP_TIMEOUT
```

Description:

The timeout for an IDLE TCP connection.

13.2.1.65 PARAM_ETH_TX_COUNT

Definition:

```
#define PARAM_ETH_TX_COUNT
```

Description:

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

13.2.1.66 PARAM_FAULT_STATUS

Definition:

```
#define PARAM_FAULT_STATUS
```

Description:

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

13.2.1.67 PARAM_FIRMWARE_VERSION

Definition:

```
#define PARAM_FIRMWARE_VERSION
```

Description:

Specifies the version of the firmware on the motor drive.

13.2.1.68 PARAM_FIXED_ON_TIME

Definition:

```
#define PARAM_FIXED_ON_TIME
```

Description:

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

13.2.1.69 PARAM_GPIO_DATA

Definition:

```
#define PARAM_GPIO_DATA
```

Description:

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

13.2.1.70 PARAM_HOLDING_CURRENT

Definition:

```
#define PARAM_HOLDING_CURRENT
```

Description:

Specifies the motor winding holding current.

13.2.1.71 PARAM_MAX_BRAKE_TIME

Definition:

```
#define PARAM_MAX_BRAKE_TIME
```

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

13.2.1.72 PARAM_MAX_BUS_VOLTAGE

Definition:
```
#define PARAM_MAX_BUS_VOLTAGE
```

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

13.2.1.73 PARAM_MAX_CURRENT

Definition:
```
#define PARAM_MAX_CURRENT
```

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

13.2.1.74 PARAM_MAX_POWER

Definition:
```
#define PARAM_MAX_POWER
```

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

13.2.1.75 PARAM_MAX_SPEED

Definition:
```
#define PARAM_MAX_SPEED
```

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

13.2.1.76 PARAM_MAX_TEMPERATURE

Definition:
```
#define PARAM_MAX_TEMPERATURE
```

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

13.2.1.77 PARAM_MIN_BUS_VOLTAGE

Definition:

```
#define PARAM_MIN_BUS_VOLTAGE
```

Description:

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

13.2.1.78 PARAM_MIN_CURRENT

Definition:

```
#define PARAM_MIN_CURRENT
```

Description:

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

13.2.1.79 PARAM_MIN_POWER

Definition:

```
#define PARAM_MIN_POWER
```

Description:

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

13.2.1.80 PARAM_MIN_SPEED

Definition:

```
#define PARAM_MIN_SPEED
```

Description:

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

13.2.1.81 PARAM_MODULATION

Definition:

```
#define PARAM_MODULATION
```

Description:

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

13.2.1.82 PARAM_MOTOR_STATUS

Definition:

```
#define PARAM_MOTOR_STATUS
```

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

13.2.1.83 PARAM_MOTOR_TYPE

Definition:

```
#define PARAM_MOTOR_TYPE
```

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

13.2.1.84 PARAM_NUM_LINES

Definition:

```
#define PARAM_NUM_LINES
```

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

13.2.1.85 PARAM_NUM_POLES

Definition:

```
#define PARAM_NUM_POLES
```

Description:
Specifies the number of pole pairs in the motor.

13.2.1.86 PARAM_POWER_I

Definition:

```
#define PARAM_POWER_I
```

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

13.2.1.87 PARAM_POWER_P

Definition:

```
#define PARAM_POWER_P
```

Description:

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

13.2.1.88 PARAM_PRECHARGE_TIME

Definition:

```
#define PARAM_PRECHARGE_TIME
```

Description:

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

13.2.1.89 PARAM_PWM_DEAD_TIME

Definition:

```
#define PARAM_PWM_DEAD_TIME
```

Description:

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

13.2.1.90 PARAM_PWM_FREQUENCY

Definition:

```
#define PARAM_PWM_FREQUENCY
```

Description:

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

13.2.1.91 PARAM_PWM_MIN_PULSE

Definition:

```
#define PARAM_PWM_MIN_PULSE
```

Description:

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

13.2.1.92 PARAM_PWM_UPDATE

Definition:

```
#define PARAM_PWM_UPDATE
```

Description:

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

13.2.1.93 PARAM_RESISTANCE

Definition:

```
#define PARAM_RESISTANCE
```

Description:

Specifies the winding resistance.

13.2.1.94 PARAM_SENSOR_POLARITY

Definition:

```
#define PARAM_SENSOR_POLARITY
```

Description:

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

13.2.1.95 PARAM_SENSOR_PRESENT

Definition:

```
#define PARAM_SENSOR_PRESENT
```

Description:

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

13.2.1.96 PARAM_SENSOR_TYPE

Definition:

```
#define PARAM_SENSOR_TYPE
```

Description:

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

13.2.1.97 PARAM_SPEED_I

Definition:

```
#define PARAM_SPEED_I
```

Description:

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

13.2.1.98 PARAM_SPEED_P

Definition:

```
#define PARAM_SPEED_P
```

Description:

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

13.2.1.99 PARAM_STARTUP_COUNT

Definition:

```
#define PARAM_STARTUP_COUNT
```

Description:

Indicates the startup count for sensorless operation.

13.2.1.100PARAM_STARTUP_DUTY

Definition:

```
#define PARAM_STARTUP_DUTY
```

Description:

Indicates the duty cycle for startup phase.

13.2.1.101PARAM_STARTUP_ENDSP

Definition:

```
#define PARAM_STARTUP_ENDSP
```

Description:

Contains the ending speed for sensorless startup operation.

13.2.1.102 PARAM_STARTUP_ENDV

Definition:

```
#define PARAM_STARTUP_ENDV
```

Description:

Contains the ending voltage for sensorless startup operation.

13.2.1.103 PARAM_STARTUP_RAMP

Definition:

```
#define PARAM_STARTUP_RAMP
```

Description:

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

13.2.1.104 PARAM_STARTUP_STARTSP

Definition:

```
#define PARAM_STARTUP_STARTSP
```

Description:

Contains the starting speed for sensorless startup operation.

13.2.1.105 PARAM_STARTUP_STARTV

Definition:

```
#define PARAM_STARTUP_STARTV
```

Description:

Contains the starting voltage for sensorless startup operation.

13.2.1.106 PARAM_STARTUP_THRESH

Definition:

```
#define PARAM_STARTUP_THRESH
```

Description:

Contains the back EMF threshhold voltage for sensorless startup.

13.2.1.107 PARAM_STEP_MODE

Definition:

```
#define PARAM_STEP_MODE
```

Description:

Specifies the motor stepping mode.

13.2.1.108 PARAM_TARGET_CURRENT

Definition:

```
#define PARAM_TARGET_CURRENT
```

Description:
Specifies the target running current of the motor.

13.2.1.109 PARAM_TARGET_POS

Definition:

```
#define PARAM_TARGET_POS
```

Description:
Specifies the target position of the motor.

13.2.1.110 PARAM_TARGET_POWER

Definition:

```
#define PARAM_TARGET_POWER
```

Description:
Specifies the target power supplied to the motor when operating.

13.2.1.111 PARAM_TARGET_SPEED

Definition:

```
#define PARAM_TARGET_SPEED
```

Description:
Specifies the desired speed of the the motor.

13.2.1.112 PARAM_USE_BUS_COMP

Definition:

```
#define PARAM_USE_BUS_COMP
```

Description:
Specifies whether DC bus voltage compensation should be performed.

13.2.1.113 PARAM_USE_DC_BRAKE

Definition:

```
#define PARAM_USE_DC_BRAKE
```

Description:
Specifies whether DC injection braking should be performed.

13.2.1.114 PARAM_USE_DYNAM_BRAKE

Definition:

```
#define PARAM_USE_DYNAM_BRAKE
```

Description:
Specifies whether dynamic braking should be performed.

13.2.1.115 PARAM_USE_ONBOARD_UI

Definition:

```
#define PARAM_USE_ONBOARD_UI
```

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

13.2.1.116 PARAM_VF_RANGE

Definition:

```
#define PARAM_VF_RANGE
```

Description:
Specifies the range of the V/f table.

13.2.1.117 PARAM_VF_TABLE

Definition:

```
#define PARAM_VF_TABLE
```

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

13.2.1.118 RESP_ID_TARGET_ACIM

Definition:

```
#define RESP_ID_TARGET_ACIM
```

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

13.2.1.119 RESP_ID_TARGET_BLDC

Definition:

```
#define RESP_ID_TARGET_BLDC
```

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

13.2.1.120 RESP_ID_TARGET_STEPPER

Definition:

```
#define RESP_ID_TARGET_STEPPER
```

Description:

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

13.2.1.121 TAG_CMD

Definition:

```
#define TAG_CMD
```

Description:

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

13.2.1.122 TAG_DATA

Definition:

```
#define TAG_DATA
```

Description:

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

13.2.1.123 TAG_STATUS

Definition:

```
#define TAG_STATUS
```

Description:

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

13.2.1.124 UISERIAL_MAX_RECV

Definition:

```
#define UISERIAL_MAX_RECV
```

Description:

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

13.2.1.125UISERIAL_MAX_XMIT

Definition:

```
#define UISERIAL_MAX_XMIT
```

Description:

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

13.2.2 Function Documentation

13.2.2.1 UART0IntHandler

Handles the UART interrupt.

Prototype:

```
void
UART0IntHandler(void)
```

Description:

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

Returns:

None.

13.2.2.2 UISerialFindParameter [static]

Finds a parameter by ID.

Prototype:

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

Parameters:

ucID is the ID of the parameter to locate.

Description:

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

Returns:

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

13.2.2.3 UISerialInit

Initializes the serial user interface.

Prototype:
```
void
UISerialInit(void)
```

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

Returns:
None.

13.2.2.4 UISerialRangeCheck [static]

Performs range checking on the value of a parameter.

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

Parameters:
ulIdx is the index of the parameter to check.

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

Returns:
None.

13.2.2.5 UISerialScanReceive [static]

Scans for packets in the receive buffer.

Prototype:
```
static void
UISerialScanReceive(void)
```

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

Returns:
None.

13.2.2.6 UISerialSendRealTimeData

Sends a real-time data packet.

Prototype:

```
void
UISerialSendRealTimeData(void)
```

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

Returns:
None.

13.2.2.7 UISerialTransmit [static]

Transmits a packet to the UART.

Prototype:

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

Parameters:
pucBuffer is a pointer to the packet to be transmitted.

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

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

13.2.3 Variable Documentation

13.2.3.1 g_bEnableRealTimeData [static]

Definition:

```
static tBoolean g_bEnableRealTimeData
```

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

13.2.3.2 g_pucUISerialData [static]

Definition:

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

Description:

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

13.2.3.3 g_pucUISerialReceive [static]

Definition:

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

Description:

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

13.2.3.4 g_pucUISerialResponse [static]

Definition:

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

Description:

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

13.2.3.5 g_pucUISerialTransmit [static]

Definition:

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

Description:

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

13.2.3.6 g_pulUIRealTimeData [static]

Definition:

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

Description:

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

13.2.3.7 g_ulUISerialReceiveRead [static]

Definition:

```
static unsigned long g_ulUISerialReceiveRead
```

Description:

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

13.2.3.8 g_ulUISerialReceiveWrite [static]

Definition:

```
static unsigned long g_ulUISerialReceiveWrite
```

Description:

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

13.2.3.9 g_ulUISerialTransmitRead [static]

Definition:

```
static unsigned long g_ulUISerialTransmitRead
```

Description:

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

13.2.3.10 g_ulUISerialTransmitWrite [static]

Definition:

```
static unsigned long g_ulUISerialTransmitWrite
```

Description:

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

14 CPU Usage Module

14.1 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.

14.2 API Functions

Functions

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

14.2.1 Function Documentation

14.2.1.1 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.

14.2.1.2 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.

14.3 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();
    }
}
```

15 Flash Parameter Block Module

15.1 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.

15.2 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)

15.2.1 Function Documentation

15.2.1.1 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.

15.2.1.2 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.

15.2.1.3 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.

15.2.1.4 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.

15.3 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);
}
```

16 Integer Square Root Module

16.1 Introduction

The integer square root module provides an integer version of the square root operation that can be used instead of the floating point version provided in the C library. The algorithm used is a derivative of the manual pencil-and-paper method that used to be taught in school, and is closely related to the pencil-and-paper division method that is likely still taught in school.

For full details of the algorithm, see the article by Jack W. Crenshaw in the February 1998 issue of Embedded System Programming. It can be found online at `http://www.embedded.com/98/9802fe2.htm`.

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

16.2 API Functions

Functions

- unsigned long isqrt (unsigned long ulValue)

16.2.1 Function Documentation

16.2.1.1 isqrt

Compute the integer square root of an integer.

Prototype:

```
unsigned long
isqrt(unsigned long ulValue)
```

Parameters:

ulValue is the value whose square root is desired.

Description:

This function will compute the integer square root of the given input value. Since the value returned is also an integer, it is actually better defined as the largest integer whose square is less than or equal to the input value.

Returns:

Returns the square root of the input value.

16.3 Programming Example

The following example shows how to compute the square root of a number.

```
unsigned long ulValue;

//
// Get the square root of 52378.  The result returned will be 228, which is
// the largest integer less than or equal to the square root of 52378.
//
ulValue = isqrt(52378);
```

IMPORTANT NOTICE

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
		TI E2E Community Home Page	e2e.ti.com