VEX 709S

Generated by Doxygen 1.8.14

Contents

Chapter 1

VEX-709S-2018

Repository for VEX Robotics Competition team 709S for the 2017-2018 game

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 debug Namespace Reference

Functions

- void debug (void)

Variables

- uint32_t **fault** = 0

6.1.1 Detailed Description

Contains debugging funtions, etc

6.1.2 Function Documentation

6.1.2.1 debug()

```
void debug::debug (
            void  )
```

Debug the Cortex if something goes wrong

6.2 drive Namespace Reference

Namespaces

- accel

Classes

- struct side_t

Functions

- void set (int lpower, int rpower)
- void init (void)
- void inches (long inches)
- void tank (void)

Variables

- double inch
- side_t left
- side_t right

6.2.1 Detailed Description

Contains everything relating to the drive

6.2.2 Function Documentation

6.2.2.1 inches()

```
void drive::inches (
            long inches )
```

Drive a specific number of inches

6.2.2.2 init()

```
void drive::init (
            void  )
```

Initialize the drive subsystem

6.2.2.3 set()

```
void drive::set (
            int lpower,
            int rpower )
```

Set both sides of the drive at their requested powers

6.2.2.4 tank()

```
void drive::tank (
            void  )
```

Tank control that should be used in a while loop

6.2.3 Variable Documentation

6.2.3.1 inch

```
double drive::inch
```

Initial value:

```
=
    28.647889756541160438399077407052585166202736233282160774580127 35
```

Multiplier for which 1 inch is used to convert into degreees rotation on 4" wheels

6.2.3.2 left

```
side_t drive::left
```

The left side of the drive

6.2.3.3 right

```
side_t drive::right
```

The right side of the drive

6.3 drive::accel Namespace Reference

Functions

- void drive (void)

Variables

- int x = 0
- int y = 0
- int prevX = 0
- int prevY = 0
- int **deadband** = 20

6.3.1 Detailed Description

Joystick accelerometer driving!

6.3.2 Function Documentation

6.3.2.1 drive()

```
void drive::accel::drive (
            void  )
```

Tilt control using the josytick accelerometer. Should be used in a while loop

6.3.3 Variable Documentation

6.3.3.1 prevX

```
int drive::accel::prevX = 0
```

Previous joystick accel x value

6.3.3.2 prevY

```
int drive::accel::prevY = 0
```

Previous joystick accel y value

6.3.3.3 x

```
int drive::accel::x = 0
```

Current x value of the joystick accel

6.3.3.4 y

```
int drive::accel::y = 0
```

Current y value of the joystick accel

6.4 lift Namespace Reference

Classes

- struct side_t

Enumerations

- enum position {
  bottom = 5, **mobile** = 60, **one** = 100, **two** = 230,
  three = 450 }

Functions

- void set (int lpower, int rpower)
- void init (void)
- void to (position pos=bottom, int int_pos=-1, int tolerance=50)
- void **set** (int power)

Variables

- double **inch**
- side_t left
- side_t right
- sensors::pot_t * sensor = &sensors::lift

6.4.1 Detailed Description

Contains everything relating to the drive

6.4.2 Enumeration Type Documentation

6.4.2.1 position

`enum lift::position`

Positions of the lift

6.4.3 Function Documentation

6.4.3.1 init()

```
void lift::init (
            void  )
```

Initialize the drive subsystem

6.4.3.2 set()

```
void lift::set (
            int lpower,
            int rpower )
```

Set both sides of the drive at their requested powers

6.4.3.3 to()

```
void lift::to (
            position pos = bottom,
            int int_pos = -1,
            int tolerance = 50 )
```

p control for the lift

6.4.4 Variable Documentation

6.4.4.1 left

```
side_t lift::left
```

The left side of the drive

6.4.4.2 right

```
side_t lift::right
```

The right side of the drive

6.4.4.3 sensor

```
sensors::pot_t * lift::sensor = &sensors::lift
```

Sensor on the lift

6.5 motors Namespace Reference

Namespaces

- slew

Functions

- void set (motor_t motor, int power)
- int get (motor_t motor)
- motor_t init (unsigned char port, int inverted, float slewRate, float scale)

6.5.1 Detailed Description

Namespace relating to the motors and setting them, initializing them, slewing, etc

6.5.2 Function Documentation

6.5.2.1 get()

```
int motors::get (
            motor_t motor )
```

Gets the current power value requested of the motor, analogous of motor.power

6.5.2.2 init()

```
motor_t motors::init (
            unsigned char port,
            int inverted,
            float slewRate,
            float scale )
```

Returns an initialized motor_t object with the specified parameters, and adds a duplicate of the motor to the motor list for slewing

6.5.2.3 set()

```
void motors::set (
            motor_t motor,
            int power )
```

Sets the motor to the power

6.6 motors::slew Namespace Reference

Functions

- void init (void)
- void **_slew** (void *none)

Variables

- motor_t list [11]
- TaskHandle handle

6.6.1 Detailed Description

Namespace relating to slewing the motors to save the gears and the PTCs

6.6.2 Function Documentation

6.6.2.1 init()

```
void motors::slew::init (
            void  )
```

Initialization function for slewing. Call in initialize()

6.6.3 Variable Documentation

6.6.3.1 handle

```
TaskHandle motors::slew::handle
```

The TaskHandle for handling the slewing task

6.6.3.2 list

```
motor_t motors::slew::list
```

The list of motors, as added to in motors::init()

6.7 pid Namespace Reference

Functions

- void enable (void)
- void disable (void)
- void controller (void *none)
- void init (void)
- void stop (void)
- void go (void)
- void request (long l, long r)
- void wait (unsigned long precision, unsigned long blockTime)

Variables

- float Kp = 0.8
- float Ki = 0.04
- float Kd = 0.35
- unsigned int default_precision = 30
- bool enabled [2] = {true, true}
- TaskHandle pidHandle
- unsigned int **deadband** = 10

6.7.1 Detailed Description

Consists of pid, and all subcomponents, etc

6.7.2 Function Documentation

6.7.2.1 controller()

```
void pid::controller (
            void * none )
```

Task to manage pid

6.7.2.2 disable()

```
void pid::disable (
            void  )
```

Disables all pid

6.7.2.3 enable()

```
void pid::enable (
            void  )
```

Enables all pid

6.7.2.4 go()

```
void pid::go (
            void  )
```

(Re)starts the pid task

6.7.2.5 init()

```
void pid::init (
            void  )
```

Initialize pid. Call in initialize()

6.7.2.6 request()

```
void pid::request (
            long l,
            long r )
```

Requests values for the left and right side of the drive

6.7.2.7 stop()

```
void pid::stop (
            void  )
```

Stops the pid task

6.7.2.8 wait()

```
void pid::wait (
            unsigned long precision,
            unsigned long blockTime )
```

Wait until pid reaches specified precision, for no longer than the specified blockTime. If 0 is passed to blockTime, it will wait indefinately until the requested values are met

6.7.3 Variable Documentation

6.7.3.1 default_precision

```
unsigned int pid::default_precision = 30
```

Default precision for waiting on pid to reach value

6.7.3.2 enabled

```
bool pid::enabled = {true, true}
```

Whether or not each side of the drive's pid is enabled, in the order of left to right

6.7.3.3 Kd

```
float pid::Kd = 0.35
```

d value

6.7.3.4 Ki

```
float pid::Ki = 0.04
```

i value

6.7.3.5 Kp

```
float pid::Kp = 0.8
```

p value

6.7.3.6 pidHandle

```
TaskHandle pid::pidHandle
```

TaskHandle for the pid task

6.8 sensors Namespace Reference

Classes

- struct button_t
- class gyro_t
- struct pot_t
- struct quad_t
- struct sonic_t

Functions

- void init (void)
- void reset (void)
- quad_t **left** (1, 2, false)
- quad_t **right** (3, 4, false)
- pot_t **lift** (1, false)
- gyro_t **gyro** (2, 197)

Variables

- quad_t left
- quad_t right
- pot_t lift
- gyro_t gyro

6.8.1 Detailed Description

The namespace containing all information, functions, objects, relating to sensors

6.8.2 Function Documentation

6.8.2.1 init()

```
void sensors::init (
            void  )
```

Initializes the sensor subsystem, calls all the funtions that need to be called in initialize(). Call in initialize()

6.8.2.2 reset()

```
void sensors::reset (
            void  )
```

Resets the important sensors

6.8.3 Variable Documentation

6.8.3.1 gyro

```
gyro_t sensors::gyro(2, 197)
```

gyro on the drive

6.8.3.2 left

```
quad_t sensors::left(1, 2, false)
```

left quad encoder on the drive

6.8.3.3 lift

```
pot_t sensors::lift(1, false)
```

potentiometer on the lift

6.8.3.4 right

```
quad_t sensors::right(3, 4, false)
```

right quad encoder on the drive

Chapter 7

Class Documentation

7.1 sensors::button_t Struct Reference

`#include <sensors.hpp>`

Public Member Functions

- bool value (void)
- void init (void)
- button_t (unsigned char _port, bool _inverted)

Public Attributes

- unsigned char port
- bool inverted

7.1.1 Detailed Description

Class for buttons

7.1.2 Constructor & Destructor Documentation

7.1.2.1 button_t()

```
sensors::button_t::button_t (
            unsigned char _port,
            bool _inverted )
```

Class constructor, but init() must also be called

7.1.3 Member Function Documentation

7.1.3.1 init()

```
void sensors::button_t::init (
            void  )
```

Initializes the button. Call in initialize()

7.1.3.2 value()

```
bool sensors::button_t::value (
            void  )
```

Returns true if the button is pressed

7.1.4 Member Data Documentation

7.1.4.1 inverted

```
bool sensors::button_t::inverted
```

Whether or not the button's value should be inverted

7.1.4.2 port

```
unsigned char sensors::button_t::port
```

the port that the button is plugged in to

The documentation for this struct was generated from the following files:

- include/sensors.hpp
- src/sensors.cpp

7.2 commit.Commit Class Reference

Inheritance diagram for commit.Commit:

Public Member Functions

- def **__init__** (self, date, commitkey, author, description, filesModified, filesAdded, filesDeleted)

Public Attributes

- **date**
- **commitkey**
- **author**
- **description**
- **filesModified**
- **filesAdded**
- **filesDeleted**

Static Public Attributes

- string **date** = ""
- string **commitkey** = ""
- string **author** = ""
- string **description** = ""
- list **filesModified** = [ ]
- list **filesAdded** = [ ]
- list **filesDeleted** = [ ]

The documentation for this class was generated from the following file:

- docs/commit.py

7.3 gyro::drive Class Reference

Public Attributes

- sensors::gyro_t * **gyro**
- int **heading**

The documentation for this class was generated from the following file:

- include/gyro.hpp

7.4 sensors::gyro_t Class Reference

```
#include <sensors.hpp>
```

Public Member Functions

- void reset (void)
- long value (void)
- void init (void)
- gyro_t (unsigned char _port, unsigned int _calibration)

Public Attributes

- Gyro gyro
- unsigned char port
- long zero
- float request

7.4.1 Detailed Description

Class for gyro objects

7.4.2 Constructor & Destructor Documentation

7.4.2.1 gyro_t()

```
sensors::gyro_t::gyro_t (
            unsigned char _port,
            unsigned int _calibration )
```

Class constructor, but it must not be forgotten to call init()

7.4.3 Member Function Documentation

7.4.3.1 init()

```
void sensors::gyro_t::init (
            void  )
```

Initialization funtion for the gyro, call in initialize()

7.4.3.2 reset()

```
void sensors::gyro_t::reset (
            void  )
```

Resets the value to 0

7.4.3.3 value()

```
long sensors::gyro_t::value (
            void  )
```

Returns the current value of the gyro, relative to the zero

7.4.4 Member Data Documentation

7.4.4.1 gyro

```
Gyro sensors::gyro_t::gyro
```

The gyro struct used in funtions

7.4.4.2 port

```
unsigned char sensors::gyro_t::port
```

The port the gyro is plugged into

7.4.4.3 request

```
float sensors::gyro_t::request
```

The pid requested value of the gyro

7.4.4.4 zero

```
long sensors::gyro_t::zero
```

The relative zero of the gyro, such that you can add it to the returned value to obtain an absolute value

The documentation for this class was generated from the following files:

- include/sensors.hpp
- src/sensors.cpp

7.5 motor_t Struct Reference

```
#include <motors.hpp>
```

Public Member Functions

- void set (int power)

Public Attributes

- unsigned char port
- char inverted
- int power
- float scale
- float **slewRate**
- unsigned long tlast

7.5.1 Detailed Description

Class for motor objects

7.5.2 Member Function Documentation

7.5.2.1 set()

```
void motor_t::set (
            int power )
```

Set the motor to the specified power

7.5.3 Member Data Documentation

7.5.3.1 inverted

```
char motor_t::inverted
```

The invered status of the motor, should be 1 or -1

7.5.3.2 port

```
unsigned char motor_t::port
```

Port the motor is pluggin in to

7.5.3.3 power

```
int motor_t::power
```

The requested power value of the motor

7.5.3.4 scale

```
float motor_t::scale
```

A multiplier for setting the motor values

7.5.3.5 tlast

```
unsigned long motor_t::tlast
```

The last update time of the motor. Is managed by the slew task, so it shouldn't need to be changed

The documentation for this struct was generated from the following files:

- include/motors.hpp
- src/motors.cpp

7.6 sensors::pot_t Struct Reference

```
#include <sensors.hpp>
```

Public Member Functions

- void reset (void)
- long value (void)
- void init (void)
- pot_t (unsigned char _port, bool _inverted)

Public Attributes

- unsigned char port
- long zero
- bool inverted
- float request

7.6.1 Detailed Description

Class for potentiometers

7.6.2 Constructor & Destructor Documentation

7.6.2.1 pot_t()

```
sensors::pot_t::pot_t (
            unsigned char _port,
            bool _inverted )
```

The class constructor for a potentiometer, also be sure to init()

7.6.3 Member Function Documentation

7.6.3.1 init()

```
void sensors::pot_t::init (
            void  )
```

The initialization funtion for the potentiometer, which must be called in initialize()

7.6.3.2 reset()

```
void sensors::pot_t::reset (
            void  )
```

Resets the value to 0

7.6.3.3 value()

```
long sensors::pot_t::value (
            void  )
```

Returns the relative value of the potentiometer

7.6.4 Member Data Documentation

7.6.4.1 inverted

```
bool sensors::pot_t::inverted
```

Whether or not the potentioeter's value should be inverted

7.6.4.2 port

```
unsigned char sensors::pot_t::port
```

The port that the pot is plugged in to

7.6.4.3 request

```
float sensors::pot_t::request
```

The pid requested value of the pot

7.6.4.4 zero

```
long sensors::pot_t::zero
```

The relative zero, that can be added to the returned value() to find the absolute value

The documentation for this struct was generated from the following files:

- include/sensors.hpp
- src/sensors.cpp

7.7 sensors::quad_t Struct Reference

```
#include <sensors.hpp>
```

Public Member Functions

- void reset (void)
- long value (void)
- void init (void)
- quad_t (unsigned char port1, unsigned char port2, bool _inverted)

Public Attributes

- Encoder enc
- unsigned char ports [2]
- long zero
- bool inverted
- float request

7.7.1 Detailed Description

A 2-wire quadrature encoder

7.7.2 Constructor & Destructor Documentation

7.7.2.1 quad_t()

```
sensors::quad_t::quad_t (
            unsigned char port1,
            unsigned char port2,
            bool _inverted )
```

Constructs the encoder object. Make sure init is also called

7.7.3 Member Function Documentation

7.7.3.1 init()

```
void sensors::quad_t::init (
            void  )
```

The initialization function for the encoder. Call in initialize()

7.7.3.2 reset()

```
void sensors::quad_t::reset (
            void  )
```

Reset the value to zero

7.7.3.3 value()

```
long sensors::quad_t::value (
            void  )
```

Returns the relative value of the encoder. If added to the encoder's zero, produces an absolute value of the encoder

7.7.4 Member Data Documentation

7.7.4.1 enc

```
Encoder sensors::quad_t::enc
```

The encoder struct used by other member functions

7.7.4.2 inverted

```
bool sensors::quad_t::inverted
```

Whether or not the encoder is inverted

7.7.4.3 ports

```
unsigned char sensors::quad_t::ports[2]
```

The ports the encoder is connected to, in order of top, then bottom, when the removable cover is facing up

7.7.4.4 request

```
float sensors::quad_t::request
```

The pid requested value of the encoder

7.7.4.5 zero

```
long sensors::quad_t::zero
```

The relative zero from which the encoder's value will be returned. Can be added to returned value to produce a true value for the encoder

The documentation for this struct was generated from the following files:

- include/sensors.hpp
- src/sensors.cpp

7.8 drive::side_t Struct Reference

```
#include <drive.hpp>
```

Public Member Functions

- void set (int power)

Public Attributes

- motor_t topM
- motor_t midM
- motor_t lowM
- sensors::quad_t * sensor

7.8.1 Detailed Description

Class for a side of the drive

7.8.2 Member Function Documentation

7.8.2.1 set()

```
void drive::side_t::set (
            int power )
```

Sets all motors on the side to the given power

7.8.3 Member Data Documentation

7.8.3.1 lowM

```
motor_t drive::side_t::lowM
```

Bottom motor on the side

7.8.3.2 midM

```
motor_t drive::side_t::midM
```

Middle motor on the side

7.8.3.3 sensor

```
sensors::quad_t* drive::side_t::sensor
```

A pointer to the sensor on the side

7.8.3.4 topM

```
motor_t drive::side_t::topM
```

Top motor on the the side

The documentation for this struct was generated from the following files:

- include/drive.hpp
- src/drive.cpp

7.9 lift::side_t Struct Reference

`#include <lift.hpp>`

Public Member Functions

- void set (int power)

Public Attributes

- motor_t topM
- motor_t midM
- motor_t lowM
- sensors::pot_t * sensor

7.9.1 Detailed Description

Class for a side of the drive

7.9.2 Member Function Documentation

7.9.2.1 set()

```
void lift::side_t::set (
            int power )
```

Sets all motors on the side to the given power

7.9.3 Member Data Documentation

7.9.3.1 lowM

`motor_t lift::side_t::lowM`

Bottom motor on the side

7.9.3.2 midM

`motor_t lift::side_t::midM`

Middle motor on the side

7.9.3.3 sensor

```
sensors::pot_t* lift::side_t::sensor
```

A pointer to the sensor on the side

7.9.3.4 topM

```
motor_t lift::side_t::topM
```

Top motor on the the side

The documentation for this struct was generated from the following files:

- include/lift.hpp
- src/lift.cpp

7.10 sensors::sonic_t Struct Reference

```
#include <sensors.hpp>
```

Public Member Functions

- long value (void)
- void init (void)
- sonic_t (unsigned char port1, unsigned char port2)

Public Attributes

- Ultrasonic sonic
- unsigned char ports [2]

7.10.1 Detailed Description

Class for ultrasonic sensors

7.10.2 Constructor & Destructor Documentation

7.10.2.1 sonic_t()

```
sensors::sonic_t::sonic_t (
            unsigned char port1,
            unsigned char port2 )
```

Class constructor, but init() must also be called

7.10.3 Member Function Documentation

7.10.3.1 init()

```
void sensors::sonic_t::init (
            void  )
```

Initializes the sensor. Call in initialize()

7.10.3.2 value()

```
long sensors::sonic_t::value (
            void  )
```

The value of the ultrasonic sensor

7.10.4 Member Data Documentation

7.10.4.1 ports

```
unsigned char sensors::sonic_t::ports[2]
```

The two ports the sensor is plugged in to, in order of the echo (aka orange) cable, then the ping (aka yellow) cable

7.10.4.2 sonic

```
Ultrasonic sensors::sonic_t::sonic
```

The Ultrasonic struct that is referenced in member funtions

The documentation for this struct was generated from the following files:

- include/sensors.hpp
- src/sensors.cpp

Chapter 8

File Documentation

8.1 include/API.h File Reference

Provides the high-level user functionality intended for use by typical VEX Cortex programmers.

```
#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
```

Macros

- #define JOY_DOWN 1
- #define JOY_LEFT 2
- #define JOY_UP 4
- #define JOY_RIGHT 8
- #define ACCEL_X 5
- #define ACCEL_Y 6
- #define BOARD_NR_ADC_PINS 8
- #define BOARD_NR_GPIO_PINS 27
- #define HIGH 1
- #define LOW 0
- #define INPUT 0x0A
- #define INPUT_ANALOG 0x00
- #define INPUT_FLOATING 0x04
- #define OUTPUT 0x01
- #define OUTPUT_OD 0x05
- #define INTERRUPT_EDGE_RISING 1
- #define INTERRUPT_EDGE_FALLING 2
- #define INTERRUPT_EDGE_BOTH 3
- #define IME_ADDR_MAX 0x1F
- #define SERIAL_DATABITS_8 0x0000
- #define SERIAL_DATABITS_9 0x1000
- #define SERIAL_STOPBITS_1 0x0000
- #define SERIAL_STOPBITS_2 0x2000
- #define SERIAL_PARITY_NONE 0x0000

- #define SERIAL_PARITY_EVEN 0x0400
- #define SERIAL_PARITY_ODD 0x0600
- #define SERIAL_8N1 0x0000
- #define stdout ((PROS_FILE*)3)
- #define stdin ((PROS_FILE*)3)
- #define uart1 ((PROS_FILE*)1)
- #define uart2 ((PROS_FILE*)2)
- #define EOF ((int)-1)
- #define SEEK_SET 0
- #define SEEK_CUR 1
- #define SEEK_END 2
- #define LCD_BTN_LEFT 1
- #define LCD_BTN_CENTER 2
- #define LCD_BTN_RIGHT 4
- #define TASK_MAX 16
- #define TASK_MAX_PRIORITIES 6
- #define TASK_PRIORITY_LOWEST 0
- #define TASK_PRIORITY_DEFAULT 2
- #define TASK_PRIORITY_HIGHEST (TASK_MAX_PRIORITIES - 1)
- #define TASK_DEFAULT_STACK_SIZE 512
- #define TASK_MINIMAL_STACK_SIZE 64
- #define TASK_DEAD 0
- #define TASK_RUNNING 1
- #define TASK_RUNNABLE 2
- #define TASK_SLEEPING 3
- #define TASK_SUSPENDED 4

Typedefs

- typedef void(* InterruptHandler) (unsigned char pin)
- typedef void * Gyro
- typedef void * Encoder
- typedef void * Ultrasonic
- typedef int PROS_FILE
- typedef void * TaskHandle
- typedef void * Mutex
- typedef void * Semaphore
- typedef void(* TaskCode) (void *)

Functions

- bool isAutonomous ()
- bool isEnabled ()
- bool isJoystickConnected (unsigned char joystick)
- bool isOnline ()
- int joystickGetAnalog (unsigned char joystick, unsigned char axis)
- bool joystickGetDigital (unsigned char joystick, unsigned char buttonGroup, unsigned char button)
- unsigned int powerLevelBackup ()
- unsigned int powerLevelMain ()
- void setTeamName (const char *name)
- int analogCalibrate (unsigned char channel)
- int analogRead (unsigned char channel)
- int analogReadCalibrated (unsigned char channel)

- int analogReadCalibratedHR (unsigned char channel)
- bool digitalRead (unsigned char pin)
- void digitalWrite (unsigned char pin, bool value)
- void pinMode (unsigned char pin, unsigned char mode)
- void ioClearInterrupt (unsigned char pin)
- void ioSetInterrupt (unsigned char pin, unsigned char edges, InterruptHandler handler)
- int motorGet (unsigned char channel)
- void motorSet (unsigned char channel, int speed)
- void motorStop (unsigned char channel)
- void motorStopAll ()
- void speakerInit ()
- void speakerPlayArray (const char **songs)
- void speakerPlayRtttl (const char *song)
- void speakerShutdown ()
- unsigned int imeInitializeAll ()
- bool imeGet (unsigned char address, int *value)
- bool imeGetVelocity (unsigned char address, int *value)
- bool imeReset (unsigned char address)
- void imeShutdown ()
- int gyroGet (Gyro gyro)
- Gyro gyroInit (unsigned char port, unsigned short multiplier)
- void gyroReset (Gyro gyro)
- void gyroShutdown (Gyro gyro)
- int encoderGet (Encoder enc)
- Encoder encoderInit (unsigned char portTop, unsigned char portBottom, bool reverse)
- void encoderReset (Encoder enc)
- void encoderShutdown (Encoder enc)
- int ultrasonicGet (Ultrasonic ult)
- Ultrasonic ultrasonicInit (unsigned char portEcho, unsigned char portPing)
- void ultrasonicShutdown (Ultrasonic ult)
- bool i2cRead (uint8_t addr, uint8_t *data, uint16_t count)
- bool i2cReadRegister (uint8_t addr, uint8_t reg, uint8_t *value, uint16_t count)
- bool i2cWrite (uint8_t addr, uint8_t *data, uint16_t count)
- bool i2cWriteRegister (uint8_t addr, uint8_t reg, uint16_t value)
- void usartInit (PROS_FILE *usart, unsigned int baud, unsigned int flags)
- void usartShutdown (PROS_FILE *usart)
- void fclose (PROS_FILE *stream)
- int fcount (PROS_FILE *stream)
- int fdelete (const char *file)
- int feof (PROS_FILE *stream)
- int fflush (PROS_FILE *stream)
- int fgetc (PROS_FILE *stream)
- char * fgets (char *str, int num, PROS_FILE *stream)
- PROS_FILE * fopen (const char *file, const char *mode)
- void fprint (const char *string, PROS_FILE *stream)
- int fputc (int value, PROS_FILE *stream)
- int fputs (const char *string, PROS_FILE *stream)
- size_t fread (void *ptr, size_t size, size_t count, PROS_FILE *stream)
- int fseek (PROS_FILE *stream, long int offset, int origin)
- long int ftell (PROS_FILE *stream)
- size_t fwrite (const void *ptr, size_t size, size_t count, PROS_FILE *stream)
- int getchar ()
- void print (const char *string)
- int putchar (int value)
- int puts (const char *string)

- int fprintf (PROS_FILE *stream, const char *formatString,...)
- int printf (const char *formatString,...)
- int snprintf (char *buffer, size_t limit, const char *formatString,...)
- int sprintf (char *buffer, const char *formatString,...)
- void lcdClear (PROS_FILE *lcdPort)
- void lcdInit (PROS_FILE *lcdPort)
- void __attribute__ ((format(printf, 3, 4))) lcdPrint(PROS_FILE *lcdPort
- void unsigned char const char unsigned int lcdReadButtons (PROS_FILE *lcdPort)
- void lcdSetBacklight (PROS_FILE *lcdPort, bool backlight)
- void lcdSetText (PROS_FILE *lcdPort, unsigned char line, const char *buffer)
- void lcdShutdown (PROS_FILE *lcdPort)
- TaskHandle taskCreate (TaskCode taskCode, const unsigned int stackDepth, void *parameters, const unsigned int priority)
- void taskDelay (const unsigned long msToDelay)
- void taskDelayUntil (unsigned long *previousWakeTime, const unsigned long cycleTime)
- void taskDelete (TaskHandle taskToDelete)
- unsigned int taskGetCount ()
- unsigned int taskGetState (TaskHandle task)
- unsigned int taskPriorityGet (const TaskHandle task)
- void taskPrioritySet (TaskHandle task, const unsigned int newPriority)
- void taskResume (TaskHandle taskToResume)
- TaskHandle taskRunLoop (void(*fn)(void), const unsigned long increment)
- void taskSuspend (TaskHandle taskToSuspend)
- Semaphore semaphoreCreate ()
- bool semaphoreGive (Semaphore semaphore)
- bool semaphoreTake (Semaphore semaphore, const unsigned long blockTime)
- void semaphoreDelete (Semaphore semaphore)
- Mutex mutexCreate ()
- bool mutexGive (Mutex mutex)
- bool mutexTake (Mutex mutex, const unsigned long blockTime)
- void mutexDelete (Mutex mutex)
- void delay (const unsigned long time)
- void delayMicroseconds (const unsigned long us)
- unsigned long micros ()
- unsigned long millis ()
- void wait (const unsigned long time)
- void waitUntil (unsigned long *previousWakeTime, const unsigned long time)
- void iwdgEnable ()

Variables

- void unsigned char **line**
- void unsigned char const char * **formatString**

8.1.1 Detailed Description

Provides the high-level user functionality intended for use by typical VEX Cortex programmers.

This file should be included for you in the predefined stubs in each new VEX Cortex PROS project through the inclusion of "main.h". In any new C source file, it is advisable to include main.h instead of referencing API.h by name, to better handle any nomenclature changes to this file or its contents.

PROS contains FreeRTOS (http://www.freertos.org) whose source code may be obtained from http://sourceforge.net/projects/freertos/files/ or on request.

8.1.2 Macro Definition Documentation

8.1.2.1 ACCEL_X

```
#define ACCEL_X 5
```

Analog axis for the X acceleration from the VEX Joystick.

8.1.2.2 ACCEL_Y

```
#define ACCEL_Y 6
```

Analog axis for the Y acceleration from the VEX Joystick.

8.1.2.3 BOARD_NR_ADC_PINS

```
#define BOARD_NR_ADC_PINS 8
```

There are 8 available analog I/O on the Cortex.

8.1.2.4 BOARD_NR_GPIO_PINS

```
#define BOARD_NR_GPIO_PINS 27
```

There are 27 available I/O on the Cortex that can be used for digital communication.

This excludes the crystal ports but includes the Communications, Speaker, and Analog ports.

The motor ports are not on the Cortex and are thus excluded from this count. Pin 0 is the Speaker port, pins 1-12 are the standard Digital I/O, 13-20 are the Analog I/O, 21+22 are UART1, 23+24 are UART2, and 25+26 are the I2C port.

8.1.2.5 EOF

```
#define EOF ((int)-1)
```

EOF is a value evaluating to -1.

8.1.2.6 HIGH

```
#define HIGH 1
```

Used for digitalWrite() to specify a logic HIGH state to output.

In reality, using any non-zero expression or "true" will work to set a pin to HIGH.

8.1.2.7 IME_ADDR_MAX

```
#define IME_ADDR_MAX 0x1F
```

IME addresses end at 0x1F. Actually using more than 10 (address 0x1A) encoders will cause unreliable communications.

8.1.2.8 INPUT

```
#define INPUT 0x0A
```

pinMode() state for digital input, with pullup.

This is the default state for the 12 Digital pins. The pullup causes the input to read as "HIGH" when unplugged, but is fairly weak and can safely be driven by most sources. Many VEX digital sensors rely on this behavior and cannot be used with INPUT_FLOATING.

8.1.2.9 INPUT_ANALOG

```
#define INPUT_ANALOG 0x00
```

pinMode() state for analog inputs.

This is the default state for the 8 Analog pins and the Speaker port. This only works on pins with analog input capabilities; use anywhere else results in undefined behavior.

8.1.2.10 INPUT_FLOATING

```
#define INPUT_FLOATING 0x04
```

pinMode() state for digital input, without pullup.

Beware of power consumption, as digital inputs left "floating" may switch back and forth and cause spurious interrupts.

8.1.2.11 INTERRUPT_EDGE_BOTH

```
#define INTERRUPT_EDGE_BOTH 3
```

When used in ioSetInterrupt(), triggers an interrupt on both rising and falling edges (LOW to HIGH or HIGH to LOW).

8.1.2.12 INTERRUPT_EDGE_FALLING

```
#define INTERRUPT_EDGE_FALLING 2
```

When used in ioSetInterrupt(), triggers an interrupt on falling edges (HIGH to LOW).

8.1.2.13 INTERRUPT_EDGE_RISING

```
#define INTERRUPT_EDGE_RISING 1
```

When used in ioSetInterrupt(), triggers an interrupt on rising edges (LOW to HIGH).

8.1.2.14 JOY_DOWN

```
#define JOY_DOWN 1
```

DOWN button (valid on channels 5, 6, 7, 8)

8.1.2.15 JOY_LEFT

```
#define JOY_LEFT 2
```

LEFT button (valid on channels 7, 8)

8.1.2.16 JOY_RIGHT

```
#define JOY_RIGHT 8
```

RIGHT button (valid on channels 7, 8)

8.1.2.17 JOY_UP

```
#define JOY_UP 4
```

UP button (valid on channels 5, 6, 7, 8)

8.1.2.18 LCD_BTN_CENTER

```
#define LCD_BTN_CENTER 2
```

CENTER button on LCD for use with lcdReadButtons()

8.1.2.19 LCD_BTN_LEFT

```
#define LCD_BTN_LEFT 1
```

LEFT button on LCD for use with lcdReadButtons()

8.1.2.20 LCD_BTN_RIGHT

```
#define LCD_BTN_RIGHT 4
```

RIGHT button on LCD for use with lcdReadButtons()

8.1.2.21 LOW

```
#define LOW 0
```

Used for digitalWrite() to specify a logic LOW state to output.

In reality, using a zero expression or "false" will work to set a pin to LOW.

8.1.2.22 OUTPUT

```
#define OUTPUT 0x01
```

pinMode() state for digital output, push-pull.

This is the mode which should be used to output a digital HIGH or LOW value from the Cortex. This mode is useful for pneumatic solenoid valves and VEX LEDs.

8.1.2.23 OUTPUT_OD

```
#define OUTPUT_OD 0x05
```

pinMode() state for open-drain outputs.

This is useful in a few cases for external electronics and should not be used for the VEX solenoid or LEDs.

8.1.2.24 SEEK_CUR

```
#define SEEK_CUR 1
```

SEEK_CUR is used in fseek() to denote an relative position in bytes from the current file location.

8.1.2.25 SEEK_END

```
#define SEEK_END 2
```

SEEK_END is used in fseek() to denote an absolute position in bytes from the end of the file. The offset will most likely be negative in this case.

8.1.2.26 SEEK_SET

```
#define SEEK_SET 0
```

SEEK_SET is used in fseek() to denote an absolute position in bytes from the start of the file.

8.1.2.27 SERIAL_8N1

```
#define SERIAL_8N1 0x0000
```

Specifies the default serial settings when used in usartInit()

8.1.2.28 SERIAL_DATABITS_8

```
#define SERIAL_DATABITS_8 0x0000
```

Bit mask for usartInit() for 8 data bits (typical)

8.1.2.29 SERIAL_DATABITS_9

```
#define SERIAL_DATABITS_9 0x1000
```

Bit mask for usartInit() for 9 data bits

8.1.2.30 SERIAL_PARITY_EVEN

```
#define SERIAL_PARITY_EVEN 0x0400
```

Bit mask for usartInit() for Even parity

8.1.2.31 SERIAL_PARITY_NONE

```
#define SERIAL_PARITY_NONE 0x0000
```

Bit mask for usartInit() for No parity (typical)

8.1.2.32 SERIAL_PARITY_ODD

```
#define SERIAL_PARITY_ODD 0x0600
```

Bit mask for usartInit() for Odd parity

8.1.2.33 SERIAL_STOPBITS_1

```
#define SERIAL_STOPBITS_1 0x0000
```

Bit mask for usartInit() for 1 stop bit (typical)

8.1.2.34 SERIAL_STOPBITS_2

```
#define SERIAL_STOPBITS_2 0x2000
```

Bit mask for usartInit() for 2 stop bits

8.1.2.35 stdin

```
#define stdin ((PROS_FILE*)3)
```

The standard input stream uses the PC debug terminal.

8.1.2.36 stdout

```
#define stdout ((PROS_FILE*)3)
```

The standard output stream uses the PC debug terminal.

8.1.2.37 TASK_DEAD

```
#define TASK_DEAD 0
```

Constant returned from taskGetState() when the task is dead or nonexistant.

8.1.2.38 TASK_DEFAULT_STACK_SIZE

```
#define TASK_DEFAULT_STACK_SIZE 512
```

The recommended stack size for a new task that does an average amount of work. This stack size is used for default tasks such as autonomous().

This is probably OK for 4-5 levels of function calls and the use of printf() with several arguments. Tasks requiring deep recursion or large local buffers will need a bigger stack.

8.1.2.39 TASK_MAX

```
#define TASK_MAX 16
```

Only this many tasks can exist at once. Attempts to create further tasks will not succeed until tasks end or are destroyed, AND the idle task cleans them up.

Changing this value will not change the limit without a kernel recompile. The idle task and VEX daemon task count against the limit. The user autonomous() or teleop() also counts against the limit, so 12 tasks usually remain for other uses.

8.1.2.40 TASK_MAX_PRIORITIES

```
#define TASK_MAX_PRIORITIES 6
```

The maximum number of available task priorities, which run from 0 to 5.

Changing this value will not change the priority count without a kernel recompile.

8.1.2.41 TASK_MINIMAL_STACK_SIZE

```
#define TASK_MINIMAL_STACK_SIZE 64
```

The minimum stack depth for a task. Scheduler state is stored on the stack, so even if the task never uses the stack, at least this much space must be allocated.

Function calls and other seemingly innocent constructs may place information on the stack. Err on the side of a larger stack when possible.

8.1.2.42 TASK_PRIORITY_DEFAULT

```
#define TASK_PRIORITY_DEFAULT 2
```

The default task priority, which should be used for most tasks.

Default tasks such as autonomous() inherit this priority.

8.1.2.43 TASK_PRIORITY_HIGHEST

```
#define TASK_PRIORITY_HIGHEST (TASK_MAX_PRIORITIES - 1)
```

The highest priority that can be assigned to a task. Unlike the lowest priority, this priority can be safely used without hampering interrupts. Beware of deadlock.

8.1.2.44 TASK_PRIORITY_LOWEST

```
#define TASK_PRIORITY_LOWEST 0
```

The lowest priority that can be assigned to a task, which puts it on a level with the idle task. This may cause severe performance problems and is generally not recommended.

8.1.2.45 TASK_RUNNABLE

```
#define TASK_RUNNABLE 2
```

Constant returned from taskGetState() when the task is exists and is available to run, but not currently running.

8.1.2.46 TASK_RUNNING

```
#define TASK_RUNNING 1
```

Constant returned from taskGetState() when the task is actively executing.

8.1.2.47 TASK_SLEEPING

```
#define TASK_SLEEPING 3
```

Constant returned from taskGetState() when the task is delayed or blocked waiting for a semaphore, mutex, or I/O operation.

8.1.2.48 TASK_SUSPENDED

```
#define TASK_SUSPENDED 4
```

Constant returned from taskGetState() when the task is suspended using taskSuspend().

8.1.2.49 uart1

```
#define uart1 ((PROS_FILE*)1)
```

UART 1 on the Cortex; must be opened first using usartInit().

8.1.2.50 uart2

```
#define uart2 ((PROS_FILE*)2)
```

UART 2 on the Cortex; must be opened first using usartInit().

8.1.3 Typedef Documentation

8.1.3.1 Encoder

```
typedef void* Encoder
```

Reference type for an initialized encoder.

Encoder information is stored as an opaque pointer to a structure in memory; as this is a pointer type, it can be safely passed or stored by value.

8.1.3.2 Gyro

```
typedef void* Gyro
```

Reference type for an initialized gyro.

Gyro information is stored as an opaque pointer to a structure in memory; as this is a pointer type, it can be safely passed or stored by value.

8.1.3.3 InterruptHandler

```
typedef void(* InterruptHandler) (unsigned char pin)
```

Type definition for interrupt handlers. Such functions must accept one argument indicating the pin which changed.

8.1.3.4 Mutex

```
typedef void* Mutex
```

Type by which mutexes are referenced.

As this is a pointer type, it can be safely passed or stored by value.

8.1.3.5 PROS_FILE

```
typedef int PROS_FILE
```

FILE is an integer referring to a stream for the standard I/O functions.

PROS_FILE * is the standard library method of referring to a file pointer, even though there is actually nothing there.

8.1.3.6 Semaphore

```
typedef void* Semaphore
```

Type by which semaphores are referenced.

As this is a pointer type, it can be safely passed or stored by value.

8.1.3.7 TaskCode

```
typedef void(* TaskCode) (void *)
```

Type for defining task functions. Task functions must accept one parameter of type "void *"; they need not use it.

For example:

void MyTask(void *ignore) { while (1); }

8.1.3.8 TaskHandle

```
typedef void* TaskHandle
```

Type by which tasks are referenced.

As this is a pointer type, it can be safely passed or stored by value.

8.1.3.9 Ultrasonic

```
typedef void* Ultrasonic
```

Reference type for an initialized ultrasonic sensor.

Ultrasonic information is stored as an opaque pointer to a structure in memory; as this is a pointer type, it can be safely passed or stored by value.

8.1.4 Function Documentation

8.1.4.1 __attribute__()

```
void __attribute__ (
             (format(printf, 3, 4))  )
```

Prints the formatted string to the attached LCD.

The output string will be truncated as necessary to fit on the LCD screen, 16 characters wide. It is probably better to generate the string in a local buffer and use lcdSetText() but this method is provided for convenience.

Parameters

lcdPort	the LCD to write, either uart1 or uart2
line	the LCD line to write, either 1 or 2
formatString	the format string as specified in fprintf()

8.1.4.2 analogCalibrate()

```
int analogCalibrate (
            unsigned char channel )
```

Calibrates the analog sensor on the specified channel.

This method assumes that the true sensor value is not actively changing at this time and computes an average from approximately 500 samples, 1 ms apart, for a 0.5 s period of calibration. The average value thus calculated is returned and stored for later calls to the analogReadCalibrated() and analogReadCalibratedHR() functions. These functions will return the difference between this value and the current sensor value when called.

Do not use this function in initializeIO(), or when the sensor value might be unstable (gyro rotation, accelerometer movement).

This function may not work properly if the VEX Cortex is tethered to a PC using the orange USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Battery provides power to sensors.

Parameters

channel	the channel to calibrate from 1-8

Returns

the average sensor value computed by this function

8.1.4.3 analogRead()

```
int analogRead (
            unsigned char channel )
```

Reads an analog input channel and returns the 12-bit value.

The value returned is undefined if the analog pin has been switched to a different mode. This function is Wiring-compatible with the exception of the larger output range. The meaning of the returned value varies depending on the sensor attached.

This function may not work properly if the VEX Cortex is tethered to a PC using the orange USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Battery provides power to sensors.

Parameters

channel	the channel to read from 1-8

Returns

the analog sensor value, where a value of 0 reflects an input voltage of nearly 0 V and a value of 4095 reflects an input voltage of nearly 5 V

8.1.4.4 analogReadCalibrated()

```
int analogReadCalibrated (
            unsigned char channel )
```

Reads the calibrated value of an analog input channel.

The analogCalibrate() function must be run first on that channel. This function is inappropriate for sensor values intended for integration, as round-off error can accumulate causing drift over time. Use analogReadCalibratedHR() instead.

This function may not work properly if the VEX Cortex is tethered to a PC using the orange USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Battery provides power to sensors.

Parameters

channel	the channel to read from 1-8

Returns

the difference of the sensor value from its calibrated default from -4095 to 4095

8.1.4.5 analogReadCalibratedHR()

```
int analogReadCalibratedHR (
            unsigned char channel )
```

Reads the calibrated value of an analog input channel 1-8 with enhanced precision.

The analogCalibrate() function must be run first. This is intended for integrated sensor values such as gyros and accelerometers to reduce drift due to round-off, and should not be used on a sensor such as a line tracker or potentiometer.

The value returned actually has 16 bits of "precision", even though the ADC only reads 12 bits, so that errors induced by the average value being between two values come out in the wash when integrated over time. Think of the value as the true value times 16.

This function may not work properly if the VEX Cortex is tethered to a PC using the orange USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Battery provides power to sensors.

Parameters

channel	the channel to read from 1-8

Returns

the difference of the sensor value from its calibrated default from -16384 to 16384

8.1.4.6 delay()

```
void delay (
            const unsigned long time )
```

Wiring-compatible alias of taskDelay().

Parameters

time	the duration of the delay in milliseconds (1 000 milliseconds per second)

8.1.4.7 delayMicroseconds()

```
void delayMicroseconds (
            const unsigned long us )
```

Wait for approximately the given number of microseconds.

The method used for delaying this length of time may vary depending on the argument. The current task will always be delayed by at least the specified period, but possibly much more depending on CPU load. In general, this function is less reliable than delay(). Using this function in a loop may hog processing time from other tasks.

Parameters

us	the duration of the delay in microseconds (1 000 000 microseconds per second)

8.1.4.8 digitalRead()

```
bool digitalRead (
            unsigned char pin )
```

Gets the digital value (1 or 0) of a pin configured as a digital input.

If the pin is configured as some other mode, the digital value which reflects the current state of the pin is returned, which may or may not differ from the currently set value. The return value is undefined for pins configured as Analog inputs, or for ports in use by a Communications interface. This function is Wiring-compatible.

This function may not work properly if the VEX Cortex is tethered to a PC using the orange USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Battery provides power to sensors.

Parameters

pin	the pin to read from 1-26

Returns

true if the pin is HIGH, or false if it is LOW

8.1.4.9 digitalWrite()

```
void digitalWrite (
            unsigned char pin,
            bool value )
```

Sets the digital value (1 or 0) of a pin configured as a digital output.

If the pin is configured as some other mode, behavior is undefined. This function is Wiring-compatible.

Parameters

pin	the pin to write from 1-26
value	an expression evaluating to "true" or "false" to set the output to HIGH or LOW respectively, or the constants HIGH or LOW themselves

8.1.4.10 encoderGet()

```
int encoderGet (
            Encoder enc )
```

Gets the number of ticks recorded by the encoder.

There are 360 ticks in one revolution.

Parameters

enc	the Encoder object from encoderInit() to read

Returns

the signed and cumulative number of counts since the last start or reset

8.1.4.11 encoderInit()

```
Encoder encoderInit (
            unsigned char portTop,
            unsigned char portBottom,
            bool reverse )
```

Initializes and enables a quadrature encoder on two digital ports.

Neither the top port nor the bottom port can be digital port 10. NULL will be returned if either port is invalid or the encoder is already in use. Initializing an encoder implicitly resets its count.

Parameters

portTop	the "top" wire from the encoder sensor with the removable cover side UP
portBottom	the "bottom" wire from the encoder sensor
reverse	if "true", the sensor will count in the opposite direction

Returns

an Encoder object to be stored and used for later calls to encoder functions

8.1.4.12 encoderReset()

```
void encoderReset (
            Encoder enc )
```

Resets the encoder to zero.

It is safe to use this method while an encoder is enabled. It is not necessary to call this method before stopping or starting an encoder.

Parameters

enc	the Encoder object from encoderInit() to reset

8.1.4.13 encoderShutdown()

```
void encoderShutdown (
            Encoder enc )
```

Stops and disables the encoder.

Encoders use processing power, so disabling unused encoders increases code performance. The encoder's count will be retained.

Parameters

enc	the Encoder object from encoderInit() to stop

8.1.4.14 fclose()

```
void fclose (
             PROS_FILE * stream )
```

Closes the specified file descriptor. This function does not work on communication ports; use usartShutdown() instead.

Parameters

stream	the file descriptor to close from fopen()

8.1.4.15 fcount()

```
int fcount (
             PROS_FILE * stream )
```

Returns the number of characters that can be read without blocking (the number of characters available) from the specified stream. This only works for communication ports and files in Read mode; for files in Write mode, 0 is always returned.

This function may underestimate, but will not overestimate, the number of characters which meet this criterion.

Parameters

stream	the stream to read (stdin, uart1, uart2, or an open file in Read mode)

Returns

the number of characters which meet this criterion; if this number cannot be determined, returns 0

8.1.4.16 fdelete()

```
int fdelete (
             const char * file )
```

Delete the specified file if it exists and is not currently open.

The file will actually be erased from memory on the next re-boot. A physical power cycle is required to purge deleted files and free their allocated space for new files to be written. Deleted files are still considered inaccessible to fopen() in Read mode.

Parameters

file	the file name to erase

Returns

0 if the file was deleted, or 1 if the file could not be found

8.1.4.17 feof()

```
int feof (
            PROS_FILE * stream )
```

Checks to see if the specified stream is at its end. This only works for communication ports and files in Read mode; for files in Write mode, 1 is always returned.

Parameters

stream	the channel to check (stdin, uart1, uart2, or an open file in Read mode)

Returns

0 if the stream is not at EOF, or 1 otherwise.

8.1.4.18 fflush()

```
int fflush (
            PROS_FILE * stream )
```

Flushes the data on the specified file channel open in Write mode. This function has no effect on a communication port or a file in Read mode, as these streams are always flushed as quickly as possible by the kernel.

Successful completion of an fflush function on a file in Write mode cannot guarantee that the file is vaild until fclose() is used on that file descriptor.

Parameters

stream	the channel to flush (an open file in Write mode)

Returns

0 if the data was successfully flushed, EOF otherwise

8.1.4.19 fgetc()

```
int fgetc (
            PROS_FILE * stream )
```

Reads and returns one character from the specified stream, blocking until complete.

Do not use fgetc() on a VEX LCD port; deadlock may occur.

Parameters

stream	the stream to read (stdin, uart1, uart2, or an open file in Read mode)

Returns

the next character from 0 to 255, or -1 if no character can be read

8.1.4.20 fgets()

```
char* fgets (
            char * str,
            int num,
            PROS_FILE * stream )
```

Reads a string from the specified stream, storing the characters into the memory at str. Characters will be read until the specified limit is reached, a new line is found, or the end of file is reached.

If the stream is already at end of file (for files in Read mode), NULL will be returned; otherwise, at least one character will be read and stored into str.

Parameters

str	the location where the characters read will be stored
num	the maximum number of characters to store; at most (num - 1) characters will be read, with a null terminator ('\0') automatically appended
stream	the channel to read (stdin, uart1, uart2, or an open file in Read mode)

Returns

str, or NULL if zero characters could be read

8.1.4.21 fopen()

```
PROS_FILE* fopen (
            const char * file,
            const char * mode )
```

Opens the given file in the specified mode. The file name is truncated to eight characters. Only four files can be in use simultaneously in any given time, with at most one of those files in Write mode. This function does not work on communication ports; use usartInit() instead.

mode can be "r" or "w". Due to the nature of the VEX Cortex memory, the "r+", "w+", and "a" modes are not supported by the file system.

Opening a file that does not exist in Read mode will fail and return NULL, but opening a new file in Write mode will create it if there is space. Opening a file that already exists in Write mode will destroy the contents and create a new blank file if space is available.

There are important considerations when using of the file system on the VEX Cortex. Reading from files is safe, but writing to files should only be performed when robot actuators have been stopped. PROS will attempt to continue to handle events during file writes, but most user tasks cannot execute during file writing. Powering down the VEX Cortex mid-write may cause file system corruption.

Parameters

file	the file name
mode	the file mode

Returns

a file descriptor pointing to the new file, or NULL if the file could not be opened

8.1.4.22 fprint()

```
void fprint (
            const char * string,
            PROS_FILE * stream )
```

Prints the simple string to the specified stream.

This method is much, much faster than fprintf() and does not add a new line like fputs(). Do not use fprint() on a VEX LCD port. Use lcdSetText() instead.

Parameters

string	the string to write
stream	the stream to write (stdout, uart1, uart2, or an open file in Write mode)

8.1.4.23 fprintf()

```
int fprintf (
              PROS_FILE * stream,
              const char * formatString,
               ...  )
```

Prints the formatted string to the specified output stream.

The specifiers supported by this minimalistic printf() function are:

- `%d`: Signed integer in base 10 (int)
- `%u`: Unsigned integer in base 10 (unsigned int)
- `%x`, `%X`: Integer in base 16 (unsigned int, int)
- `%p`: Pointer (void *, int *, ...)
- `%c`: Character (char)
- `%s`: Null-terminated string (char *)
- `%%`: Single literal percent sign
- `%f`: Floating-point number

Specifiers can be modified with:

- `0`: Zero-pad, instead of space-pad
- `a.b`: Make the field at least "a" characters wide. If "b" is specified for "%f", changes the number of digits after the decimal point
- `-`: Left-align, instead of right-align
- `+`: Always display the sign character (displays a leading "+" for positive numbers)
- `l`: Ignored for compatibility

Invalid format specifiers, or mismatched parameters to specifiers, cause undefined behavior. Other characters are written out verbatim. Do not use fprintf() on a VEX LCD port. Use lcdPrint() instead.

Parameters

stream	the stream to write (stdout, uart1, or uart2)
formatString	the format string as specified above

Returns

the number of characters written

8.1.4.24 fputc()

```
int fputc (
              int value,
              PROS_FILE * stream )
```

Writes one character to the specified stream.

Do not use fputc() on a VEX LCD port. Use lcdSetText() instead.

Parameters

value	the character to write (a value of type "char" can be used)
stream	the stream to write (stdout, uart1, uart2, or an open file in Write mode)

Returns

the character written

8.1.4.25 fputs()

```
int fputs (
              const char * string,
              PROS_FILE * stream )
```

Behaves the same as the "fprint" function, and appends a trailing newline ("\n").

Do not use fputs() on a VEX LCD port. Use lcdSetText() instead.

Parameters

string	the string to write
stream	the stream to write (stdout, uart1, uart2, or an open file in Write mode)

Returns

the number of characters written, excluding the new line

8.1.4.26 fread()

```
size_t fread (
              void * ptr,
              size_t size,
              size_t count,
              PROS_FILE * stream )
```

Reads data from a stream into memory. Returns the number of bytes thus read.

If the memory at ptr cannot store (size * count) bytes, undefined behavior occurs.

Parameters

ptr	a pointer to where the data will be stored
size	the size of each data element to read in bytes
count	the number of data elements to read
stream	the stream to read (stdout, uart1, uart2, or an open file in Read mode)

Returns

the number of bytes successfully read

8.1.4.27 fseek()

```
int fseek (
            PROS_FILE * stream,
            long int offset,
            int origin )
```

Seeks within a file open in Read mode. This function will fail when used on a file in Write mode or on any communications port.

Parameters

stream	the stream to seek within
offset	the location within the stream to seek
origin	the reference location for offset: SEEK_CUR, SEEK_SET, or SEEK_END

Returns

0 if the seek was successful, or 1 otherwise

8.1.4.28 ftell()

```
long int ftell (
            PROS_FILE * stream )
```

Returns the current position of the stream. This function works on files in either Read or Write mode, but will fail on communications ports.

Parameters

stream	the stream to check

Returns

the offset of the stream, or -1 if the offset could not be determined

8.1.4.29 fwrite()

```
size_t fwrite (
            const void * ptr,
            size_t size,
            size_t count,
            PROS_FILE * stream )
```

Writes data from memory to a stream. Returns the number of bytes thus written.

If the memory at ptr is not as long as (size * count) bytes, undefined behavior occurs.

Parameters

ptr	a pointer to the data to write
size	the size of each data element to write in bytes
count	the number of data elements to write
stream	the stream to write (stdout, uart1, uart2, or an open file in Write mode)

Returns

the number of bytes successfully written

8.1.4.30 getchar()

```
int getchar ( )
```

Reads and returns one character from "stdin", which is the PC debug terminal.

Returns

the next character from 0 to 255, or -1 if no character can be read

8.1.4.31 gyroGet()

```
int gyroGet (
            Gyro gyro )
```

Gets the current gyro angle in degrees, rounded to the nearest degree.

There are 360 degrees in a circle.

Parameters

gyro	the Gyro object from gyroInit() to read

Returns

the signed and cumulative number of degrees rotated around the gyro's vertical axis since the last start or reset

8.1.4.32 gyroInit()

```
Gyro gyroInit (
              unsigned char port,
              unsigned short multiplier )
```

Initializes and enables a gyro on an analog port.

NULL will be returned if the port is invalid or the gyro is already in use. Initializing a gyro implicitly calibrates it and resets its count. Do not move the robot while the gyro is being calibrated. It is suggested to call this function in initialize() and to place the robot in its final position before powering it on.

The multiplier parameter can tune the gyro to adapt to specific sensors. The default value at this time is 196; higher values will increase the number of degrees reported for a fixed actual rotation, while lower values will decrease the number of degrees reported. If your robot is consistently turning too far, increase the multiplier, and if it is not turning far enough, decrease the multiplier.

Parameters

port	the analog port to use from 1-8
multiplier	an optional constant to tune the gyro readings; use 0 for the default value

Returns

a Gyro object to be stored and used for later calls to gyro functions

8.1.4.33 gyroReset()

```
void gyroReset (
              Gyro gyro )
```

Resets the gyro to zero.

It is safe to use this method while a gyro is enabled. It is not necessary to call this method before stopping or starting a gyro.

Parameters

gyro	the Gyro object from gyroInit() to reset

8.1.4.34 gyroShutdown()

```
void gyroShutdown (
            Gyro gyro )
```

Stops and disables the gyro.

Gyros use processing power, so disabling unused gyros increases code performance. The gyro's position will be retained.

Parameters

gyro	the Gyro object from gyroInit() to stop

8.1.4.35 i2cRead()

```
bool i2cRead (
            uint8_t addr,
            uint8_t * data,
            uint16_t count )
```

i2cRead - Reads the specified number of data bytes from the specified 7-bit I2C address. The bytes will be stored at the specified location. Returns true if successful or false if failed. If only some bytes could be read, false is still returned.

The I2C address should be right-aligned; the R/W bit is automatically supplied.

Since most I2C devices use an 8-bit register architecture, this method has limited usefulness. Consider i2cRead↩ Register instead for the vast majority of applications.

8.1.4.36 i2cReadRegister()

```
bool i2cReadRegister (
            uint8_t addr,
            uint8_t reg,
            uint8_t * value,
            uint16_t count )
```

i2cReadRegister - Reads the specified amount of data from the given register address on the specified 7-bit I2C address. Returns true if successful or false if failed. If only some bytes could be read, false is still returned.

The I2C address should be right-aligned; the R/W bit is automatically supplied.

Most I2C devices support an auto-increment address feature, so using this method to read more than one byte will usually read a block of sequential registers. Try to merge reads to separate registers into a larger read using this function whenever possible to improve code reliability, even if a few intermediate values need to be thrown away.

8.1.4.37 i2cWrite()

```
bool i2cWrite (
            uint8_t addr,
            uint8_t * data,
            uint16_t count )
```

i2cWrite - Writes the specified number of data bytes to the specified 7-bit I2C address. Returns true if successful or false if failed. If only smoe bytes could be written, false is still returned.

The I2C address should be right-aligned; the R/W bit is automatically supplied.

Since most I2C devices use an 8-bit register architecture, this method is mostly useful for setting the register position (most devices remember the last-used address) or writing a sequence of bytes to one register address using an auto-increment feature. In these cases, the first byte written from the data buffer should have the register address to use.

8.1.4.38 i2cWriteRegister()

```
bool i2cWriteRegister (
            uint8_t addr,
            uint8_t reg,
            uint16_t value )
```

i2cWriteRegister - Writes the specified data byte to a register address on the specified 7-bit I2C address. Returns true if successful or false if failed.

The I2C address should be right-aligned; the R/W bit is automatically supplied.

Only one byte can be written to each register address using this method. While useful for the vast majority of I2C operations, writing multiple bytes requires the i2cWrite method.

8.1.4.39 imeGet()

```
bool imeGet (
            unsigned char address,
            int * value )
```

Gets the current 32-bit count of the specified IME.

Much like the count for a quadrature encoder, the tick count is signed and cumulative. The value reflects total counts since the last reset. Different VEX Motor Encoders have a different number of counts per revolution:

- `240.448` for the 269 IME
- `627.2` for the 393 IME in high torque mode (factory default)
- `392` for the 393 IME in high speed mode

If the IME address is invalid, or the IME has not been reset or initialized, the value stored in ∗value is undefined.

Parameters

address	the IME address to fetch from 0 to IME_ADDR_MAX
value	a pointer to the location where the value will be stored (obtained using the "&" operator on the target variable name e.g. `imeGet(2, &counts)`)

Returns

true if the count was successfully read and the value stored in ∗value is valid; false otherwise

8.1.4.40 imeGetVelocity()

```
bool imeGetVelocity (
            unsigned char address,
            int * value )
```

Gets the current rotational velocity of the specified IME.

In this version of PROS, the velocity is positive if the IME count is increasing and negative if the IME count is decreasing. The velocity is in RPM of the internal encoder wheel. Since checking the IME for its type cannot reveal whether the motor gearing is high speed or high torque (in the 2-Wire Motor 393 case), the user must divide the return value by the number of output revolutions per encoder revolution:

- `30.056` for the 269 IME
- `39.2` for the 393 IME in high torque mode (factory default)
- `24.5` for the 393 IME in high speed mode

If the IME address is invalid, or the IME has not been reset or initialized, the value stored in ∗value is undefined.

Parameters

address	the IME address to fetch from 0 to IME_ADDR_MAX
value	a pointer to the location where the value will be stored (obtained using the "&" operator on the target variable name e.g. `imeGetVelocity(2, &counts)`)

Returns

true if the velocity was successfully read and the value stored in ∗value is valid; false otherwise

8.1.4.41 imeInitializeAll()

```
unsigned int imeInitializeAll ( )
```

Initializes all IMEs.

IMEs are assigned sequential incrementing addresses, beginning with the first IME on the chain (closest to the VEX Cortex I2C port). Therefore, a given configuration of IMEs will always have the same ID assigned to each encoder. The addresses range from 0 to IME_ADDR_MAX, so the first encoder gets 0, the second gets 1, ...

This function should most likely be used in initialize(). Do not use it in initializeIO() or at any other time when the scheduler is paused (like an interrupt). Checking the return value of this function is important to ensure that all IMEs are plugged in and responding as expected.

This function, unlike the other IME functions, is not thread safe. If using imeInitializeAll to re-initialize encoders, calls to other IME functions might behave unpredictably during this function's execution.

Returns

the number of IMEs successfully initialized.

8.1.4.42 imeReset()

```
bool imeReset (
              unsigned char address )
```

Resets the specified IME's counters to zero.

This method can be used while the IME is rotating.

Parameters

address	the IME address to reset from 0 to IME_ADDR_MAX

Returns

true if the reset succeeded; false otherwise

8.1.4.43 imeShutdown()

```
void imeShutdown ( )
```

Shuts down all IMEs on the chain; their addresses return to the default and the stored counts and velocities are lost. This function, unlike the other IME functions, is not thread safe.

To use the IME chain again, wait at least 0.25 seconds before using imeInitializeAll again.

8.1.4.44 ioClearInterrupt()

```
void ioClearInterrupt (
              unsigned char pin )
```

Disables interrupts on the specified pin.

Disabling interrupts on interrupt pins which are not in use conserves processing time.

Parameters

pin	the pin on which to reset interrupts from 1-9,11-12

8.1.4.45 ioSetInterrupt()

```
void ioSetInterrupt (
            unsigned char pin,
            unsigned char edges,
            InterruptHandler handler )
```

Sets up an interrupt to occur on the specified pin, and resets any counters or timers associated with the pin.

Each time the specified change occurs, the function pointer passed in will be called with the pin that changed as an argument. Enabling pin-change interrupts consumes processing time, so it is best to only enable necessary interrupts and to keep the InterruptHandler function short. Pin change interrupts can only be enabled on pins 1-9 and 11-12.

Do not use API functions such as delay() inside the handler function, as the function will run in an ISR where the scheduler is paused and no other interrupts can execute. It is best to quickly update some state and allow a task to perform the work.

Do not use this function on pins that are also being used by the built-in ultrasonic or shaft encoder drivers, or on pins which have been switched to output mode.

Parameters

pin	the pin on which to enable interrupts from 1-9,11-12
edges	one of INTERRUPT_EDGE_RISING, INTERRUPT_EDGE_FALLING, or INTERRUPT_EDGE_BOTH
handler	the function to call when the condition is satisfied

8.1.4.46 isAutonomous()

```
bool isAutonomous ( )
```

Returns true if the robot is in autonomous mode, or false otherwise. While in autonomous mode, joystick inputs will return a neutral value, but serial port communications (even over VexNET) will still work properly.

8.1.4.47 isEnabled()

```
bool isEnabled ( )
```

Returns true if the robot is enabled, or false otherwise. While disabled via the VEX Competition Switch or VEX Field Controller, motors will not function. However, the digital I/O ports can still be changed, which may indirectly affect the robot state (e.g. solenoids). Avoid performing externally visible actions while disabled (the kernel should take care of this most of the time).

8.1.4.48 isJoystickConnected()

```
bool isJoystickConnected (
            unsigned char joystick )
```

Returns true if a joystick is connected to the specified slot number (1 or 2), or false otherwise. Useful for automatically merging joysticks for one operator, or splitting for two. This function does not work properly during initialize() or initializeIO() and can return false positives. It should be checked once and stored at the beginning of operator↩ Control().

8.1.4.49 isOnline()

```
bool isOnline ( )
```

Returns true if a VEX field controller or competition switch is connected, or false otherwise. When in online mode, the switching between autonomous() and operatorControl() tasks is managed by the PROS kernel.

8.1.4.50 iwdgEnable()

```
void iwdgEnable ( )
```

Enables IWDG watchdog timer which will reset the cortex if it locks up due to static shock or a misbehaving task preventing the timer to be reset. Not recovering from static shock will cause the robot to continue moving its motors indefinitely until turned off manually.

This function should only be called once in initializeIO()

8.1.4.51 joystickGetAnalog()

```
int joystickGetAnalog (
            unsigned char joystick,
            unsigned char axis )
```

Gets the value of a control axis on the VEX joystick. Returns the value from -127 to 127, or 0 if no joystick is connected to the requested slot.

Parameters

joystick	the joystick slot to check
axis	one of 1, 2, 3, 4, ACCEL_X, or ACCEL_Y

8.1.4.52 joystickGetDigital()

```
bool joystickGetDigital (
            unsigned char joystick,
```

```
            unsigned char buttonGroup,
            unsigned char button )
```

Gets the value of a button on the VEX joystick. Returns true if that button is pressed, or false otherwise. If no joystick is connected to the requested slot, returns false.

Parameters

joystick	the joystick slot to check
buttonGroup	one of 5, 6, 7, or 8 to request that button as labelled on the joystick
button	one of JOY_UP, JOY_DOWN, JOY_LEFT, or JOY_RIGHT; requesting JOY_LEFT or JOY_RIGHT for groups 5 or 6 will cause an undefined value to be returned

8.1.4.53 lcdClear()

```
void lcdClear (
            PROS_FILE * lcdPort )
```

Clears the LCD screen on the specified port.

Printing to a line implicitly overwrites the contents, so clearing should only be required at startup.

Parameters

lcdPort	the LCD to clear, either uart1 or uart2

8.1.4.54 lcdInit()

```
void lcdInit (
            PROS_FILE * lcdPort )
```

Initializes the LCD port, but does not change the text or settings.

If the LCD was not initialized before, the text currently on the screen will be undefined. The port will not be usable with standard serial port functions until the LCD is stopped.

Parameters

lcdPort	the LCD to initialize, either uart1 or uart2

8.1.4.55 lcdReadButtons()

```
void unsigned char const char unsigned int lcdReadButtons (
            PROS_FILE * lcdPort )
```

Reads the user button status from the LCD display.

For example, if the left and right buttons are pushed, (1 | 4) = 5 will be returned. 0 is returned if no buttons are pushed.

Parameters

lcdPort	the LCD to poll, either uart1 or uart2

Returns

the buttons pressed as a bit mask

8.1.4.56 lcdSetBacklight()

```
void lcdSetBacklight (
            PROS_FILE * lcdPort,
            bool backlight )
```

Sets the specified LCD backlight to be on or off.

Turning it off will save power but may make it more difficult to read in dim conditions.

Parameters

lcdPort	the LCD to adjust, either uart1 or uart2
backlight	true to turn the backlight on, or false to turn it off

8.1.4.57 lcdSetText()

```
void lcdSetText (
            PROS_FILE * lcdPort,
            unsigned char line,
            const char * buffer )
```

Prints the string buffer to the attached LCD.

The output string will be truncated as necessary to fit on the LCD screen, 16 characters wide. This function, like fprint(), is much, much faster than a formatted routine such as lcdPrint() and consumes less memory.

Parameters

lcdPort	the LCD to write, either uart1 or uart2
line	the LCD line to write, either 1 or 2
buffer	the string to write

8.1.4.58 lcdShutdown()

```
void lcdShutdown (
            PROS_FILE * lcdPort )
```

Shut down the specified LCD port.

Parameters

lcdPort	the LCD to stop, either uart1 or uart2

8.1.4.59 micros()

```
unsigned long micros ( )
```

Returns the number of microseconds since Cortex power-up. There are 10^6 microseconds in a second, so as a 32-bit integer, this will overflow and wrap back to zero every two hours or so.

This function is Wiring-compatible.

Returns

the number of microseconds since the Cortex was turned on or the last overflow

8.1.4.60 millis()

```
unsigned long millis ( )
```

Returns the number of milliseconds since Cortex power-up. There are 1000 milliseconds in a second, so as a 32-bit integer, this will not overflow for 50 days.

This function is Wiring-compatible.

Returns

the number of milliseconds since the Cortex was turned on

8.1.4.61 motorGet()

```
int motorGet (
            unsigned char channel )
```

Gets the last set speed of the specified motor channel.

This speed may have been set by any task or the PROS kernel itself. This is not guaranteed to be the speed that the motor is actually running at, or even the speed currently being sent to the motor, due to latency in the Motor Controller 29 protocol and physical loading. To measure actual motor shaft revolution speed, attach a VEX Integrated Motor Encoder or VEX Quadrature Encoder and use the velocity functions associated with each.

Parameters

channel	the motor channel to fetch from 1-10

Returns

the speed last sent to this channel; -127 is full reverse and 127 is full forward, with 0 being off

8.1.4.62 motorSet()

```
void motorSet (
            unsigned char channel,
            int speed )
```

Sets the speed of the specified motor channel.

Do not use motorSet() with the same channel argument from two different tasks. It is safe to use motorSet() with different channel arguments from different tasks.

Parameters

channel	the motor channel to modify from 1-10
speed	the new signed speed; -127 is full reverse and 127 is full forward, with 0 being off

8.1.4.63 motorStop()

```
void motorStop (
            unsigned char channel )
```

Stops the motor on the specified channel, equivalent to calling motorSet() with an argument of zero.

This performs a coasting stop, not an active brake. Since motorStop is similar to motorSet(0), see the note for motorSet() about use from multiple tasks.

Parameters

channel	the motor channel to stop from 1-10

8.1.4.64 motorStopAll()

```
void motorStopAll ( )
```

Stops all motors; significantly faster than looping through all motor ports and calling motorSet(channel, 0) on each one.

8.1.4.65 mutexCreate()

```
Mutex mutexCreate ( )
```

Creates a mutex intended to allow only one task to use a resource at a time. For signalling and synchronization, try using semaphores.

Mutexes created using this function can be accessed using the mutexTake() and mutexGive() functions. The semaphore functions must not be used on objects of this type.

This type of object uses a priority inheritance mechanism so a task 'taking' a mutex MUST ALWAYS 'give' the mutex back once the mutex is no longer required.

Returns

a handle to the created mutex

8.1.4.66 mutexDelete()

```
void mutexDelete (
            Mutex mutex )
```

Deletes the specified mutex. This function can be dangerous; deleting semaphores being waited on by a task may cause deadlock or a crash.

Parameters

mutex	the mutex to destroy

8.1.4.67 mutexGive()

```
bool mutexGive (
            Mutex mutex )
```

Relinquishes a mutex so that other tasks can use the resource it guards. The mutex must be held by the current task using a corresponding call to mutexTake.

Parameters

mutex	the mutex to release

Returns

true if the mutex was released, or false if the mutex was not already held

8.1.4.68 mutexTake()

```
bool mutexTake (
            Mutex mutex,
            const unsigned long blockTime )
```

Requests a mutex so that other tasks cannot simultaneously use the resource it guards. The mutex must not already be held by the current task. If another task already holds the mutex, the function will wait for the mutex to be released. Other tasks can run during this time.

Parameters

mutex	the mutex to request
blockTime	the maximum time to wait for the mutex to be available, where -1 specifies an infinite timeout

Returns

true if the mutex was successfully taken, or false if the timeout expired

8.1.4.69 pinMode()

```
void pinMode (
            unsigned char pin,
            unsigned char mode )
```

Configures the pin as an input or output with a variety of settings.

Do note that INPUT by default turns on the pull-up resistor, as most VEX sensors are open-drain active low. It should not be a big deal for most push-pull sources. This function is Wiring-compatible.

Parameters

pin	the pin to modify from 1-26
mode	one of INPUT, INPUT_ANALOG, INPUT_FLOATING, OUTPUT, or OUTPUT_OD

8.1.4.70 powerLevelBackup()

```
unsigned int powerLevelBackup ( )
```

Returns the backup battery voltage in millivolts.

If no backup battery is connected, returns 0.

8.1.4.71 powerLevelMain()

```
unsigned int powerLevelMain ( )
```

Returns the main battery voltage in millivolts.

In rare circumstances, this method might return 0. Check the output value for reasonability before blindly blasting the user.

8.1.4.72 print()

```
void print (
            const char * string )
```

Prints the simple string to the debug terminal without formatting.

This method is much, much faster than printf().

Parameters

string	the string to write

8.1.4.73 printf()

```
int printf (
            const char * formatString,
             ...  )
```

Prints the formatted string to the debug stream (the PC terminal).

Parameters

formatString	the format string as specified in fprintf()

Returns

the number of characters written

8.1.4.74 putchar()

```
int putchar (
            int value )
```

Writes one character to "stdout", which is the PC debug terminal, and returns the input value.

When using a wireless connection, one may need to press the spacebar before the input is visible on the terminal.

Parameters

value	the character to write (a value of type "char" can be used)

Returns

the character written

8.1.4.75 puts()

```
int puts (
            const char * string )
```

Behaves the same as the "print" function, and appends a trailing newline ("\n").

Parameters

string	the string to write

Returns

the number of characters written, excluding the new line

8.1.4.76 semaphoreCreate()

```
Semaphore semaphoreCreate ( )
```

Creates a semaphore intended for synchronizing tasks. To prevent some critical code from simultaneously modifying a shared resource, use mutexes instead.

Semaphores created using this function can be accessed using the semaphoreTake() and semaphoreGive() functions. The mutex functions must not be used on objects of this type.

This type of object does not need to have balanced take and give calls, so priority inheritance is not used. Semaphores can be signalled by an interrupt routine.

Returns

a handle to the created semaphore

8.1.4.77 semaphoreDelete()

```
void semaphoreDelete (
            Semaphore semaphore )
```

Deletes the specified semaphore. This function can be dangerous; deleting semaphores being waited on by a task may cause deadlock or a crash.

Parameters

semaphore	the semaphore to destroy

8.1.4.78 semaphoreGive()

```
bool semaphoreGive (
            Semaphore semaphore )
```

Signals a semaphore. Tasks waiting for a signal using semaphoreTake() will be unblocked by this call and can continue execution.

Slow processes can give semaphores when ready, and fast processes waiting to take the semaphore will continue at that point.

Parameters

semaphore	the semaphore to signal

Returns

true if the semaphore was successfully given, or false if the semaphore was not taken since the last give

8.1.4.79 semaphoreTake()

```
bool semaphoreTake (
            Semaphore semaphore,
            const unsigned long blockTime )
```

Waits on a semaphore. If the semaphore is already in the "taken" state, the current task will wait for the semaphore to be signaled. Other tasks can run during this time.

Parameters

semaphore	the semaphore to wait
blockTime	the maximum time to wait for the semaphore to be given, where -1 specifies an infinite timeout

Returns

true if the semaphore was successfully taken, or false if the timeout expired

8.1.4.80 setTeamName()

```
void setTeamName (
             const char * name )
```

Sets the team name displayed to the VEX field control and VEX Firmware Upgrade.

Parameters

name	a string containing the team name; only the first eight characters will be shown

8.1.4.81 snprintf()

```
int snprintf (
             char * buffer,
             size_t limit,
             const char * formatString,
              ...  )
```

Prints the formatted string to the string buffer with the specified length limit.

The length limit, as per the C standard, includes the trailing null character, so an argument of 256 will cause a maximum of 255 non-null characters to be printed, and one null terminator in all cases.

Parameters

buffer	the string buffer where characters can be placed
limit	the maximum number of characters to write
formatString	the format string as specified in fprintf()

Returns

the number of characters stored

8.1.4.82 speakerInit()

```
void speakerInit ( )
```

Initializes VEX speaker support.

The VEX speaker is not thread safe; it can only be used from one task at a time. Using the VEX speaker may impact robot performance. Teams may benefit from an if statement that only enables sound if isOnline() returns false.

8.1.4.83 speakerPlayArray()

```
void speakerPlayArray (
            const char ** songs )
```

Plays up to three RTTTL (Ring Tone Text Transfer Language) songs simultaneously over the VEX speaker. The audio is mixed to allow polyphonic sound to be played. Many simple songs are available in RTTTL format online, or compose your own.

The song must not be NULL, but unused tracks within the song can be set to NULL. If any of the three song tracks is invalid, the result of this function is undefined.

The VEX speaker is not thread safe; it can only be used from one task at a time. Using the VEX speaker may impact robot performance. Teams may benefit from an if statement that only enables sound if isOnline() returns false.

Parameters

songs	an array of up to three (3) RTTTL songs as string values to play

8.1.4.84 speakerPlayRtttl()

```
void speakerPlayRtttl (
            const char * song )
```

Plays an RTTTL (Ring Tone Text Transfer Language) song over the VEX speaker. Many simple songs are available in RTTTL format online, or compose your own.

The song must not be NULL. If an invalid song is specified, the result of this function is undefined.

The VEX speaker is not thread safe; it can only be used from one task at a time. Using the VEX speaker may impact robot performance. Teams may benefit from an if statement that only enables sound if isOnline() returns false.

Parameters

song	the RTTTL song as a string value to play

8.1.4.85 speakerShutdown()

```
void speakerShutdown ( )
```

Powers down and disables the VEX speaker.

If a song is currently being played in another task, the behavior of this function is undefined, since the VEX speaker is not thread safe.

8.1.4.86 sprintf()

```
int sprintf (
             char * buffer,
             const char * formatString,
              ...  )
```

Prints the formatted string to the string buffer.

If the buffer is not big enough to contain the complete formatted output, undefined behavior occurs. See snprintf() for a safer version of this function.

Parameters

buffer	the string buffer where characters can be placed
formatString	the format string as specified in fprintf()

Returns

the number of characters stored

8.1.4.87 taskCreate()

```
TaskHandle taskCreate (
             TaskCode taskCode,
             const unsigned int stackDepth,
             void * parameters,
             const unsigned int priority )
```

Creates a new task and add it to the list of tasks that are ready to run.

Parameters

taskCode	the function to execute in its own task
stackDepth	the number of variables available on the stack (4 * stackDepth bytes will be allocated on the Cortex)
parameters	an argument passed to the taskCode function
priority	a value from TASK_PRIORITY_LOWEST to TASK_PRIORITY_HIGHEST determining the initial priority of the task

Returns

a handle to the created task, or NULL if an error occurred

8.1.4.88 taskDelay()

```
void taskDelay (
             const unsigned long msToDelay )
```

Delays the current task for a given number of milliseconds.

Delaying for a period of zero will force a reschedule, where tasks of equal priority may be scheduled if available. The calling task will still be available for immediate rescheduling once the other tasks have had their turn or if nothing of equal or higher priority is available to be scheduled.

This is not the best method to have a task execute code at predefined intervals, as the delay time is measured from when the delay is requested. To delay cyclically, use taskDelayUntil().

Parameters

msToDelay	the number of milliseconds to wait, with 1000 milliseconds per second

8.1.4.89 taskDelayUntil()

```
void taskDelayUntil (
            unsigned long * previousWakeTime,
            const unsigned long cycleTime )
```

Delays the current task until a specified time. The task will be unblocked at the time ∗previousWakeTime + cycle↩ Time, and ∗previousWakeTime will be changed to reflect the time at which the task will unblock.

If the target time is in the past, no delay occurs, but a reschedule is forced, as if taskDelay() was called with an argument of zero. If the sum of cycleTime and ∗previousWakeTime overflows or underflows, undefined behavior occurs.

This function should be used by cyclical tasks to ensure a constant execution frequency. While taskDelay() specifies a wake time relative to the time at which the function is called, taskDelayUntil() specifies the absolute future time at which it wishes to unblock. Calling taskDelayUntil with the same cycleTime parameter value in a loop, with previousWakeTime referring to a local variable initialized to millis(), will cause the loop to execute with a fixed period.

Parameters

previousWakeTime	a pointer to the location storing the last unblock time, obtained by using the "&" operator on a variable (e.g. "taskDelayUntil(&now, 50);")
cycleTime	the number of milliseconds to wait, with 1000 milliseconds per second

8.1.4.90 taskDelete()

```
void taskDelete (
            TaskHandle taskToDelete )
```

Kills and removes the specified task from the kernel task list.

Deleting the last task will end the program, possibly leading to undesirable states as some outputs may remain in their last set configuration.

NOTE: The idle task is responsible for freeing the kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of processing time. Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

Parameters

taskToDelete	the task to kill; passing NULL kills the current task

8.1.4.91 taskGetCount()

```
unsigned int taskGetCount ( )
```

Determines the number of tasks that are currently being managed.

This includes all ready, blocked and suspended tasks. A task that has been deleted but not yet freed by the idle task will also be included in the count. Tasks recently created may take one context switch to be counted.

Returns

the number of tasks that are currently running, waiting, or suspended

8.1.4.92 taskGetState()

```
unsigned int taskGetState (
            TaskHandle task )
```

Retrieves the state of the specified task. Note that the state of tasks which have died may be re-used for future tasks, causing the value returned by this function to reflect a different task than possibly intended in this case.

Parameters

task	Handle to the task to query. Passing NULL will query the current task status (which will, by definition, be TASK_RUNNING if this call returns)

Returns

A value reflecting the task's status, one of the constants TASK_DEAD, TASK_RUNNING, TASK_RUNNABLE, TASK_SLEEPING, or TASK_SUSPENDED

8.1.4.93 taskPriorityGet()

```
unsigned int taskPriorityGet (
            const TaskHandle task )
```

Obtains the priority of the specified task.

Parameters

task	the task to check; passing NULL checks the current task

Returns

the priority of that task from 0 to TASK_MAX_PRIORITIES

8.1.4.94 taskPrioritySet()

```
void taskPrioritySet (
            TaskHandle task,
            const unsigned int newPriority )
```

Sets the priority of the specified task.

A context switch may occur before the function returns if the priority being set is higher than the currently executing task and the task being mutated is available to be scheduled.

Parameters

task	the task to change; passing NULL changes the current task
newPriority	a value between TASK_PRIORITY_LOWEST and TASK_PRIORITY_HIGHEST inclusive indicating the new task priority

8.1.4.95 taskResume()

```
void taskResume (
            TaskHandle taskToResume )
```

Resumes the specified task.

A task that has been suspended by one or more calls to taskSuspend() will be made available for scheduling again by a call to taskResume(). If the task was not suspended at the time of the call to taskResume(), undefined behavior occurs.

Parameters

taskToResume	the task to change; passing NULL is not allowed as the current task cannot be suspended (it is obviously running if this function is called)

8.1.4.96 taskRunLoop()

```
TaskHandle taskRunLoop (
            void(*)(void) fn,
            const unsigned long increment )
```

Starts a task which will periodically call the specified function.

Intended for use as a quick-start skeleton for cyclic tasks with higher priority than the "main" tasks. The created task will have priority TASK_PRIORITY_DEFAULT + 1 with the default stack size. To customize behavior, create a task manually with the specified function.

This task will automatically terminate after one further function invocation when the robot is disabled or when the robot mode is switched.

Parameters

fn	the function to call in this loop
increment	the delay between successive calls in milliseconds; the taskDelayUntil() function is used for accurate cycle timing

Returns

a handle to the task, or NULL if an error occurred

8.1.4.97 taskSuspend()

```
void taskSuspend (
            TaskHandle taskToSuspend )
```

Suspends the specified task.

When suspended a task will not be scheduled, regardless of whether it might be otherwise available to run.

Parameters

taskToSuspend	the task to suspend; passing NULL suspends the current task

8.1.4.98 ultrasonicGet()

```
int ultrasonicGet (
            Ultrasonic ult )
```

Gets the current ultrasonic sensor value in centimeters.

If no object was found, zero is returned. If the ultrasonic sensor was never started, the return value is undefined. Round and fluffy objects can cause inaccurate values to be returned.

Parameters

ult	the Ultrasonic object from ultrasonicInit() to read

Returns

the distance to the nearest object in centimeters

8.1.4.99 ultrasonicInit()

```
Ultrasonic ultrasonicInit (
            unsigned char portEcho,
            unsigned char portPing )
```

Initializes an ultrasonic sensor on the specified digital ports.

The ultrasonic sensor will be polled in the background in concert with the other sensors registered using this method. NULL will be returned if either port is invalid or the ultrasonic sensor port is already in use.

Parameters

portEcho	the port connected to the orange cable from 1-9,11-12
portPing	the port connected to the yellow cable from 1-12

Returns

an Ultrasonic object to be stored and used for later calls to ultrasonic functions

8.1.4.100 ultrasonicShutdown()

```
void ultrasonicShutdown (
            Ultrasonic ult )
```

Stops and disables the ultrasonic sensor.

The last distance it had before stopping will be retained. One more ping operation may occur before the sensor is fully disabled.

Parameters

ult	the Ultrasonic object from ultrasonicInit() to stop

8.1.4.101 usartInit()

```
void usartInit (
            PROS_FILE * usart,
            unsigned int baud,
            unsigned int flags )
```

Initialize the specified serial interface with the given connection parameters.

I/O to the port is accomplished using the "standard" I/O functions such as fputs(), fprintf(), and fputc().

Re-initializing an open port may cause loss of data in the buffers. This routine may be safely called from initializeIO() or when the scheduler is paused. If I/O is attempted on a serial port which has never been opened, the behavior will be the same as if the port had been disabled.

Parameters

usart	the port to open, either "uart1" or "uart2"
baud	the baud rate to use from 2400 to 1000000 baud
flags	a bit mask combination of the SERIAL_* flags specifying parity, stop, and data bits

8.1.4.102 usartShutdown()

```
void usartShutdown (
            PROS_FILE * usart )
```

Disables the specified USART interface.

Any data in the transmit and receive buffers will be lost. Attempts to read from the port when it is disabled will deadlock, and attempts to write to it may deadlock depending on the state of the buffer.

Parameters

usart	the port to close, either "uart1" or "uart2"

8.1.4.103 wait()

```
void wait (
            const unsigned long time )
```

Alias of taskDelay() intended to help EasyC users.

Parameters

time	the duration of the delay in milliseconds (1 000 milliseconds per second)

8.1.4.104 waitUntil()

```
void waitUntil (
            unsigned long * previousWakeTime,
            const unsigned long time )
```

Alias of taskDelayUntil() intended to help EasyC users.

Parameters

previousWakeTime	a pointer to the last wakeup time
time	the duration of the delay in milliseconds (1 000 milliseconds per second)

8.2 include/main.h File Reference

Header file for global functions.

```
#include "API.h"
#include "debug.hpp"
```

Functions

- void autonomous ()
- void initializeIO ()
- void initialize ()
- void operatorControl ()

8.2.1 Detailed Description

Header file for global functions.

Any experienced C or C++ programmer knows the importance of header files. For those who do not, a header file allows multiple files to reference functions in other files without necessarily having to see the code (and therefore causing a multiple definition). To make a function in "opcontrol.c", "auto.c", "main.c", or any other C file visible to the core implementation files, prototype it here. main

- Neither the name of Purdue University ACM SIG BOTS nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Purdue Robotics OS contains FreeRTOS (http://www.freertos.org) whose source code may be obtained from http://sourceforge.net/projects/freertos/files/ or on request.

8.2.2 Function Documentation

8.2.2.1 autonomous()

```
void autonomous ( )
```

Runs the user autonomous code. This function will be started in its own task with the default priority and stack size whenever the robot is enabled via the Field Management System or the VEX Competition Switch in the autonomous mode. If the robot is disabled or communications is lost, the autonomous task will be stopped by the kernel. Re-enabling the robot will restart the task, not re-start it from where it left off.

Code running in the autonomous task cannot access information from the VEX Joystick. However, the autonomous function can be invoked from another task if a VEX Competition Switch is not available, and it can access joystick information if called in this way.

The autonomous task may exit, unlike operatorControl() which should never exit. If it does so, the robot will await a switch to another mode or disable/enable cycle.

8.2.2.2 initialize()

```
void initialize ( )
```

Runs user initialization code. This function will be started in its own task with the default priority and stack size once when the robot is starting up. It is possible that the VEXnet communication link may not be fully established at this time, so reading from the VEX Joystick may fail.

This function should initialize most sensors (gyro, encoders, ultrasonics), LCDs, global variables, and IMEs.

This function must exit relatively promptly, or the operatorControl() and autonomous() tasks will not start. An autonomous mode selection menu like the pre_auton() in other environments can be implemented in this task if desired.

8.2.2.3 initializeIO()

```
void initializeIO ( )
```

Runs pre-initialization code. This function will be started in kernel mode one time while the VEX Cortex is starting up. As the scheduler is still paused, most API functions will fail.

The purpose of this function is solely to set the default pin modes (pinMode()) and port states (digitalWrite()) of limit switches, push buttons, and solenoids. It can also safely configure a UART port (usartOpen()) but cannot set up an LCD (lcdInit()).

8.2.2.4 operatorControl()

```
void operatorControl ( )
```

Runs the user operator control code. This function will be started in its own task with the default priority and stack size whenever the robot is enabled via the Field Management System or the VEX Competition Switch in the operator control mode. If the robot is disabled or communications is lost, the operator control task will be stopped by the kernel. Re-enabling the robot will restart the task, not resume it from where it left off.

If no VEX Competition Switch or Field Management system is plugged in, the VEX Cortex will run the operator control task. Be warned that this will also occur if the VEX Cortex is tethered directly to a computer via the USB A to A cable without any VEX Joystick attached.

Code running in this task can take almost any action, as the VEX Joystick is available and the scheduler is operational. However, proper use of delay() or taskDelayUntil() is highly recommended to give other tasks (including system tasks such as updating LCDs) time to run.

This task should never exit; it should end with some kind of infinite loop, even if empty.