9228B Turning Point Code

2.1.3

Generated by Doxygen 1.8.14

Contents

1 Todo List

Global GYRO_MULTIPLIER (p. **??**)

2 Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

3 File Index

3.1 File List

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

4 Data Structure Documentation

4.1 lcd_buttons Struct Reference

represents the state of the lcd buttons

`#include <lcd.h>`

Data Fields

- **button_state left**
- **button_state middle**
- **button_state right**

4.1.1 Detailed Description

represents the state of the lcd buttons

Author

Chris Jerrett

Date

9/9/2017

Definition at line **48** of file **lcd.h**.

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

- include/ **lcd.h**

5 File Documentation

5.1 include/API.h File Reference

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

```
#include <stdlib.h>
#include <stdbool.h>
#include <stdarg.h>
#include <stdint.h>
#include <math.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 **ULTRA_BAD_RESPONSE** -1
- #define **FILE** **PROS_FILE**
- #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)

Typedefs

Functions

- 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 **watchdogInit** ()
- void **standaloneModeEnable** ()

Variables

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

5.1.1 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** (p. **??**) instead of referencing **API.h** (p. **??**) 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.

Definition in file **API.h**.

5.1.2 Macro Definition Documentation

5.1.2.1 ACCEL_X

```
#define ACCEL_X 5
```

Analog axis for the X acceleration from the VEX Joystick.

Definition at line **58** of file **API.h**.

5.1.2.2 ACCEL_Y

```
#define ACCEL_Y 6
```

Analog axis for the Y acceleration from the VEX Joystick.

Definition at line **62** of file **API.h**.

5.1.2.3 BOARD_NR_ADC_PINS

```
#define BOARD_NR_ADC_PINS 8
```

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

Definition at line **143** of file **API.h**.

5.1.2.4 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.

Definition at line **153** of file **API.h**.

5.1.2.5 EOF

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

EOF is a value evaluating to -1.

Definition at line **848** of file **API.h**.

5.1.2.6 FILE

```
#define FILE  PROS_FILE
```

For convenience, FILE is defined as PROS_FILE if it wasn't already defined. This provides backwards compatability with PROS, but also allows libraries such as newlib to be incorporated into PROS projects. If you're not using C++/newlib, you can disregard this and just use FILE.

Definition at line **761** of file **API.h**.

5.1.2.7 HIGH

```
#define HIGH 1
```

Used for **digitalWrite()** (p. **??**) 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.

Definition at line **159** of file **API.h**.

5.1.2.8 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.

Definition at line **460** of file **API.h**.

5.1.2.9 INPUT

```
#define INPUT 0x0A
```

pinMode() (p. **??**) 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.

Definition at line **174** of file **API.h**.

5.1.2.10 INPUT_ANALOG

```
#define INPUT_ANALOG 0x00
```

pinMode() (p. **??**) 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.

Definition at line **181** of file **API.h**.

5.1.2.11 INPUT_FLOATING

```
#define INPUT_FLOATING 0x04
```

pinMode() (p. **??**) state for digital input, without pullup.

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

Definition at line **188** of file **API.h**.

5.1.2.12 INTERRUPT_EDGE_BOTH

```
#define INTERRUPT_EDGE_BOTH 3
```

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

Definition at line **329** of file **API.h**.

5.1.2.13 INTERRUPT_EDGE_FALLING

```
#define INTERRUPT_EDGE_FALLING 2
```

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

Definition at line **324** of file **API.h**.

5.1.2.14 INTERRUPT_EDGE_RISING

```
#define INTERRUPT_EDGE_RISING 1
```

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

Definition at line **320** of file **API.h**.

5.1.2.15 JOY_DOWN

```
#define JOY_DOWN 1
```

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

Definition at line **42** of file **API.h**.

5.1.2.16 JOY_LEFT

```
#define JOY_LEFT 2
```

LEFT button (valid on channels 7, 8)

Definition at line **46** of file **API.h**.

5.1.2.17 JOY_RIGHT

```
#define JOY_RIGHT 8
```

RIGHT button (valid on channels 7, 8)

Definition at line **54** of file **API.h**.

5.1.2.18 JOY_UP

```
#define JOY_UP 4
```

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

Definition at line **50** of file **API.h**.

Referenced by **drive()**.

5.1.2.19 LCD_BTN_CENTER

```
#define LCD_BTN_CENTER 2
```

CENTER button on LCD for use with **lcdReadButtons()** (p. **??**)

Definition at line **1146** of file **API.h**.

5.1.2.20 LCD_BTN_LEFT

```
#define LCD_BTN_LEFT 1
```

LEFT button on LCD for use with **lcdReadButtons()** (p. **??**)

Definition at line **1142** of file **API.h**.

5.1.2.21 LCD_BTN_RIGHT

```
#define LCD_BTN_RIGHT 4
```

RIGHT button on LCD for use with **lcdReadButtons()** (p. **??**)

Definition at line **1150** of file **API.h**.

5.1.2.22 LOW

```
#define LOW 0
```

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

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

Definition at line **165** of file **API.h**.

5.1.2.23 OUTPUT

```
#define OUTPUT 0x01
```

pinMode() (p. **??**) 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.

Definition at line **195** of file **API.h**.

5.1.2.24 OUTPUT_OD

```
#define OUTPUT_OD 0x05
```

pinMode() (p. **??**) 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.

Definition at line **202** of file **API.h**.

5.1.2.25 SEEK_CUR

```
#define SEEK_CUR 1
```

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

Definition at line **863** of file **API.h**.

5.1.2.26 SEEK_END

```
#define SEEK_END 2
```

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

Definition at line **870** of file **API.h**.

5.1.2.27 SEEK_SET

```
#define SEEK_SET 0
```

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

Definition at line **856** of file **API.h**.

5.1.2.28 SERIAL_8N1

```
#define SERIAL_8N1 0x0000
```

Specifies the default serial settings when used in **usartInit()** (p. **??**)

Definition at line **795** of file **API.h**.

5.1.2.29 SERIAL_DATABITS_8

```
#define SERIAL_DATABITS_8 0x0000
```

Bit mask for **usartInit()** (p. **??**) for 8 data bits (typical)

Definition at line **767** of file **API.h**.

5.1.2.30 SERIAL_DATABITS_9

```
#define SERIAL_DATABITS_9 0x1000
```

Bit mask for **usartInit()** (p. **??**) for 9 data bits

Definition at line **771** of file **API.h**.

5.1.2.31 SERIAL_PARITY_EVEN

```
#define SERIAL_PARITY_EVEN 0x0400
```

Bit mask for **usartInit()** (p. **??**) for Even parity

Definition at line **787** of file **API.h**.

5.1.2.32 SERIAL_PARITY_NONE

```
#define SERIAL_PARITY_NONE 0x0000
```

Bit mask for **usartInit()** (p. **??**) for No parity (typical)

Definition at line **783** of file **API.h**.

5.1.2.33 SERIAL_PARITY_ODD

```
#define SERIAL_PARITY_ODD 0x0600
```

Bit mask for **usartInit()** (p. **??**) for Odd parity

Definition at line **791** of file **API.h**.

5.1.2.34 SERIAL_STOPBITS_1

```
#define SERIAL_STOPBITS_1 0x0000
```

Bit mask for **usartInit()** (p. **??**) for 1 stop bit (typical)

Definition at line **775** of file **API.h**.

5.1.2.35 SERIAL_STOPBITS_2

```
#define SERIAL_STOPBITS_2 0x2000
```

Bit mask for **usartInit()** (p. **??**) for 2 stop bits

Definition at line **779** of file **API.h**.

5.1.2.36 stdin

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

The standard input stream uses the PC debug terminal.

Definition at line **834** of file **API.h**.

5.1.2.37 stdout

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

The standard output stream uses the PC debug terminal.

Definition at line **830** of file **API.h**.

5.1.2.38 TASK_DEAD

```
#define TASK_DEAD 0
```

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

Definition at line **1277** of file **API.h**.

5.1.2.39 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()** (p. **??**).

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

Definition at line **1264** of file **API.h**.

5.1.2.40 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()** (p. **??**) or teleop() also counts against the limit, so 12 tasks usually remain for other uses.

Definition at line **1234** of file **API.h**.

5.1.2.41 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.

Definition at line **1240** of file **API.h**.

5.1.2.42 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.

Definition at line **1272** of file **API.h**.

5.1.2.43 TASK_PRIORITY_DEFAULT

```
#define TASK_PRIORITY_DEFAULT 2
```

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

Default tasks such as **autonomous()** (p. **??**) inherit this priority.

Definition at line **1251** of file **API.h**.

5.1.2.44 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.

Definition at line **1256** of file **API.h**.

5.1.2.45 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.

Definition at line **1245** of file **API.h**.

5.1.2.46 TASK_RUNNABLE

```
#define TASK_RUNNABLE 2
```

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

Definition at line **1286** of file **API.h**.

5.1.2.47 TASK_RUNNING

```
#define TASK_RUNNING 1
```

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

Definition at line **1281** of file **API.h**.

5.1.2.48 TASK_SLEEPING

```
#define TASK_SLEEPING 3
```

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

Definition at line **1291** of file **API.h**.

5.1.2.49 TASK_SUSPENDED

```
#define TASK_SUSPENDED 4
```

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

Definition at line **1295** of file **API.h**.

5.1.2.50 uart1

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

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

Definition at line **838** of file **API.h**.

5.1.2.51 uart2

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

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

Definition at line **842** of file **API.h**.

5.1.2.52 ULTRA_BAD_RESPONSE

```
#define ULTRA_BAD_RESPONSE -1
```

This value is returned if the sensor cannot find a reasonable value to return.

Definition at line **652** of file **API.h**.

5.1.3 Typedef Documentation

5.1.3.1 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.

Definition at line **607** of file **API.h**.

5.1.3.2 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.

Definition at line **550** of file **API.h**.

5.1.3.3 InterruptHandler

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

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

Definition at line **334** of file **API.h**.

5.1.3.4 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.

Definition at line **1308** of file **API.h**.

5.1.3.5 PROS_FILE

```
typedef int  PROS_FILE
```

PROS_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.

Definition at line **752** of file **API.h**.

5.1.3.6 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.

Definition at line **1314** of file **API.h**.

5.1.3.7 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); }

Definition at line **1325** of file **API.h**.

5.1.3.8 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.

Definition at line **1302** of file **API.h**.

5.1.3.9 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.

Definition at line **660** of file **API.h**.

5.1.4 Function Documentation

5.1.4.1 __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()** (p. **??**) 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()** (p. **??**)

5.1.4.2 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()** (p. **??**) and **analogReadCalibratedHR()** (p. **??**) functions. These functions will return the difference between this value and the current sensor value when called.

Do not use this function in **initializeIO()** (p. **??**), 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

5.1.4.3 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

5.1.4.4 analogReadCalibrated()

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

Reads the calibrated value of an analog input channel.

The **analogCalibrate()** (p. **??**) 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()** (p. **??**) 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

5.1.4.5 analogReadCalibratedHR()

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

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

The **analogCalibrate()** (p. **??**) 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

5.1.4.6 delay()

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

Wiring-compatible alias of **taskDelay()** (p. **??**).

Parameters

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

Referenced by **operatorControl()**, and **promt_confirmation()**.

5.1.4.7 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()** (p. **??**). 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)

5.1.4.8 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

5.1.4.9 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

5.1.4.10 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()** (p. **??**) to read

Returns

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

5.1.4.11 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

5.1.4.12 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()** (p. **??**) to reset

5.1.4.13 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()** (p. **??**) to stop

5.1.4.14 fclose()

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

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

Parameters

stream	the file descriptor to close from **fopen()** (p. **??**)

5.1.4.15 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

5.1.4.16 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()** (p. **??**) 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

5.1.4.17 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.

5.1.4.18 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()** (p. **??**) 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

5.1.4.19 fgetc()

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

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

Do not use **fgetc()** (p. **??**) 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

5.1.4.20 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

5.1.4.21 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()** (p. **??**) 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

5.1.4.22 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()** (p. **??**) and does not add a new line like **fputs()** (p. **??**). Do not use **fprint()** (p. **??**) on a VEX LCD port. Use **lcdSetText()** (p. **??**) instead.

Parameters

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

5.1.4.23 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()** (p. **??**) 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()** (p. **??**) 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

5.1.4.24 fputc()

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

Writes one character to the specified stream.

Do not use **fputc()** (p. **??**) on a VEX LCD port. Use **lcdSetText()** (p. **??**) 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

5.1.4.25 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()** (p. **??**) on a VEX LCD port. Use **lcdSetText()** (p. **??**) 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

5.1.4.26 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

5.1.4.27 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

5.1.4.28 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

5.1.4.29 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

5.1.4.30 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

5.1.4.31 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()** (p. **??**) to read

Returns

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

Referenced by **gyroGetA()**.

5.1.4.32 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()** (p. **??**) 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

Referenced by **init_main_gyro()**.

5.1.4.33 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()** (p. **??**) to reset

5.1.4.34 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()** (p. **??**) to stop

5.1.4.35 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 i2cReadRegister instead for the vast majority of applications.

5.1.4.36 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.

5.1.4.37 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.

5.1.4.38 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.

5.1.4.39 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

Referenced by **imeGetTicks()**.

5.1.4.40 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

Referenced by **imeGetVelocityA()**.

5.1.4.41 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()** (p. **??**). Do not use it in **initializeIO()** (p. **??**) 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.

Referenced by **imeInit()**, and **initialize()**.

5.1.4.42 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

Referenced by **imeInit()**, and **imeResetAll()**.

5.1.4.43 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.

5.1.4.44 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

5.1.4.45 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()** (p. **??**) 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

5.1.4.46 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.

5.1.4.47 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).

5.1.4.48 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()** (p. **??**) or **initializeIO()** (p. **??**) and can return false positives. It should be checked once and stored at the beginning of **operatorControl()** (p. **??**).

Parameters

joystick	the joystick slot to check

5.1.4.49 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()** (p. **??**) and **operatorControl()** (p. **??**) tasks is managed by the PROS kernel.

5.1.4.50 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

Referenced by **joystickGetAnalogA()**.

5.1.4.51 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

Referenced by **drive()**.

5.1.4.52 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

Referenced by **lcd_clear()**.

5.1.4.53 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

Referenced by **init_main_lcd()**.

5.1.4.54 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

Referenced by **lcd_get_pressed_buttons()**.

5.1.4.55 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

Referenced by **lcd_set_backlight()**.

5.1.4.56 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()** (p. **??**), 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

5.1.4.57 lcdShutdown()

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

Shut down the specified LCD port.

Parameters

lcdPort	the LCD to stop, either uart1 or uart2

5.1.4.58 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

5.1.4.59 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

5.1.4.60 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

5.1.4.61 motorSet()

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

Sets the speed of the specified motor channel.

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

Referenced by **drive()**, **flipperStop()**, **flywheelSet()**, **intakeStop()**, **robotDriveForward()**, **robotDriveReverse()**, and **robotStop()**.

5.1.4.62 motorStop()

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

Stops the motor on the specified channel, equivalent to calling **motorSet()** (p. **??**) 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()** (p. **??**) about use from multiple tasks.

Parameters

channel	the motor channel to stop from 1-10

5.1.4.63 motorStopAll()

```
void motorStopAll ( )
```

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

5.1.4.64 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()** (p. **??**) and **mutexGive()** (p. **??**) 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

5.1.4.65 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

5.1.4.66 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

5.1.4.67 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

5.1.4.68 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

5.1.4.69 powerLevelBackup()

```
unsigned int powerLevelBackup ( )
```

Returns the backup battery voltage in millivolts.

If no backup battery is connected, returns 0.

5.1.4.70 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.

5.1.4.71 print()

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

Prints the simple string to the debug terminal without formatting.

This method is much, much faster than **printf()** (p. **??**).

Parameters

string	the string to write

5.1.4.72 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()** (p. **??**)

Returns

the number of characters written

Referenced by **imeInit()**.

5.1.4.73 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

5.1.4.74 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

5.1.4.75 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()** (p. **??**) and **semaphoreGive()** (p. **??**) 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

5.1.4.76 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

5.1.4.77 semaphoreGive()

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

Signals a semaphore. Tasks waiting for a signal using **semaphoreTake()** (p. **??**) 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

5.1.4.78 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

5.1.4.79 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

5.1.4.80 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()** (p. **??**)

Returns

the number of characters stored

5.1.4.81 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()** (p. **??**) returns false.

5.1.4.82 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()** (p. **??**) returns false.

Parameters

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

5.1.4.83 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()** (p. **??**) returns false.

Parameters

song	the RTTTL song as a string value to play

5.1.4.84 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.

5.1.4.85 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()** (p. **??**) 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()** (p. **??**)

Returns

the number of characters stored

5.1.4.86 standaloneModeEnable()

```
void standaloneModeEnable ( )
```

Enables the Cortex to run the op control task in a standalone mode- no VEXnet connection required.

This function should only be called once in **initializeIO()** (p. **??**)

5.1.4.87 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

5.1.4.88 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()** (p. **??**).

Parameters

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

Referenced by **farRoutine()**, and **nearRoutine()**.

5.1.4.89 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 + cycleTime, 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()** (p. **??**) 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()** (p. **??**) specifies a wake time relative to the time at which the function is called, **taskDelayUntil()** (p. **??**) 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()** (p. **??**), 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

5.1.4.90 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

5.1.4.91 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

5.1.4.92 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

5.1.4.93 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

5.1.4.94 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

5.1.4.95 taskResume()

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

Resumes the specified task.

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

5.1.4.96 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()** (p. **??**) function is used for accurate cycle timing

Returns

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

5.1.4.97 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

5.1.4.98 ultrasonicGet()

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

Gets the current ultrasonic sensor value in centimeters.

If no object was found or if the ultrasonic sensor is polled while it is pinging and waiting for a response, -1 (ULTRA_B↩AD_RESPONSE) 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()** (p. **??**) to read

Returns

the distance to the nearest object in centimeters

5.1.4.99 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

5.1.4.100 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()** (p. **??**) to stop

5.1.4.101 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()** (p. **??**), **fprintf()** (p. **??**), and **fputc()** (p. **??**).

Re-initializing an open port may cause loss of data in the buffers. This routine may be safely called from **initializeIO()** (p. **??**) 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

5.1.4.102 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"

5.1.4.103 wait()

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

Alias of **taskDelay()** (p. **??**) intended to help EasyC users.

Parameters

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

5.1.4.104 waitUntil()

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

Alias of **taskDelayUntil()** (p. **??**) 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)

5.1.4.105 watchdogInit()

```
void watchdogInit ( )
```

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()** (p. **??**)

5.2 API.h

```
00001
00021 #ifndef API_H_
00022 #define API_H_
00023
00024 // System includes
00025 #include <stdlib.h>
00026 #include <stdbool.h>
00027 #include <stdarg.h>
00028 #include <stdint.h>
00029 #include <math.h>
00030 #include "stdbool.h"
00031
00032 // Begin C++ extern to C
00033 #ifdef __cplusplus
00034 extern "C" {
00035 #endif
00036
00037 // -------------------- VEX competition functions --------------------
00038
00042 #define JOY_DOWN 1
00043
00046 #define JOY_LEFT 2
00047
00050 #define JOY_UP 4
00051
00054 #define JOY_RIGHT 8
00055
00058 #define ACCEL_X 5
00059
00062 #define ACCEL_Y 6
00063
00070 bool isAutonomous();
00079 bool isEnabled();
00090 bool isJoystickConnected(unsigned char joystick);
00098 bool isOnline();
```

```
00106 int joystickGetAnalog(unsigned char joystick, unsigned char axis);
00116 bool joystickGetDigital(unsigned char joystick, unsigned char buttonGroup,
00117        unsigned char button);
00123 unsigned int powerLevelBackup();
00130 unsigned int powerLevelMain();
00136 void setTeamName(const char *name);
00137
00138 // --------------------- Pin control functions --------------------
00139
00143 #define BOARD_NR_ADC_PINS 8
00144
00153 #define BOARD_NR_GPIO_PINS 27
00154
00159 #define HIGH 1
00160
00165 #define LOW 0
00166
00174 #define INPUT 0x0A
00175
00181 #define INPUT_ANALOG 0x00
00182
00188 #define INPUT_FLOATING 0x04
00189
00195 #define OUTPUT 0x01
00196
00202 #define OUTPUT_OD 0x05
00203
00223 int analogCalibrate(unsigned char channel);
00239 int analogRead(unsigned char channel);
00254 int analogReadCalibrated(unsigned char channel);
00273 int analogReadCalibratedHR(unsigned char channel);
00289 bool digitalRead(unsigned char pin);
00300 void digitalWrite(unsigned char pin, bool value);
00311 void pinMode(unsigned char pin, unsigned char mode);
00312
00313 /*
00314  * Digital port 10 cannot be used as an interrupt port, or for an encoder. Plan accordingly.
00315  */
00316
00320 #define INTERRUPT_EDGE_RISING 1
00321
00324 #define INTERRUPT_EDGE_FALLING 2
00325
00329 #define INTERRUPT_EDGE_BOTH 3
00330
00334 typedef void (*InterruptHandler)(unsigned char pin);
00335
00343 void ioClearInterrupt(unsigned char pin);
00364 void ioSetInterrupt(unsigned char pin, unsigned char edges, InterruptHandler handler);
00365
00366 // --------------------- Physical output control functions --------------------
00367
00381 int motorGet(unsigned char channel);
00392 void motorSet(unsigned char channel, int speed);
00402 void motorStop(unsigned char channel);
00407 void motorStopAll();
00408
00416 void speakerInit();
00431 void speakerPlayArray(const char * * songs);
00445 void speakerPlayRtttl(const char *song);
00452 void speakerShutdown();
00453
00454 // --------------------- VEX sensor control functions --------------------
00455
00460 #define IME_ADDR_MAX 0x1F
00461
00481 unsigned int imeInitializeAll();
00502 bool imeGet(unsigned char address, int *value);
00525 bool imeGetVelocity(unsigned char address, int *value);
00534 bool imeReset(unsigned char address);
00542 void imeShutdown();
00543
00550 typedef void * Gyro;
00551
00561 int gyroGet(Gyro gyro);
00581 Gyro gyroInit(unsigned char port, unsigned short multiplier);
00590 void gyroReset(Gyro gyro);
00599 void gyroShutdown(Gyro gyro);
00600
00607 typedef void * Encoder;
00616 int encoderGet(Encoder enc);
```

```
00629 Encoder encoderInit(unsigned char portTop, unsigned char portBottom, bool reverse);
00638 void encoderReset(Encoder enc);
00647 void encoderShutdown(Encoder enc);
00648
00652 #define ULTRA_BAD_RESPONSE -1
00653
00660 typedef void * Ultrasonic;
00672 int ultrasonicGet(Ultrasonic ult);
00684 Ultrasonic ultrasonicInit(unsigned char portEcho, unsigned char portPing);
00693 void ultrasonicShutdown(Ultrasonic ult);
00694
00695 // -------------------- Custom sensor control functions --------------------
00696
00697 // ---- I2C port control ----
00708 bool i2cRead(uint8_t addr, uint8_t *data, uint16_t count);
00721 bool i2cReadRegister(uint8_t addr, uint8_t reg, uint8_t *value, uint16_t count);
00734 bool i2cWrite(uint8_t addr, uint8_t *data, uint16_t count);
00744 bool i2cWriteRegister(uint8_t addr, uint8_t reg, uint16_t value);
00745
00752 typedef int PROS_FILE;
00753
00754
00755 #ifndef FILE
00756
00761 #define FILE PROS_FILE
00762 #endif
00763
00767 #define SERIAL_DATABITS_8 0x0000
00768
00771 #define SERIAL_DATABITS_9 0x1000
00772
00775 #define SERIAL_STOPBITS_1 0x0000
00776
00779 #define SERIAL_STOPBITS_2 0x2000
00780
00783 #define SERIAL_PARITY_NONE 0x0000
00784
00787 #define SERIAL_PARITY_EVEN 0x0400
00788
00791 #define SERIAL_PARITY_ODD 0x0600
00792
00795 #define SERIAL_8N1 0x0000
00796
00813 void usartInit(PROS_FILE *usart, unsigned int baud, unsigned int flags);
00823 void usartShutdown(PROS_FILE *usart);
00824
00825 // -------------------- Character input and output --------------------
00826
00830 #define stdout ((PROS_FILE *)3)
00831
00834 #define stdin ((PROS_FILE *)3)
00835
00838 #define uart1 ((PROS_FILE *)1)
00839
00842 #define uart2 ((PROS_FILE *)2)
00843
00844 #ifndef EOF
00845
00848 #define EOF ((int)-1)
00849 #endif
00850
00851 #ifndef SEEK_SET
00852
00856 #define       SEEK_SET 0
00857 #endif
00858 #ifndef SEEK_CUR
00859
00863 #define       SEEK_CUR 1
00864 #endif
00865 #ifndef SEEK_END
00866
00870 #define       SEEK_END 2
00871 #endif
00872
00879 void fclose(PROS_FILE *stream);
00892 int fcount(PROS_FILE *stream);
00903 int fdelete(const char *file);
00911 int feof(PROS_FILE *stream);
00923 int fflush(PROS_FILE *stream);
00932 int fgetc(PROS_FILE *stream);
00947 char* fgets(char *str, int num, PROS_FILE *stream);
```

```
00971 PROS_FILE * fopen(const char *file, const char *mode);
00981 void fprint(const char *string, PROS_FILE *stream);
00991 int fputc(int value, PROS_FILE *stream);
01001 int fputs(const char *string, PROS_FILE *stream);
01013 size_t fread(void *ptr, size_t size, size_t count, PROS_FILE *stream);
01023 int fseek(PROS_FILE *stream, long int offset, int origin);
01031 long int ftell(PROS_FILE *stream);
01043 size_t fwrite(const void *ptr, size_t size, size_t count, PROS_FILE *stream);
01049 int getchar();
01057 void print(const char *string);
01068 int putchar(int value);
01075 int puts(const char *string);
01076
01106 int fprintf(PROS_FILE *stream, const char *formatString, ...);
01113 int printf(const char *formatString, ...);
01126 int snprintf(char *buffer, size_t limit, const char *formatString, ...);
01137 int sprintf(char *buffer, const char *formatString, ...);
01138
01142 #define LCD_BTN_LEFT 1
01143
01146 #define LCD_BTN_CENTER 2
01147
01150 #define LCD_BTN_RIGHT 4
01151
01160 void lcdClear(PROS_FILE *lcdPort);
01169 void lcdInit(PROS_FILE *lcdPort);
01181 #ifdef DOXYGEN
01182 void lcdPrint(PROS_FILE *lcdPort, unsigned char line, const char *formatString, ...);
01183 #else
01184 void __attribute__ ((format (printf, 3, 4))) lcdPrint(PROS_FILE *lcdPort, unsigned char line,
01185        const char *formatString, ...);
01186 #endif
01187
01196 unsigned int lcdReadButtons(PROS_FILE *lcdPort);
01205 void lcdSetBacklight(PROS_FILE *lcdPort, bool backlight);
01217 void lcdSetText(PROS_FILE *lcdPort, unsigned char line, const char *buffer);
01223 void lcdShutdown(PROS_FILE *lcdPort);
01224
01225 // -------------------- Real-time scheduler functions --------------------
01234 #define TASK_MAX 16
01235
01240 #define TASK_MAX_PRIORITIES 6
01241
01245 #define TASK_PRIORITY_LOWEST 0
01246
01251 #define TASK_PRIORITY_DEFAULT 2
01252
01256 #define TASK_PRIORITY_HIGHEST (TASK_MAX_PRIORITIES - 1)
01257
01264 #define TASK_DEFAULT_STACK_SIZE 512
01265
01272 #define TASK_MINIMAL_STACK_SIZE    64
01273
01277 #define TASK_DEAD 0
01278
01281 #define TASK_RUNNING 1
01282
01286 #define TASK_RUNNABLE 2
01287
01291 #define TASK_SLEEPING 3
01292
01295 #define TASK_SUSPENDED 4
01296
01302 typedef void * TaskHandle;
01308 typedef void * Mutex;
01314 typedef void * Semaphore;
01325 typedef void (*TaskCode)(void *);
01326
01338 TaskHandle taskCreate(TaskCode taskCode, const unsigned int stackDepth, void *parameters,
01339        const unsigned int priority);
01354 void taskDelay(const unsigned long msToDelay);
01375 void taskDelayUntil(unsigned long *previousWakeTime, const unsigned long cycleTime);
01389 void taskDelete(TaskHandle taskToDelete);
01399 unsigned int taskGetCount();
01411 unsigned int taskGetState(TaskHandle task);
01418 unsigned int taskPriorityGet(const TaskHandle task);
01429 void taskPrioritySet(TaskHandle task, const unsigned int newPriority);
01440 void taskResume(TaskHandle taskToResume);
01456 TaskHandle taskRunLoop(void (*fn)(void), const unsigned long increment);
01465 void taskSuspend(TaskHandle taskToSuspend);
01466
```

```
01479 Semaphore semaphoreCreate();
01491 bool semaphoreGive(Semaphore semaphore);
01501 bool semaphoreTake(Semaphore semaphore, const unsigned long blockTime);
01508 void semaphoreDelete(Semaphore semaphore);
01509
01522 Mutex mutexCreate();
01530 bool mutexGive(Mutex mutex);
01542 bool mutexTake(Mutex mutex, const unsigned long blockTime);
01549 void mutexDelete(Mutex mutex);
01550
01556 void delay(const unsigned long time);
01567 void delayMicroseconds(const unsigned long us);
01577 unsigned long micros();
01586 unsigned long millis();
01592 void wait(const unsigned long time);
01599 void waitUntil(unsigned long *previousWakeTime, const unsigned long time);
01607 void watchdogInit();
01613 void standaloneModeEnable();
01614
01615 // End C++ extern to C
01616 #ifdef __cplusplus
01617 }
01618 #endif
01619
01620 #endif
```

5.3 include/auton.h File Reference

Defines different pieces and functions for the autonomous.

```
#include "API.h"
```

Enumerations

- enum **AutonRoutine** { **far** = 0, **near** = 1, **none** = 2 }
  defines the different possible autonomous routines

Functions

- void **nearRoutine** ()
  begins the autonomous routine from the square nearest to the flag
- void **farRoutine** ()
  begins the autonomous routine from the square farthest from the flag

5.3.1 Detailed Description

Defines different pieces and functions for the autonomous.

Date

11/10/2018

Author

Michael Baraty

Definition in file **auton.h**.

5.3.2 Enumeration Type Documentation

5.3.2.1 AutonRoutine

`enum` **`AutonRoutine`**

defines the different possible autonomous routines

Date

11/10/2018

Author

Michael Baraty

Definition at line **18** of file **auton.h**.

```
00018             {
00019   far = 0,
00020   near = 1,
00021   none = 2
00022 } AutonRoutine;
```

5.3.3 Function Documentation

5.3.3.1 farRoutine()

`void farRoutine ( )`

begins the autonomous routine from the square farthest from the flag

Date

11/10/2018

Author

Michael Baraty

Definition at line **70** of file **auto.c**.

References **flywheelSet()**, **intakeSet()**, and **taskDelay()**.

```
00070                   {
00071   flywheelSet(120);
00072   taskDelay(4000);
00073   intakeSet(forward);
00074   taskDelay(1800);
00075   motorStopAll();
00076   }
```

5.3.3.2 nearRoutine()

```
void nearRoutine ( )
```

begins the autonomous routine from the square nearest to the flag

Date

11/10/2018

Author

Michael Baraty

Definition at line **44** of file **auto.c**.

References **flywheelSet()**, **intakeSet()**, and **taskDelay()**.

```
00044                   {
00045   //fire up the flywheels
00046   flywheelSet(70);
00047   taskDelay(4000);
00048   intakeSet(forward);
00049   taskDelay(1800);
00050   motorStopAll();
00051
00052   //drive for bottom flag
00053   motorSet(MOTOR_BACK_LEFT, MAX_SPEED / 2);
00054   motorSet(MOTOR_BACK_RIGHT, MIN_SPEED / 2);
00055   motorSet(MOTOR_FRONT_RIGHT, MIN_SPEED / 2 + 30);
00056   motorSet(MOTOR_FRONT_LEFT, MIN_SPEED / 2);
00057   taskDelay(2000);
00058   robotStop();
00059   taskDelay(500);
00060
00061   //prepare for tele-op
00062   motorSet(MOTOR_BACK_LEFT, -MAX_SPEED / 2);
00063   motorSet(MOTOR_BACK_RIGHT, -MIN_SPEED / 2);
00064   motorSet(MOTOR_FRONT_RIGHT, -MIN_SPEED / 2);
00065   motorSet(MOTOR_FRONT_LEFT, -MIN_SPEED / 2);
00066   taskDelay(1500);
00067   robotStop();
00068 }
```

5.4 auton.h

```
00001
00008 #ifndef _AUTON_H_
00009 #define _AUTON_H_
00010
00011 #include "API.h"
00012
00018 typedef enum {
00019   far = 0,
00020   near = 1,
00021   none = 2
00022 } AutonRoutine;
00023
00029 static AutonRoutine routine = none;
00030
00031
00037 void nearRoutine();
00038
00044 void farRoutine();
00045
00046
00047
00048 #endif
```

5.5 include/controller.h File Reference

Controller macro declarations.

Macros

- #define **JOYSTICK_MAIN** 1
  The main joystick.
- #define **JOYSTICK_PARTNER** 2
  The paartner joystick.
- #define **JOYSTICK_RIGHT_X** 1
  The x-axis on the right joystick.
- #define **JOYSTICK_RIGHT_Y** 2
  The y-axis on the right joystick.
- #define **JOYSTICK_LEFT_X** 4
  The x-axis on the left joystick.
- #define **JOYSTICK_LEFT_Y** 3
  The y-axis on the left joystick.

Functions

- int **joystickGetAnalogA** (unsigned char joystick, unsigned char channel)
  reads the joysticks leaving a tolerance for a given THRESHOLD

5.5.1 Detailed Description

Controller macro declarations.

Author

Michael Baraty

Date

1026/2018

Definition in file **controller.h**.

5.5.2 Macro Definition Documentation

5.5.2.1 JOYSTICK_LEFT_X

```
#define JOYSTICK_LEFT_X 4
```

The x-axis on the left joystick.

Author

Michael Baraty

Date

10/26/2018

Definition at line **43** of file **controller.h**.

Referenced by **drive()**.

5.5.2.2 JOYSTICK_LEFT_Y

```
#define JOYSTICK_LEFT_Y 3
```

The y-axis on the left joystick.

Author

Michael Baraty

Date

10/26/2018

Definition at line **50** of file **controller.h**.

Referenced by **drive()**.

5.5.2.3 JOYSTICK_MAIN

```
#define JOYSTICK_MAIN 1
```

The main joystick.

Author

Michael Baraty

Date

10/26/2018

Definition at line **15** of file **controller.h**.

Referenced by **drive()**.

5.5.2.4 JOYSTICK_PARTNER

```
#define JOYSTICK_PARTNER 2
```

The paartner joystick.

Author

Michael Baraty

Date

10/26/2018

Definition at line **22** of file **controller.h**.

5.5.2.5 JOYSTICK_RIGHT_X

```
#define JOYSTICK_RIGHT_X 1
```

The x-axis on the right joystick.

Author

Michael Baraty

Date

10/26/2018

Definition at line **29** of file **controller.h**.

Referenced by **drive()**.

5.5.2.6 JOYSTICK_RIGHT_Y

```
#define JOYSTICK_RIGHT_Y 2
```

The y-axis on the right joystick.

Author

Michael Baraty

Date

10/26/2018

Definition at line **36** of file **controller.h**.

5.5.3 Function Documentation

5.5.3.1 joystickGetAnalogA()

```
int joystickGetAnalogA (
            unsigned char joystick,
            unsigned char channel )
```

reads the joysticks leaving a tolerance for a given THRESHOLD

Parameters

joystick	
channel	

Author

Michael Baraty

Date

10/26/2018

Definition at line **5** of file **controller.c**.

References **joystickGetAnalog()**, and **THRESHOLD**.

Referenced by **drive()**.

```
00005                                                                    {
00006   return abs(joystickGetAnalog(joystick, channel)) > THRESHOLD?
      joystickGetAnalog(joystick, channel): 0;
00007 }
```

5.6 controller.h

```
00001
00007 #ifndef _CONTROLLER_H_
00008 #define _CONTROLLER_H_
00009
00015 #define JOYSTICK_MAIN 1
00016
00022 #define JOYSTICK_PARTNER 2
00023
00029 #define JOYSTICK_RIGHT_X 1
00030
00036 #define JOYSTICK_RIGHT_Y 2
00037
00043 #define JOYSTICK_LEFT_X 4
00044
00050 #define JOYSTICK_LEFT_Y 3
00051
00059 int joystickGetAnalogA(unsigned char joystick, unsigned char channel);
00060
00061
00062
00063 #endif
```

5.7 include/drive.h File Reference

Basic drive definitions and functions.

```
#include <API.h>
#include "motor_ports.h"
#include "controller.h"
#include "gyro.h"
```

Macros

- #define **THRESHOLD** 20

  defines the minimum reading from the joysticks to avoid underpowering the motors

Enumerations

- enum **Direction** { **forward**, **reverse**, **clockwise**, **counterclockwise** }

  defines the directions in which the drive base can be moving
- enum **FlipperDirection** { **up**, **down** }

  defines the directions in which the intake mechanism can move

Functions

- void **motorSpeedStick** (unsigned char channel, int speed)

  sets the speed of the motors according to a joystick taking into account the threshold
- void **robotStop** ()

  stops the robot drivebase
- void **flywheelSet** (int speed)

  sets the speed for the flywheel
- void **flywheelStop** ()

  stops the flywheel
- void **intakeSet** (**Direction** dir)

  activates the intake mechanism at a certain speed
- void **intakeStop** ()

  disables the intake mechanism
- void **drive** ()

  drives the robot
- void **robotSpin** (**Direction** dir, int speed)

  spins the robot in a given direction at a given speed
- void **robotStraighten** ()

  straightens the robot to face the starting direction
- void **robotDriveForward** ()

  drives the robot forward at half speed
- void **robotDriveStraight** (**Direction** dir)

  drives the robot straight in a given direction
- void **robotDriveReverse** ()

  drives the robot backwards at half speed
- void **flipperMove** (**FlipperDirection** dir)

  moves the flipper in a certain direction
- void **flipperStop** ()

  stops the cap flipper

5.7.1 Detailed Description

Basic drive definitions and functions.

Definition in file **drive.h**.

5.7.2 Macro Definition Documentation

5.7.2.1 THRESHOLD

```
#define THRESHOLD 20
```

defines the minimum reading from the joysticks to avoid underpowering the motors

Author

Michael Baraty

Date

10/25/2018

Definition at line **42** of file **drive.h**.

Referenced by **joystickGetAnalogA()**.

5.7.3 Enumeration Type Documentation

5.7.3.1 Direction

```
enum Direction
```

defines the directions in which the drive base can be moving

Author

Michael Baraty

Date

10/31/2018

Definition at line **20** of file **drive.h**.

```
00020             {
00021   forward,
00022   reverse,
00023   clockwise,
00024   counterclockwise
00025 } Direction;
```

5.7.3.2 FlipperDirection

```
enum FlipperDirection
```

defines the directions in which the intake mechanism can move

Author

Michael Baraty

Date

11/10/2018

Definition at line **32** of file **drive.h**.

```
00032             {
00033   up,
00034   down
00035 } FlipperDirection;
```

5.7.4 Function Documentation

5.7.4.1 drive()

```
void drive ( )
```

drives the robot

Author

Michael Baraty

Date

10/26/2018

Definition at line **41** of file **drive.c**.

References **intakeSet()**, **JOY_UP**, **JOYSTICK_LEFT_X**, **JOYSTICK_LEFT_Y**, **JOYSTICK_MAIN**, **JOYSTICK_↩ RIGHT_X**, **joystickGetAnalogA()**, **joystickGetDigital()**, **MOTOR_BACK_LEFT**, **MOTOR_BACK_RIGHT**, **MOT↩ OR_FRONT_LEFT**, **MOTOR_FRONT_RIGHT**, and **motorSet()**.

Referenced by **operatorControl()**.

```
00041             {
00042
00043
00044   int stickLX = joystickGetAnalogA(JOYSTICK_MAIN, JOYSTICK_LEFT_X);
00045   int stickLY = joystickGetAnalogA(JOYSTICK_MAIN, JOYSTICK_LEFT_Y);
00046   int stickRX = joystickGetAnalogA(JOYSTICK_MAIN, JOYSTICK_RIGHT_X);
00047
00048   motorSet(MOTOR_FRONT_RIGHT, -(stickLY - stickRX - stickLX) * .99);
00049   motorSet(MOTOR_BACK_RIGHT, -(stickLY - stickRX + stickLX) * .99);
00050   motorSet(MOTOR_FRONT_LEFT, -(stickLY + stickRX + stickLX) * .99);
00051   motorSet(MOTOR_BACK_LEFT, (stickLY + stickRX - stickLX) * .99);
00052
00053   if(joystickGetDigital(JOYSTICK_MAIN, 6, JOY_UP)) {
00054     intakeSet(forward);
00055   } else if(joystickGetDigital(JOYSTICK_MAIN, 6, JOY_DOWN)) {
00056     intakeSet(reverse);
00057   } else {
00058     intakeStop();
00059   }
00060
00061   if(joystickGetDigital(JOYSTICK_MAIN, 5, JOY_UP)) {
00062     firingMode = true;
00063   }
00064   if(joystickGetDigital(JOYSTICK_MAIN, 5, JOY_DOWN)) {
00065     firingMode = false;
00066   }
00067
00068   if(firingMode && joystickGetDigital(JOYSTICK_MAIN, 7, JOY_LEFT)) {
00069     firingSpeed = true;
00070   } else if(firingMode && joystickGetDigital(JOYSTICK_MAIN, 7, JOY_DOWN)) {
00071     firingSpeed = false;
00072   }
00073
00074   if(firingMode && firingSpeed) {
00075     flywheelSet(110);
00076   } else if(firingMode && !firingSpeed) {
00077     flywheelSet(60);
00078   } else {
00079     flywheelSet(motorGet(MOTOR_FLYWHEEL_A) / 3 >= 1? motorGet(MOTOR_FLYWHEEL_A) / 3: 0);
00080   }
00081
00082 /*  if(joystickGetDigital(JOYSTICK_MAIN, 8, JOY_DOWN)) {
00083     robotStraighten();
00084   }*/
00085
00086   if(joystickGetDigital(JOYSTICK_MAIN, 8, JOY_UP)) {
00087     flipperMove(up);
00088   }  else if(joystickGetDigital(JOYSTICK_MAIN, 8, JOY_LEFT)) {
00089     flipperMove(down);
00090   } else {
00091     flipperStop();
00092   }
00093
00094 }
```

5.7.4.2 flipperMove()

```
void flipperMove (
              FlipperDirection dir )
```

moves the flipper in a certain direction

Parameters

dir	

Author

Michael Baraty

Date

11/6/2018

Definition at line **140** of file **drive.c**.

```
00140                                       {
00141   if(dir == up) {
00142     motorSet(MOTOR_FLIPPER, MIN_SPEED);
00143   } else if(dir == down) {
00144     motorSet(MOTOR_FLIPPER, MAX_SPEED);
00145   }
00146 }
```

5.7.4.3 flipperStop()

```
void flipperStop ( )
```

stops the cap flipper

Author

Michael Baraty

Date

11/10/2018

Definition at line **148** of file **drive.c**.

References **MOTOR_FLIPPER**, and **motorSet()**.

```
00148                     {
00149   motorSet(MOTOR_FLIPPER, 0);
00150 }
```

5.7.4.4 flywheelSet()

```
void flywheelSet (
              int speed )
```

sets the speed for the flywheel

Parameters

speed	

Author

Michael Baraty

Date

10/26/2018

Definition at line **14** of file **drive.c**.

References **MOTOR_FLYWHEEL_A**, **MOTOR_FLYWHEEL_B**, and **motorSet()**.

Referenced by **farRoutine()**, **flywheelStop()**, and **nearRoutine()**.

```
00014                             {
00015   motorSet(MOTOR_FLYWHEEL_A, -speed);
00016   motorSet(MOTOR_FLYWHEEL_B, -speed);
00017 }
```

5.7.4.5 flywheelStop()

```
void flywheelStop ( )
```

stops the flywheel

Author

Michael Baraty

Date

10/26/2018

Definition at line **19** of file **drive.c**.

References **flywheelSet()**.

```
00019                       {
00020   flywheelSet(0);
00021 }
```

5.7.4.6 intakeSet()

```
void intakeSet (
              Direction dir )
```

activates the intake mechanism at a certain speed

Author

Michael Baraty

Date

10/31/2018

Definition at line **23** of file **drive.c**.

Referenced by **drive()**, **farRoutine()**, and **nearRoutine()**.

```
00023                                {
00024   if(dir == forward) {
00025     motorSet(MOTOR_INTAKE, MIN_SPEED);
00026     motorSet(MOTOR_BELT, MAX_SPEED);
00027   } else if (dir == reverse) {
00028     motorSet(MOTOR_INTAKE, MAX_SPEED);
00029     motorSet(MOTOR_BELT, MIN_SPEED);
00030   } else {
00031     motorSet(MOTOR_INTAKE, 0);
00032     motorSet(MOTOR_BELT, 0);
00033   }
00034 }
```

5.7.4.7 intakeStop()

```
void intakeStop ( )
```

disables the intake mechanism

Author

Michael Baraty

Date

10/26/2018

Definition at line **36** of file **drive.c**.

References **MOTOR_BELT**, **MOTOR_INTAKE**, and **motorSet()**.

```
00036                    {
00037   motorSet(MOTOR_INTAKE, 0);
00038   motorSet(MOTOR_BELT, 0);
00039 }
```

5.7.4.8 motorSpeedStick()

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

sets the speed of the motors according to a joystick taking into account the threshold

Parameters

channel	
speed	

Author

Michael Baraty

Date

10/25/2018

5.7.4.9 robotDriveForward()

```
void robotDriveForward ( )
```

drives the robot forward at half speed

Author

Michael Baraty

Date

11/6/2018

Definition at line **116** of file **drive.c**.

References **MAX_SPEED**, **MIN_SPEED**, **MOTOR_BACK_LEFT**, **MOTOR_BACK_RIGHT**, **MOTOR_FRONT_LEFT**, **MOTOR_FRONT_RIGHT**, and **motorSet()**.

```
00116                           {
00117   motorSet(MOTOR_BACK_LEFT, MAX_SPEED / 2);
00118   motorSet(MOTOR_BACK_RIGHT, MIN_SPEED / 2);
00119   motorSet(MOTOR_FRONT_RIGHT, MIN_SPEED / 2);
00120   motorSet(MOTOR_FRONT_LEFT, MIN_SPEED / 2);
00121 }
```

5.7.4.10 robotDriveReverse()

```
void robotDriveReverse ( )
```

drives the robot backwards at half speed

Author

Michael Baraty

Date

11/6/2018

Definition at line **123** of file **drive.c**.

References **MAX_SPEED**, **MIN_SPEED**, **MOTOR_BACK_LEFT**, **MOTOR_BACK_RIGHT**, **MOTOR_FRONT_LEFT**, **MOTOR_FRONT_RIGHT**, and **motorSet()**.

```
00123                           {
00124   motorSet(MOTOR_BACK_LEFT, MIN_SPEED / 2);
00125   motorSet(MOTOR_BACK_RIGHT, MAX_SPEED / 2);
00126   motorSet(MOTOR_FRONT_RIGHT, MAX_SPEED / 2);
00127   motorSet(MOTOR_FRONT_LEFT, MAX_SPEED / 2);
00128 }
```

5.7.4.11 robotDriveStraight()

```
void robotDriveStraight (
              Direction dir )
```

drives the robot straight in a given direction

Parameters

dir	

Author

Michael Baraty

Date

11/6/2018

Definition at line **130** of file **drive.c**.

```
00130                                       {
00131   if(dir == forward) {
00132     robotDriveForward();
00133   } else if(dir == reverse) {
00134     robotDriveReverse();
00135   } else {
00136     robotStop();
00137   }
00138 }
```

5.7.4.12 robotSpin()

```
void robotSpin (
              Direction dir,
             int speed )
```

spins the robot in a given direction at a given speed

Parameters

dir	
speed	

Author

Michael Baraty

Date

11/6/2018

Definition at line **96** of file **drive.c**.

Referenced by **gyroTurn()**.

```
00096                                         {
00097   if(dir == clockwise) {
00098     motorSet(MOTOR_FRONT_RIGHT, -speed);
00099     motorSet(MOTOR_FRONT_LEFT, speed);
00100     motorSet(MOTOR_BACK_RIGHT, -speed);
00101     motorSet(MOTOR_FRONT_LEFT, speed);
00102   } else if(dir == counterclockwise) {
00103     motorSet(MOTOR_FRONT_RIGHT, speed);
00104     motorSet(MOTOR_FRONT_LEFT, -speed);
00105     motorSet(MOTOR_BACK_RIGHT, speed);
00106     motorSet(MOTOR_FRONT_LEFT, -speed);
00107   } else {
00108     robotStop();
00109   }
00110 }
```

5.7.4.13 robotStop()

```
void robotStop ( )
```

stops the robot drivebase

Author

Michael Baraty

Date

10/26/2018

Definition at line **7** of file **drive.c**.

References **MOTOR_BACK_LEFT**, **MOTOR_BACK_RIGHT**, **MOTOR_FRONT_LEFT**, **MOTOR_FRONT_RIGHT**, and **motorSet()**.

```
00007                 {
00008   motorSet(MOTOR_BACK_LEFT, 0);
00009   motorSet(MOTOR_BACK_RIGHT, 0);
00010   motorSet(MOTOR_FRONT_RIGHT, 0);
00011   motorSet(MOTOR_FRONT_LEFT, 0);
00012 }
```

5.7.4.14 robotStraighten()

```
void robotStraighten ( )
```

straightens the robot to face the starting direction

Author

Michael Baraty

Date

11/6/2018

Definition at line **112** of file **drive.c**.

References **gyroTurn()**.

```
00112                       {
00113   gyroTurn(0);
00114 }
```

5.8 drive.h

```
00001
00005  #ifndef _DRIVE_H_
00006  #define _DRIVE_H_
00007
00008  #include <API.h>
00009  #include "motor_ports.h"
00010  #include "controller.h"
00011  #include "gyro.h"
00012
00013
00014
00020 typedef enum {
00021   forward,
00022   reverse,
00023   clockwise,
00024   counterclockwise
00025 } Direction;
00026
00032 typedef enum {
00033   up,
00034   down
00035 } FlipperDirection;
00036
00042 #define THRESHOLD 20
00043
00051 void motorSpeedStick(unsigned char channel, int speed);
00052
00058 void robotStop();
00059
00066 void flywheelSet(int speed);
00067
00073 void flywheelStop();
00074
00080 void intakeSet(Direction dir);
00081
00087 void intakeStop();
00088
00094 void drive();
00095
00103 void robotSpin(Direction dir, int speed);
00104
00110 void robotStraighten();
00111
00117 void robotDriveForward();
00118
00125 void robotDriveStraight(Direction dir);
00126
00132 void robotDriveReverse();
00133
00140 void flipperMove(FlipperDirection dir);
00141
00147 void flipperStop();
00148
00149 #endif
```

5.9 include/encoders.h File Reference

wrapper around encoder functions

```
#include "API.h"
#include "drive.h"
```

Macros

- #define **IME_NUMBER** 4

The number of IMEs. This number is compared against the number detect in init_encoders.

- #define **ENCODER_TILE** (392 / 4 ∗ 3.14 ∗ 12)

  defines the distance the robot needs to drive in encoder ticks
- #define **IME_FRONT_LEFT** 1

  defines the front left IME in the order
- #define **IME_FRONT_RIGHT** 2

  defines the front right IME in the order
- #define **IME_BACK_LEFT** 0

  defines the back left IME in the order
- #define **IME_BACK_RIGHT** 3

  defines the back right IME in the order

Functions

- bool **imeInit** ()

  Initializes all motor encoders.
- int **imeGetTicks** (const unsigned char address)

  Gets the encoder ticks since last reset.
- int **imeGetVelocityA** (const unsigned char address)

  Gets the encoder velocity.
- void **imeMoveFor** (int ticks, **Direction** dir)

  moves at a given velocity for a certain number of ticks
- int **imeGetAverageTicks** ()

  Gets the average encoder ticks since the last reset.
- void **imeResetAll** ()

  Resets all the encoders.

5.9.1 Detailed Description

wrapper around encoder functions

Author

Michael Baraty

Date

11/2/2018

Definition in file **encoders.h**.

5.9.2 Macro Definition Documentation

5.9.2.1 ENCODER_TILE

```
#define ENCODER_TILE (392 / 4 * 3.14 * 12)
```

defines the distance the robot needs to drive in encoder ticks

Author

Michael Baraty

Date

11/10/2018

Definition at line **28** of file **encoders.h**.

5.9.2.2 IME_BACK_LEFT

```
#define IME_BACK_LEFT 0
```

defines the back left IME in the order

Author

Michael Baraty

Date

11/10/2018

Definition at line **49** of file **encoders.h**.

Referenced by **imeGetAverageTicks()**, **imeInit()**, and **imeResetAll()**.

5.9.2.3 IME_BACK_RIGHT

```
#define IME_BACK_RIGHT 3
```

defines the back right IME in the order

Author

Michael Baraty

Date

11/10/2018

Definition at line **56** of file **encoders.h**.

Referenced by **imeInit()**, and **imeResetAll()**.

5.9.2.4 IME_FRONT_LEFT

```
#define IME_FRONT_LEFT 1
```

defines the front left IME in the order

Author

Michael Baraty

Date

11/10/2018

Definition at line **35** of file **encoders.h**.

Referenced by **imeGetAverageTicks()**, **imeInit()**, and **imeResetAll()**.

5.9.2.5 IME_FRONT_RIGHT

```
#define IME_FRONT_RIGHT 2
```

defines the front right IME in the order

Author

Michael Baraty

Date

11/10/2018

Definition at line **42** of file **encoders.h**.

Referenced by **imeGetAverageTicks()**, **imeInit()**, and **imeResetAll()**.

5.9.2.6 IME_NUMBER

```
#define IME_NUMBER 4
```

The number of IMEs. This number is compared against the number detect in init_encoders.

See also

init_encoders()

Author

Michael Baraty

Date

10/25/2018

See also

IME_NUMBER (p. **??**)

Definition at line **21** of file **encoders.h**.

Referenced by **imeInit()**.

5.9.3 Function Documentation

5.9.3.1 imeGetAverageTicks()

```
int imeGetAverageTicks ( )
```

Gets the average encoder ticks since the last reset.

Author

Michael Baraty

Date

11/8/2018

Definition at line **48** of file **encoders.c**.

References **IME_BACK_LEFT**, **IME_FRONT_LEFT**, **IME_FRONT_RIGHT**, and **imeGetTicks()**.

```
00048                           {
00049   return ((imeGetTicks(IME_FRONT_LEFT) + imeGetTicks(IME_FRONT_RIGHT) +
00050     imeGetTicks(IME_BACK_LEFT) + imeGetTicks(IME_FRONT_RIGHT)) / 4);
00051 }
```

5.9.3.2 imeGetTicks()

```
int imeGetTicks (
            const unsigned char address )
```

Gets the encoder ticks since last reset.

Parameters

address	

Author

Michael Baraty

Date

10/25/2018

Definition at line **21** of file **encoders.c**.

References **imeGet()**.

Referenced by **imeGetAverageTicks()**.

```
00021                                           {
00022   int i = 0;
00023   imeGet(address, &i);
00024   return i;
00025 }
```

5.9.3.3 imeGetVelocityA()

```
int imeGetVelocityA (
            const unsigned char address )
```

Gets the encoder velocity.

Returns

velocity

Parameters

address	

Author

Michael Baraty

Date

11/8/2018

Definition at line **27** of file **encoders.c**.

References **imeGetVelocity()**.

```
00027                                                   {
00028   int i = 0;
00029   imeGetVelocity(address, &i);
00030   return i;
00031 }
```

5.9.3.4 imeInit()

```
bool imeInit ( )
```

Initializes all motor encoders.

Author

Michael Baraty

Date

10/25/2018

See also

IME_NUMBER (p. **??**)

Definition at line **3** of file **encoders.c**.

References **IME_BACK_LEFT**, **IME_BACK_RIGHT**, **IME_FRONT_LEFT**, **IME_FRONT_RIGHT**, **IME_NUMBER**, **imeInitializeAll()**, **imeReset()**, and **printf()**.

```
00003               {
00004   #ifdef IME_NUMBER
00005     int count = imeInitializeAll();
00006     if(count != IME_NUMBER) {
00007       printf("detected only %d\n", count);
00008       imeReset(0);
00009       return false;
00010     }
00011     imeReset(IME_FRONT_LEFT);
00012     imeReset(IME_FRONT_RIGHT);
00013     imeReset(IME_BACK_LEFT);
00014     imeReset(IME_BACK_RIGHT);
00015     return true;
00016   #else
00017     return imeInitializeAll()
00018   #endif
00019 }
```

5.9.3.5 imeMoveFor()

```
void imeMoveFor (
              int ticks,
               Direction dir )
```

moves at a given velocity for a certain number of ticks

Parameters

ticks	
dir	

Author

Michael Baraty

Date

11/8/2018

Definition at line **33** of file **encoders.c**.

```
00033                                           {
00034   if(dir == forward) {
00035     while(true) {
00036       if(imeGetAverageTicks() <= ticks) {
00037         robotDriveStraight(forward);
00038       }
00039   }
00040 } else if(dir == reverse) {
00041     while(imeGetAverageTicks() >= ticks){
00042       robotDriveStraight(reverse);
00043     }
00044   }
00045 }
```

5.9.3.6 imeResetAll()

```
void imeResetAll ( )
```

Resets all the encoders.

Author

Michael Baraty

Date

11/8/2018

Definition at line **53** of file **encoders.c**.

References **IME_BACK_LEFT**, **IME_BACK_RIGHT**, **IME_FRONT_LEFT**, **IME_FRONT_RIGHT**, and **imeReset()**.

```
00053                   {
00054   imeReset(IME_FRONT_LEFT);
00055   imeReset(IME_FRONT_RIGHT);
00056   imeReset(IME_BACK_LEFT);
00057   imeReset(IME_BACK_RIGHT);
00058 }
```

5.10 encoders.h

```
00001
00007 #ifndef _ENCODERS_H_
00008 #define _ENCODERS_H_
00009
00010 #include "API.h"
00011 #include "drive.h"
00012
00021 #define IME_NUMBER 4
00022
00028 #define ENCODER_TILE (392 / 4 * 3.14 * 12)
00029
00035 #define IME_FRONT_LEFT 1
00036
00042 #define IME_FRONT_RIGHT 2
00043
00049 #define IME_BACK_LEFT 0
00050
00056 #define IME_BACK_RIGHT 3
00057
00064 bool imeInit();
00065
00072 int imeGetTicks(const unsigned char address);
00073
00081 int imeGetVelocityA(const unsigned char address);
00082
00090 void imeMoveFor(int ticks, Direction dir);
00091
00097 int imeGetAverageTicks();
00098
00104 void imeResetAll();
00105
00106 #endif
```

5.11 include/gyro.h File Reference

Declarations and macros for the VEX gyroscope.

```
#include "API.h"
#include "drive.h"
```

Macros

- #define **GYRO_PORT** 1

  Port the Gyroscope plugs into.
- #define **GYRO_MULTIPLIER** 196
- #define **GYRO_TURN_SPEED_MAX** 80
- #define **GYRO_TURN_SPEED_MIN_NORMAL** 33
- #define **GYRO_TURN_SPEED_MIN_FAST** 45

Functions

- int **gyroGetA** ()

  returns the main gyroscope reading
- bool **init_main_gyro** ()

  Initializes the main robot gryoscope/ Only call function when robot still and ready to start autonomous. Robot should not move for five seconds while Gyro calibrates.
- void **gyroTurn** (int degrees)

  Spins the robot to face a certain degree.

5.11.1 Detailed Description

Declarations and macros for the VEX gyroscope.

Author

Michael Baraty

Date

10/31/18

Definition in file **gyro.h**.

5.11.2 Macro Definition Documentation

5.11.2.1 GYRO_MULTIPLIER

```
#define GYRO_MULTIPLIER 196
```

Special Gyro multiplier for tuning in Gyro

Todo

Definition at line **32** of file **gyro.h**.

Referenced by **init_main_gyro()**.

5.11.2.2 GYRO_PORT

```
#define GYRO_PORT 1
```

Port the Gyroscope plugs into.

Author

Michael Baraty

Date

11/10/2018

Definition at line **18** of file **gyro.h**.

Referenced by **init_main_gyro()**.

5.11.3 Function Documentation

5.11.3.1 gyroGetA()

```
int gyroGetA ( )
```

returns the main gyroscope reading

Returns

gyro direction

Author

Michael Baraty

Date

11/10/2018

Definition at line **10** of file **gyro.c**.

References **gyroGet()**.

Referenced by **gyroTurn()**.

```
00010                {
00011   return gyroGet(main_gyro);
00012 }
```

5.11.3.2 gyroTurn()

```
void gyroTurn (
              int degrees )
```

Spins the robot to face a certain degree.

Author

Michael Baraty

Date

10/31/18

Definition at line **14** of file **gyro.c**.

References **gyroGetA()**, and **robotSpin()**.

Referenced by **robotStraighten()**.

```
00014                           {
00015   if(gyroGetA() >= abs(degree - 180)) {
00016     while(gyroGetA() != degree) {
00017       robotSpin(counterclockwise, 70);
00018     }
00019   } else {
00020     while(gyroGetA() != degree) {
00021       robotSpin(clockwise, 70);
00022     }
00023   }
00024 }
```

5.11.3.3 init_main_gyro()

```
bool init_main_gyro ( )
```

Initializes the main robot gryoscope/ Only call function when robot still and ready to start autonomous. Robot should not move for five seconds while Gyro calibrates.

Returns

if the Gyro was successfully calibrated

Author

Michael Baraty

Date

10/31/18

Definition at line **5** of file **gyro.c**.

References **GYRO_MULTIPLIER**, **GYRO_PORT**, and **gyroInit()**.

Referenced by **initialize()**.

```
00005                       {
00006   main_gyro = gyroInit(GYRO_PORT, GYRO_MULTIPLIER);
00007   return main_gyro != NULL;
00008 }
```

5.12 gyro.h

```
00001
00007 #ifndef _GYRO_H_
00008 #define _GYRO_H_
00009
00010 #include "API.h"
00011 #include "drive.h"
00012
00018 #define GYRO_PORT 1
00019
00026 int gyroGetA();
00027
00032 #define GYRO_MULTIPLIER 196
00033 #define GYRO_TURN_SPEED_MAX 80
00034 #define GYRO_TURN_SPEED_MIN_NORMAL 33
00035 #define GYRO_TURN_SPEED_MIN_FAST 45
00036
00045 bool init_main_gyro();
00046
00052 void gyroTurn(int degrees);
00053
00054 #endif
```

5.13 include/lcd.h File Reference

LCD wrapper functions and macros.

```
#include <API.h>
```

Data Structures

- struct **lcd_buttons**

  represents the state of the lcd buttons

Macros

- #define **TOP_ROW** 1

  The top row on the lcd screen.
- #define **BOTTOM_ROW** 2

  The bottom row on the lcd screen.

Enumerations

- enum **button_state** { **RELEASED** = false, **PRESSED** = true }

  Represents the state of a button.

Functions

- **lcd_buttons lcd_get_pressed_buttons** ()

  Returns the pressed buttons.
- void **lcd_clear** ()

  Clears the lcd.
- void **init_main_lcd** (**FILE** *lcd)

  Initializes the lcd screen. Also will initialize the lcd_port var. Must be called before any lcd function can be called.
- void **lcd_print** (unsigned int line, const char *str)

  prints a string to a line on the lcd
- void **lcd_printf** (unsigned int line, const char *format_str,...)

  prints a formatted string to a line on the lcd. Smilar to printf
- void **lcd_set_backlight** (bool state)

  sets the backlight of the lcd
- void **promt_confirmation** (const char *confirm_text)

  Prompts the user to confirm a string. User must press middle button to confirm. Function is not thread safe and will stall a thread.

5.13.1 Detailed Description

LCD wrapper functions and macros.

Author

Chris Jerrett

Date

9/9/2017

Definition in file **lcd.h**.

5.13.2 Macro Definition Documentation

5.13.2.1 BOTTOM_ROW

```
#define BOTTOM_ROW 2
```

The bottom row on the lcd screen.

Author

Chris Jerrett

Date

9/9/2017

Definition at line **25** of file **lcd.h**.

5.13.2.2 TOP_ROW

```
#define TOP_ROW 1
```

The top row on the lcd screen.

Author

Chris Jerrett

Date

9/9/2017

Definition at line **18** of file **lcd.h**.

5.13.3 Enumeration Type Documentation

5.13.3.1 button_state

```
enum button_state
```

Represents the state of a button.

A button can be pressed of RELEASED. Release is false which is also 0. PRESSED is true or 1.

Author

Chris Jerrett

Date

9/9/2017

Enumerator

RELEASED	A released button
PRESSED	A pressed button

Definition at line **36** of file **lcd.h**.

```
00036             {
00038   RELEASED = false,
00040   PRESSED = true,
00041 } button_state;
```

5.13.4 Function Documentation

5.13.4.1 init_main_lcd()

```
void init_main_lcd (
              FILE * lcd )
```

Initializes the lcd screen. Also will initialize the lcd_port var. Must be called before any lcd function can be called.

Parameters

lcd	the urart port of the lcd screen

See also

uart1 (p. **??**)
uart2 (p. **??**)

Author

Chris Jerrett

Date

9/9/2017

Definition at line **33** of file **lcd.c**.

References **lcd_clear()**, and **lcdInit()**.

```
00033                                {
00034   lcd_port = lcd;
00035   lcdInit(lcd);
00036   lcd_clear();
00037 }
```

5.13.4.2 lcd_clear()

```
void lcd_clear ( )
```

Clears the lcd.

Author

Chris Jerrett

Date

9/9/2017

Definition at line **28** of file **lcd.c**.

References **lcdClear()**.

Referenced by **init_main_lcd()**.

```
00028                {
00029   lcd_assert();
00030   lcdClear(lcd_port);
00031 }
```

5.13.4.3 lcd_get_pressed_buttons()

```
lcd_buttons lcd_get_pressed_buttons ( )
```

Returns the pressed buttons.

Returns

a struct containing the states of all three buttons.

Author

Chris Jerrett

Date

9/9/2017

See also

lcd_buttons (p. **??**)

Definition at line **14** of file **lcd.c**.

References **lcdReadButtons()**, **PRESSED**, and **RELEASED**.

Referenced by **promt_confirmation()**.

```
00014                                       {
00015   lcd_assert();
00016   unsigned int btn_binary = lcdReadButtons(lcd_port);
00017   bool left = btn_binary & 0x1;   // 0001
00018   bool middle = btn_binary & 0x2; // 0010
00019   bool right = btn_binary & 0x4;  // 0100
00020   lcd_buttons btns;
00021   btns.left = left ? PRESSED : RELEASED;
00022   btns.middle = middle ? PRESSED : RELEASED;
00023   btns.right = right ? PRESSED : RELEASED;
00024
00025   return btns;
00026 }
```

5.13.4.4 lcd_print()

```
void lcd_print (
            unsigned int line,
            const char * str )
```

prints a string to a line on the lcd

Parameters

line	the line to print on
str	string to print

Author

Chris Jerrett

Date

9/9/2017

Definition at line **39** of file **lcd.c**.

Referenced by **promt_confirmation()**.

```
00039                                                   {
00040   lcd_assert();
00041   lcdSetText(lcd_port, line, str);
00042 }
```

5.13.4.5 lcd_printf()

```
void lcd_printf (
             unsigned int line,
             const char * format_str,
              ...  )
```

prints a formatted string to a line on the lcd. Smilar to printf

Parameters

line	the line to print on
format_str	format string string to print

Author

Chris Jerrett

Date

9/9/2017

5.13.4.6 lcd_set_backlight()

```
void lcd_set_backlight (
            bool state )
```

sets the backlight of the lcd

Parameters

state	a boolean representing the state of the backlight. true = on, false = off.

Author

Chris Jerrett

Date

9/9/2017

Definition at line **44** of file **lcd.c**.

References **lcdSetBacklight()**.

```
00044                                       {
00045   lcd_assert();
00046   lcdSetBacklight(lcd_port, state);
00047 }
```

5.13.4.7 promt_confirmation()

```
void promt_confirmation (
            const char * confirm_text )
```

Prompts the user to confirm a string. User must press middle button to confirm. Function is not thread safe and will stall a thread.

Parameters

confirm_text	the text for the user to confirm.

Author

Chris Jerrett

Date

9/9/2017

Definition at line **49** of file **lcd.c**.

References **delay()**, **lcd_get_pressed_buttons()**, **lcd_print()**, and **PRESSED**.

```
00049                                                 {
00050   lcd_assert();
00051   lcd_print(1, confirm_text);
00052   while (lcd_get_pressed_buttons().middle != PRESSED) {
00053     delay(200);
00054   }
00055 }
```

5.14 lcd.h

```
00001
00008 #ifndef _LCD_H_
00009 #define _LCD_H_
00010
00011 #include <API.h>
00012
00018 #define TOP_ROW 1
00019
00025 #define BOTTOM_ROW 2
00026
00036 typedef enum {
00038   RELEASED = false,
00040   PRESSED = true,
00041 } button_state;
00042
00048 typedef struct {
00049   button_state left;
00050   button_state middle;
00051   button_state right;
00052 } lcd_buttons;
00053
00061 lcd_buttons lcd_get_pressed_buttons();
00062
00068 void lcd_clear();
00069
00080 void init_main_lcd(FILE *lcd);
00081
00089 void lcd_print(unsigned int line, const char *str);
00090
00098 void lcd_printf(unsigned int line, const char *format_str, ...);
00099
00107 void lcd_set_backlight(bool state);
00108
00118 void promt_confirmation(const char *confirm_text);
00119
00120 #endif
```

5.15 include/main.h File Reference

Header file for global functions.

```
#include "API.h"
```

Functions

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

5.15.1 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.

This file is included by default in the predefined stubs in each VEX Cortex PROS Project.

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

Definition in file **main.h**.

5.15.2 Function Documentation

5.15.2.1 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()** (p. **??**) which should never exit. If it does so, the robot will await a switch to another mode or disable/enable cycle.

Definition at line **31** of file **auto.c**.

```
00031                   {
00032
00033   routine = near;
00034
00035   if(routine == near) {
00036     nearRoutine();
00037   } else if(routine == far) {
00038     farRoutine();
00039   }
00040
00041
00042 }
```

5.15.2.2 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()** (p. **??**) and **autonomous()** (p. **??**) tasks will not start. An autonomous mode selection menu like the pre_auton() in other environments can be implemented in this task if desired.

Definition at line **42** of file **init.c**.

References **imeInitializeAll()**, and **init_main_gyro()**.

```
00042                   {
00043   init_main_gyro();
00044   imeInitializeAll();
00045 /*  init_main_lcd(uart2);
00046
00047   setTeamName("9228B");
00048
00049
00050 lcdSetBacklight(uart2, false);
00051
00052 lcdSetText(uart2, 1, "Auton Routine");
00053 lcdSetText(uart2, 2, "NEAR   FAR   NONE");
00054
00055
00056 //lcdPrint(uart1, 1, "Auton Routine");
00057 //lcdPrint(uart1, 2, "NEAR       FAR       NONE");
00058
00059 unsigned int btnsPressed = lcdReadButtons(uart2);
00060
00061   if(btnsPressed & 0x1) {
00062     routine = near;
00063     lcdClear(uart2);
00064     lcdPrint(uart2, 1, "left pressed");
00065   } else if (btnsPressed & 0x2) {
00066     routine = far;
00067     lcdClear(uart2);
00068     lcdPrint(uart2, 1, "middle pressed");
00069   } else if (btnsPressed & 0x4) {
00070     routine = none;
00071     lcdClear(uart2);
00072     lcdPrint(uart2, 1, "right pressed");
00073   }
00074
00075
00076   lcdShutdown(uart2);
00077
00078
00079   //lcdSetBacklight(uart2, false);*/
00080
00081
00082
00083
00084 }
```

5.15.2.3 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()** (p. **??**)) and port states (**digitalWrite()** (p. **??**)) of limit switches, push buttons, and solenoids. It can also safely configure a UART port (usartOpen()) but cannot set up an LCD (**lcdInit()** (p. **??**)).

Definition at line **26** of file **init.c**.

```
00026                     {
00027 }
```

5.15.2.4 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()** (p. **??**) or **taskDelayUntil()** (p. **??**) 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.

Definition at line **35** of file **opcontrol.c**.

References **delay()**, and **drive()**.

```
00035                         {
00036         while (1) {
00037
00038                 drive();
00039
00040                 delay(20);
00041         }
00042 }
```

5.16 main.h

```
00001
00023 #ifndef MAIN_H_
00024
00025 // This prevents multiple inclusion, which isn't bad for this file but is good practice
00026 #define MAIN_H_
00027
00028 #include "API.h"
00029
00030 // Allow usage of this file in C++ programs
00031 #ifdef __cplusplus
00032 extern "C" {
00033 #endif
00034
00035 // A function prototype looks exactly like its declaration, but with a semicolon instead of
00036 // actual code. If a function does not match a prototype, compile errors will occur.
00037
00038 // Prototypes for initialization, operator control and autonomous
00039
00054 void autonomous();
00063 void initializeIO();
00077 void initialize();
00095 void operatorControl();
00096
00097 // End C++ export structure
00098 #ifdef __cplusplus
00099 }
00100 #endif
00101
00102 #endif
```

5.17 include/motor_ports.h File Reference

The motor port definitions

Macros for the different motors ports.

Macros

- #define **MAX_SPEED** 127

  The max speed of a motor].
- #define **MIN_SPEED** -127

  The min speed of a motor.
- #define **MOTOR_FRONT_LEFT** 1

  Front left drive motor of robot base.
- #define **MOTOR_FRONT_RIGHT** 2

  Front right drive motor of robot base.
- #define **MOTOR_BACK_LEFT** 3

  Back left drive motor of robot base.
- #define **MOTOR_BACK_RIGHT** 4

  Back right drive motor of robot base.
- #define **MOTOR_INTAKE** 5

  Port for the goliath intake motor.
- #define **MOTOR_BELT** 6

  Port for intake belt motor.
- #define **MOTOR_FLIPPER** 7

  The port for the cap flipper.
- #define **MOTOR_FLYWHEEL_A** 8

  Port for the first flywheel.
- #define **MOTOR_FLYWHEEL_B** 9

  Port for the second flywheel.

5.17.1 Detailed Description

The motor port definitions

Macros for the different motors ports.

Definition in file **motor_ports.h**.

5.17.2 Macro Definition Documentation

5.17.2.1 MAX_SPEED

```
#define MAX_SPEED 127
```

The max speed of a motor].

Author

Michael Baraty

Date

10/25/2018

Definition at line **14** of file **motor_ports.h**.

Referenced by **robotDriveForward()**, and **robotDriveReverse()**.

5.17.2.2 MIN_SPEED

```
#define MIN_SPEED -127
```

The min speed of a motor.

Author

Michael Baraty

Date

10/25/2018

Definition at line **22** of file **motor_ports.h**.

Referenced by **robotDriveForward()**, and **robotDriveReverse()**.

5.17.2.3 MOTOR_BACK_LEFT

```
#define MOTOR_BACK_LEFT 3
```

Back left drive motor of robot base.

Author

Michael Baraty

Date

10/25/2018

Definition at line **43** of file **motor_ports.h**.

Referenced by **drive()**, **robotDriveForward()**, **robotDriveReverse()**, and **robotStop()**.

5.17.2.4 MOTOR_BACK_RIGHT

```
#define MOTOR_BACK_RIGHT 4
```

Back right drive motor of robot base.

Author

Michael Baraty

Date

10/25/2018

Definition at line **50** of file **motor_ports.h**.

Referenced by **drive()**, **robotDriveForward()**, **robotDriveReverse()**, and **robotStop()**.

5.17.2.5 MOTOR_BELT

```
#define MOTOR_BELT 6
```

Port for intake belt motor.

Author

Michael Baraty

Date

10/25/2018

Definition at line **64** of file **motor_ports.h**.

Referenced by **intakeStop()**.

5.17.2.6 MOTOR_FLIPPER

```
#define MOTOR_FLIPPER 7
```

The port for the cap flipper.

Author

Michael Baraty

Date

10/25/2018

Definition at line **71** of file **motor_ports.h**.

Referenced by **flipperStop()**.

5.17.2.7 MOTOR_FLYWHEEL_A

```
#define MOTOR_FLYWHEEL_A 8
```

Port for the first flywheel.

Author

Michael Baraty

Date

10/25/2018

Definition at line **78** of file **motor_ports.h**.

Referenced by **flywheelSet()**.

5.17.2.8 MOTOR_FLYWHEEL_B

```
#define MOTOR_FLYWHEEL_B 9
```

Port for the second flywheel.

Author

Michael Baraty

Date

10/25/2018

Definition at line **85** of file **motor_ports.h**.

Referenced by **flywheelSet()**.

5.17.2.9 MOTOR_FRONT_LEFT

```
#define MOTOR_FRONT_LEFT 1
```

Front left drive motor of robot base.

Author

Michael Baraty

Date

10/25/2018

Definition at line **29** of file **motor_ports.h**.

Referenced by **drive()**, **robotDriveForward()**, **robotDriveReverse()**, and **robotStop()**.

5.17.2.10 MOTOR_FRONT_RIGHT

```
#define MOTOR_FRONT_RIGHT 2
```

Front right drive motor of robot base.

Author

Michael Baraty

Date

10/25/2018

Definition at line **36** of file **motor_ports.h**.

Referenced by **drive()**, **robotDriveForward()**, **robotDriveReverse()**, and **robotStop()**.

5.17.2.11 MOTOR_INTAKE

```
#define MOTOR_INTAKE 5
```

Port for the goliath intake motor.

Author

Michael Baraty

Date

10/25/2018

Definition at line **57** of file **motor_ports.h**.

Referenced by **intakeStop()**.

5.18 motor_ports.h

```
00001
00006 #ifndef _MOTOR_PORTS_H_
00007 #define _MOTOR_PORTS_H_
00008
00014 #define MAX_SPEED 127
00015
00016
00022 #define MIN_SPEED -127
00023
00029 #define MOTOR_FRONT_LEFT 1
00030
00036 #define MOTOR_FRONT_RIGHT 2
00037
00043 #define MOTOR_BACK_LEFT 3
00044
00050 #define MOTOR_BACK_RIGHT 4
00051
00057 #define MOTOR_INTAKE 5
00058
00064 #define MOTOR_BELT 6
00065
00071 #define MOTOR_FLIPPER 7
00072
00078 #define MOTOR_FLYWHEEL_A 8
00079
00085 #define MOTOR_FLYWHEEL_B 9
00086
00087 #endif
```

5.19 src/auto.c File Reference

File for autonomous code.

```
#include <main.h>
#include "drive.h"
#include "auton.h"
```

Functions

- void **autonomous** ()
- void **nearRoutine** ()

  begins the autonomous routine from the square nearest to the flag
- void **farRoutine** ()

  begins the autonomous routine from the square farthest from the flag

5.19.1 Detailed Description

File for autonomous code.

This file should contain the user **autonomous()** (p. **??**) function and any functions related to it.

Any copyright is dedicated to the Public Domain. `http://creativecommons.org/publicdomain/zero/1.0/`

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

Definition in file **auto.c**.

5.19.2 Function Documentation

5.19.2.1 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()** (p. **??**) which should never exit. If it does so, the robot will await a switch to another mode or disable/enable cycle.

Definition at line **31** of file **auto.c**.

```
00031                   {
00032
00033   routine = near;
00034
00035   if(routine == near) {
00036     nearRoutine();
00037   } else if(routine == far) {
00038     farRoutine();
00039   }
00040
00041
00042 }
```

5.19.2.2 farRoutine()

```
void farRoutine ( )
```

begins the autonomous routine from the square farthest from the flag

Date

11/10/2018

Author

Michael Baraty

Definition at line **70** of file **auto.c**.

References **flywheelSet()**, **intakeSet()**, and **taskDelay()**.

```
00070                   {
00071   flywheelSet(120);
00072   taskDelay(4000);
00073   intakeSet(forward);
00074   taskDelay(1800);
00075   motorStopAll();
00076   }
```

5.19.2.3 nearRoutine()

```
void nearRoutine ( )
```

begins the autonomous routine from the square nearest to the flag

Date

11/10/2018

Author

Michael Baraty

Definition at line **44** of file **auto.c**.

References **flywheelSet()**, **intakeSet()**, and **taskDelay()**.

```
00044                   {
00045   //fire up the flywheels
00046   flywheelSet(70);
00047   taskDelay(4000);
00048   intakeSet(forward);
00049   taskDelay(1800);
00050   motorStopAll();
00051
00052   //drive for bottom flag
00053   motorSet(MOTOR_BACK_LEFT, MAX_SPEED / 2);
00054   motorSet(MOTOR_BACK_RIGHT, MIN_SPEED / 2);
00055   motorSet(MOTOR_FRONT_RIGHT, MIN_SPEED / 2 + 30);
00056   motorSet(MOTOR_FRONT_LEFT, MIN_SPEED / 2);
00057   taskDelay(2000);
00058   robotStop();
00059   taskDelay(500);
00060
00061   //prepare for tele-op
00062   motorSet(MOTOR_BACK_LEFT, -MAX_SPEED / 2);
00063   motorSet(MOTOR_BACK_RIGHT, -MIN_SPEED / 2);
00064   motorSet(MOTOR_FRONT_RIGHT, -MIN_SPEED / 2);
00065   motorSet(MOTOR_FRONT_LEFT, -MIN_SPEED / 2);
00066   taskDelay(1500);
00067   robotStop();
00068 }
```

5.20 auto.c

```
00001
00013 #include <main.h>
00014 #include "drive.h"
00015 #include "auton.h"
00016
00017 /*
00018  * Runs the user autonomous code. This function will be started in its own task with the default
00019  * priority and stack size whenever the robot is enabled via the Field Management System or the
00020  * VEX Competition Switch in the autonomous mode. If the robot is disabled or communications is
00021  * lost, the autonomous task will be stopped by the kernel. Re-enabling the robot will restart
00022  * the task, not re-start it from where it left off.
00023  *
00024  * Code running in the autonomous task cannot access information from the VEX Joystick. However,
00025  * the autonomous function can be invoked from another task if a VEX Competition Switch is not
00026  * available, and it can access joystick information if called in this way.
00027  *
00028  * The autonomous task may exit, unlike operatorControl() which should never exit. If it does
00029  * so, the robot will await a switch to another mode or disable/enable cycle.
00030  */
```

```
00031 void autonomous() {
00032
00033   routine = near;
00034
00035   if(routine == near) {
00036     nearRoutine();
00037   } else if(routine == far) {
00038     farRoutine();
00039   }
00040
00041
00042 }
00043
00044 void nearRoutine() {
00045   //fire up the flywheels
00046   flywheelSet(70);
00047   taskDelay(4000);
00048   intakeSet(forward);
00049   taskDelay(1800);
00050   motorStopAll();
00051
00052   //drive for bottom flag
00053   motorSet(MOTOR_BACK_LEFT, MAX_SPEED / 2);
00054   motorSet(MOTOR_BACK_RIGHT, MIN_SPEED / 2);
00055   motorSet(MOTOR_FRONT_RIGHT, MIN_SPEED / 2 + 30);
00056   motorSet(MOTOR_FRONT_LEFT, MIN_SPEED / 2);
00057   taskDelay(2000);
00058   robotStop();
00059   taskDelay(500);
00060
00061   //prepare for tele-op
00062   motorSet(MOTOR_BACK_LEFT, -MAX_SPEED / 2);
00063   motorSet(MOTOR_BACK_RIGHT, -MIN_SPEED / 2);
00064   motorSet(MOTOR_FRONT_RIGHT, -MIN_SPEED / 2);
00065   motorSet(MOTOR_FRONT_LEFT, -MIN_SPEED / 2);
00066   taskDelay(1500);
00067   robotStop();
00068 }
00069
00070 void farRoutine() {
00071   flywheelSet(120);
00072   taskDelay(4000);
00073   intakeSet(forward);
00074   taskDelay(1800);
00075   motorStopAll();
00076   }
```

5.21 src/init.c File Reference

File for initialization code.

```
#include "main.h"
#include "gyro.h"
#include "potentiometer.h"
#include "encoders.h"
```

Functions

- void **initializeIO** ()
- void **initialize** ()

5.21.1 Detailed Description

File for initialization code.

This file should contain the user **initialize()** (p. **??**) function and any functions related to it.

Any copyright is dedicated to the Public Domain. http://creativecommons.org/publicdomain/zero/1.↩
0/

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

Definition in file **init.c**.

5.21.2 Function Documentation

5.21.2.1 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()** (p. **??**) and **autonomous()** (p. **??**) tasks will not start. An autonomous mode selection menu like the pre_auton() in other environments can be implemented in this task if desired.

Definition at line **42** of file **init.c**.

References **imeInitializeAll()**, and **init_main_gyro()**.

```
00042                   {
00043   init_main_gyro();
00044   imeInitializeAll();
00045 /*  init_main_lcd(uart2);
00046
00047   setTeamName("9228B");
00048
00049
00050 lcdSetBacklight(uart2, false);
00051
00052 lcdSetText(uart2, 1, "Auton Routine");
00053 lcdSetText(uart2, 2, "NEAR   FAR   NONE");
00054
00055
00056 //lcdPrint(uart1, 1, "Auton Routine");
00057 //lcdPrint(uart1, 2, "NEAR       FAR       NONE");
00058
00059 unsigned int btnsPressed = lcdReadButtons(uart2);
00060
00061   if(btnsPressed & 0x1) {
00062     routine = near;
00063     lcdClear(uart2);
```

```
00064     lcdPrint(uart2, 1, "left pressed");
00065   } else if (btnsPressed & 0x2) {
00066     routine = far;
00067     lcdClear(uart2);
00068     lcdPrint(uart2, 1, "middle pressed");
00069   } else if (btnsPressed & 0x4) {
00070     routine = none;
00071     lcdClear(uart2);
00072     lcdPrint(uart2, 1, "right pressed");
00073   }
00074
00075
00076   lcdShutdown(uart2);
00077
00078
00079   //lcdSetBacklight(uart2, false);*/
00080
00081
00082
00083
00084 }
```

5.21.2.2 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()** (p. **??**)) and port states (**digitalWrite()** (p. **??**)) of limit switches, push buttons, and solenoids. It can also safely configure a UART port (usartOpen()) but cannot set up an LCD (**lcdInit()** (p. **??**)).

Definition at line **26** of file **init.c**.

```
00026                   {
00027 }
```

5.22 init.c

```
00001
00013 #include "main.h"
00014 #include "gyro.h"
00015 #include "potentiometer.h"
00016 #include "encoders.h"
00017
00018 /*
00019  * Runs pre-initialization code. This function will be started in kernel mode one time while the
00020  * VEX Cortex is starting up. As the scheduler is still paused, most API functions will fail.
00021  *
00022  * The purpose of this function is solely to set the default pin modes (pinMode()) and port
00023  * states (digitalWrite()) of limit switches, push buttons, and solenoids. It can also safely
00024  * configure a UART port (usartOpen()) but cannot set up an LCD (lcdInit()).
00025  */
00026 void initializeIO() {
00027 }
00028
00029 /*
00030  * Runs user initialization code. This function will be started in its own task with the default
00031  * priority and stack size once when the robot is starting up. It is possible that the VEXnet
00032  * communication link may not be fully established at this time, so reading from the VEX
00033  * Joystick may fail.
00034  *
```

```
00035  * This function should initialize most sensors (gyro, encoders, ultrasonics), LCDs, global
00036  * variables, and IMEs.
00037  *
00038  * This function must exit relatively promptly, or the operatorControl() and autonomous() tasks
00039  * will not start. An autonomous mode selection menu like the pre_auton() in other environments
00040  * can be implemented in this task if desired.
00041  */
00042 void initialize() {
00043   init_main_gyro();
00044   imeInitializeAll();
00045 /*  init_main_lcd(uart2);
00046
00047   setTeamName("9228B");
00048
00049
00050 lcdSetBacklight(uart2, false);
00051
00052 lcdSetText(uart2, 1, "Auton Routine");
00053 lcdSetText(uart2, 2, "NEAR   FAR   NONE");
00054
00055
00056 //lcdPrint(uart1, 1, "Auton Routine");
00057 //lcdPrint(uart1, 2, "NEAR       FAR       NONE");
00058
00059 unsigned int btnsPressed = lcdReadButtons(uart2);
00060
00061   if(btnsPressed & 0x1) {
00062     routine = near;
00063     lcdClear(uart2);
00064     lcdPrint(uart2, 1, "left pressed");
00065   } else if (btnsPressed & 0x2) {
00066     routine = far;
00067     lcdClear(uart2);
00068     lcdPrint(uart2, 1, "middle pressed");
00069   } else if (btnsPressed & 0x4) {
00070     routine = none;
00071     lcdClear(uart2);
00072     lcdPrint(uart2, 1, "right pressed");
00073   }
00074
00075
00076   lcdShutdown(uart2);
00077
00078
00079   //lcdSetBacklight(uart2, false);*/
00080
00081
00082
00083
00084 }
```

5.23 src/opcontrol.c File Reference

File for operator control code.

```
#include "main.h"
#include "controller.h"
#include "motor_ports.h"
#include "drive.h"
#include "potentiometer.h"
```

Functions

- void **operatorControl** ()

5.23.1 Detailed Description

File for operator control code.

This file should contain the user **operatorControl()** (p. **??**) function and any functions related to it.

Any copyright is dedicated to the Public Domain. `http://creativecommons.org/publicdomain/zero/1.0/`

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

Definition in file **opcontrol.c**.

5.23.2 Function Documentation

5.23.2.1 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()** (p. **??**) or **taskDelayUntil()** (p. **??**) 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.

Definition at line **35** of file **opcontrol.c**.

References **delay()**, and **drive()**.

```
00035                         {
00036         while (1) {
00037
00038                 drive();
00039
00040                 delay(20);
00041         }
00042 }
```

5.24 opcontrol.c

```
00001
00013 #include "main.h"
00014 #include "controller.h"
00015 #include "motor_ports.h"
00016 #include "drive.h"
00017 #include "potentiometer.h"
00018 /*
00019  * Runs the user operator control code. This function will be started in its own task with the
00020  * default priority and stack size whenever the robot is enabled via the Field Management System
00021  * or the VEX Competition Switch in the operator control mode. If the robot is disabled or
00022  * communications is lost, the operator control task will be stopped by the kernel. Re-enabling
00023  * the robot will restart the task, not resume it from where it left off.
00024  *
00025  * If no VEX Competition Switch or Field Management system is plugged in, the VEX Cortex will
00026  * run the operator control task. Be warned that this will also occur if the VEX Cortex is
00027  * tethered directly to a computer via the USB A to A cable without any VEX Joystick attached.
00028  *
00029  * Code running in this task can take almost any action, as the VEX Joystick is available and
00030  * the scheduler is operational. However, proper use of delay() or taskDelayUntil() is highly
00031  * recommended to give other tasks (including system tasks such as updating LCDs) time to run.
00032  *
00033  * This task should never exit; it should end with some kind of infinite loop, even if empty.
00034  */
00035 void operatorControl() {
00036       while (1) {
00037
00038             drive();
00039
00040             delay(20);
00041       }
00042 }
```