```c
/** @file API.h
 * @brief Provides the high-level user functionality intended for use by typical VEX Cortex
 * programmers.
 *
 * This file should be included for you in the predefined stubs in each new VEX Cortex PROS
 * project through the inclusion of "main.h". In any new C source file, it is advisable to
 * include main.h instead of referencing API.h by name, to better handle any nomenclature
 * changes to this file or its contents.
 *
 * Copyright (c) 2011-2016, Purdue University ACM SIGBots.
 * All rights reserved.
 *
 * This Source Code Form is subject to the terms of the Mozilla Public
 * License, v. 2.0. If a copy of the MPL was not distributed with this
 * file, You can obtain one at http://mozilla.org/MPL/2.0/.
 *
 * PROS contains FreeRTOS (http://www.freertos.org) whose source code may be
 * obtained from http://sourceforge.net/projects/freertos/files/ or on request.
 */

#ifndef API_H_
#define API_H_

// System includes
#include <stdlib.h>
#include <stdbool.h>
#include <stdarg.h>
#include <stdint.h>

// Begin C++ extern to C
#ifdef __cplusplus
extern "C" {
#endif

// -------------------- VEX competition functions --------------------

/**
 * DOWN button (valid on channels 5, 6, 7, 8)
 */
#define JOY_DOWN 1
/**
 * LEFT button (valid on channels 7, 8)
 */
#define JOY_LEFT 2
/**
 * UP button (valid on channels 5, 6, 7, 8)
```

```
 */
#define JOY_UP 4
/**
 * RIGHT button (valid on channels 7, 8)
 */
#define JOY_RIGHT 8
/**
 * Analog axis for the X acceleration from the VEX Joystick.
 */
#define ACCEL_X 5
/**
 * Analog axis for the Y acceleration from the VEX Joystick.
 */
#define ACCEL_Y 6

/**
 * 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.
 */
bool isAutonomous();
/**
 * 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 affe
 * the robot state (e.g. solenoids). Avoid performing externally visible actions while
 * disabled (the kernel should take care of this most of the time).
 */
bool isEnabled();
/**
 * Returns true if a joystick is connected to the specified slot number (1 or 2), or false
 * otherwise.
 *
 * Useful for automatically merging joysticks for one operator, or splitting for two. This
 * function does not work properly during initialize() or initializeIO() and can return fal
 * positives. It should be checked once and stored at the beginning of operatorControl().
 *
 * @param joystick the joystick slot to check
 */
bool isJoystickConnected(unsigned char joystick);
/**
 * Returns true if a VEX field controller or competition switch is connected, or false
 * otherwise.
 *
```

```
 * When in online mode, the switching between autonomous() and operatorControl() tasks is
 * managed by the PROS kernel.
 */
bool isOnline();
/**
 * 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.
 *
 * @param joystick the joystick slot to check
 * @param axis one of 1, 2, 3, 4, ACCEL_X, or ACCEL_Y
 */
int joystickGetAnalog(unsigned char joystick, unsigned char axis);
/**
 * Gets the value of a button on the VEX joystick. Returns true if that button is pressed, o
 * false otherwise. If no joystick is connected to the requested slot, returns false.
 *
 * @param joystick the joystick slot to check
 * @param buttonGroup one of 5, 6, 7, or 8 to request that button as labelled on the joystic
 * @param 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
 */
bool joystickGetDigital(unsigned char joystick, unsigned char buttonGroup,
        unsigned char button);
/**
 * Returns the backup battery voltage in millivolts.
 *
 * If no backup battery is connected, returns 0.
 */
unsigned int powerLevelBackup();
/**
 * Returns the main battery voltage in millivolts.
 *
 * In rare circumstances, this method might return 0. Check the output value for reasonabil
 * before blindly blasting the user.
 */
unsigned int powerLevelMain();
/**
 * Sets the team name displayed to the VEX field control and VEX Firmware Upgrade.
 *
 * @param name a string containing the team name; only the first eight characters will be sh
 */
void setTeamName(const char *name);

// -------------------- Pin control functions --------------------

/**
```

```
 * There are 8 available analog I/O on the Cortex.
 */
#define BOARD_NR_ADC_PINS 8
/**
 * 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 port
 *
 * 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.
 */
#define BOARD_NR_GPIO_PINS 27
/**
 * Used for digitalWrite() to specify a logic HIGH state to output.
 *
 * In reality, using any non-zero expression or "true" will work to set a pin to HIGH.
 */
#define HIGH 1
/**
 * Used for digitalWrite() to specify a logic LOW state to output.
 *
 * In reality, using a zero expression or "false" will work to set a pin to LOW.
 */
#define LOW 0

/**
 * pinMode() state for digital input, with pullup.
 *
 * This is the default state for the 12 Digital pins. The pullup causes the input to read as
 * "HIGH" when unplugged, but is fairly weak and can safely be driven by most sources. Many
 * digital sensors rely on this behavior and cannot be used with INPUT_FLOATING.
 */
#define INPUT 0x0A
/**
 * pinMode() state for analog inputs.
 *
 * This is the default state for the 8 Analog pins and the Speaker port. This only works on
 * pins with analog input capabilities; use anywhere else results in undefined behavior.
 */
#define INPUT_ANALOG 0x00
/**
 * pinMode() state for digital input, without pullup.
 *
 * Beware of power consumption, as digital inputs left "floating" may switch back and forth
 * and cause spurious interrupts.
```

```
 */
#define INPUT_FLOATING 0x04
/**
 * pinMode() state for digital output, push-pull.
 *
 * This is the mode which should be used to output a digital HIGH or LOW value from the Cor
 * This mode is useful for pneumatic solenoid valves and VEX LEDs.
 */
#define OUTPUT 0x01
/**
 * pinMode() state for open-drain outputs.
 *
 * This is useful in a few cases for external electronics and should not be used for the VE
 * solenoid or LEDs.
 */
#define OUTPUT_OD 0x05

/**
 * 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
 * analogReadCalibrated() and analogReadCalibratedHR() functions. These functions will retu
 * the difference between this value and the current sensor value when called.
 *
 * Do not use this function in initializeIO(), or when the sensor value might be unstable
 * (gyro rotation, accelerometer movement).
 *
 * This function may not work properly if the VEX Cortex is tethered to a PC using the oran
 * USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Batter
 * provides power to sensors.
 *
 * @param channel the channel to calibrate from 1-8
 * @return the average sensor value computed by this function
 */
int analogCalibrate(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 oran
 * USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Batter
```

```
 * provides power to sensors.
 *
 * @param channel the channel to read from 1-8
 * @return the analog sensor value, where a value of 0 reflects an input voltage of nearly 0
 * and a value of 4095 reflects an input voltage of nearly 5 V
 */
int analogRead(unsigned char channel);
/**
 * Reads the calibrated value of an analog input channel.
 *
 * The analogCalibrate() function must be run first on that channel. This function is
 * inappropriate for sensor values intended for integration, as round-off error can accumulate
 * causing drift over time. Use analogReadCalibratedHR() instead.
 *
 * This function may not work properly if the VEX Cortex is tethered to a PC using the orange
 * USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Battery
 * provides power to sensors.
 *
 * @param channel the channel to read from 1-8
 * @return the difference of the sensor value from its calibrated default from -4095 to 4095
 */
int analogReadCalibrated(unsigned char channel);
/**
 * Reads the calibrated value of an analog input channel 1-8 with enhanced precision.
 *
 * The analogCalibrate() function must be run first. This is intended for integrated sensor
 * values such as gyros and accelerometers to reduce drift due to round-off, and should not
 * 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.
 *
 * @param channel the channel to read from 1-8
 * @return the difference of the sensor value from its calibrated default from -16384 to 163
 */
int analogReadCalibratedHR(unsigned char channel);
/**
 * 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. T
```

```
 * 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 orang
 * USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Batter
 * provides power to sensors.
 *
 * @param pin the pin to read from 1-26
 * @return true if the pin is HIGH, or false if it is LOW
 */
bool digitalRead(unsigned char pin);
/**
 * 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.
 *
 * @param pin the pin to write from 1-26
 * @param value an expression evaluating to "true" or "false" to set the output to HIGH or L
 * respectively, or the constants HIGH or LOW themselves
 */
void digitalWrite(unsigned char pin, bool value);
/**
 * 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 funct
 * is Wiring-compatible.
 *
 * @param pin the pin to modify from 1-26
 * @param mode one of INPUT, INPUT_ANALOG, INPUT_FLOATING, OUTPUT, or OUTPUT_OD
 */
void pinMode(unsigned char pin, unsigned char mode);

/*
 * Digital port 10 cannot be used as an interrupt port, or for an encoder. Plan accordingly.
 */

/**
 * When used in ioSetInterrupt(), triggers an interrupt on rising edges (LOW to HIGH).
 */
#define INTERRUPT_EDGE_RISING 1
/**
 * When used in ioSetInterrupt(), triggers an interrupt on falling edges (HIGH to LOW).
 */
#define INTERRUPT_EDGE_FALLING 2
```

```
/**
 * When used in ioSetInterrupt(), triggers an interrupt on both rising and falling edges
 * (LOW to HIGH or HIGH to LOW).
 */
#define INTERRUPT_EDGE_BOTH 3
/**
 * Type definition for interrupt handlers. Such functions must accept one argument indicati
 * the pin which changed.
 */
typedef void (*InterruptHandler)(unsigned char pin);

/**
 * Disables interrupts on the specified pin.
 *
 * Disabling interrupts on interrupt pins which are not in use conserves processing time.
 *
 * @param pin the pin on which to reset interrupts from 1-9,11-12
 */
void ioClearInterrupt(unsigned char pin);
/**
 * 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 wit
 * the pin that changed as an argument. Enabling pin-change interrupts consumes processing
 * time, so it is best to only enable necessary interrupts and to keep the InterruptHandler
 * function short. Pin change interrupts can only be enabled on pins 1-9 and 11-12.
 *
 * Do not use API functions such as delay() inside the handler function, as the function wi
 * run in an ISR where the scheduler is paused and no other interrupts can execute. It is b
 * 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.
 *
 * @param pin the pin on which to enable interrupts from 1-9,11-12
 * @param edges one of INTERRUPT_EDGE_RISING, INTERRUPT_EDGE_FALLING, or INTERRUPT_EDGE_BOT
 * @param handler the function to call when the condition is satisfied
 */
void ioSetInterrupt(unsigned char pin, unsigned char edges, InterruptHandler handler);

// -------------------- Physical output control functions --------------------

/**
 * 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 guarantee
 * 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 loadi
 * 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.
 *
 * @param channel the motor channel to fetch from 1-10
 * @return the speed last sent to this channel; -127 is full reverse and 127 is full forward
 * with 0 being off
 */
int motorGet(unsigned char channel);
/**
 * Sets the speed of the specified motor channel.
 *
 * Do not use motorSet() with the same channel argument from two different tasks. It is safe
 * use motorSet() with different channel arguments from different tasks.
 *
 * @param channel the motor channel to modify from 1-10
 * @param speed the new signed speed; -127 is full reverse and 127 is full forward, with 0
 * being off
 */
void motorSet(unsigned char channel, int speed);
/**
 * Stops the motor on the specified channel, equivalent to calling motorSet() with an argume
 * of zero.
 *
 * This performs a coasting stop, not an active brake. Since motorStop is similar to
 * motorSet(0), see the note for motorSet() about use from multiple tasks.
 *
 * @param channel the motor channel to stop from 1-10
 */
void motorStop(unsigned char channel);
/**
 * Stops all motors; significantly faster than looping through all motor ports and calling
 * motorSet(channel, 0) on each one.
 */
void motorStopAll();

/**
 * Initializes VEX speaker support.
 *
 * The VEX speaker is not thread safe; it can only be used from one task at a time. Using th
 * VEX speaker may impact robot performance. Teams may benefit from an if statement that onl
 * enables sound if isOnline() returns false.
 */
void speakerInit();
```

```
/**
 * 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 o
 * 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 th
 * VEX speaker may impact robot performance. Teams may benefit from an if statement that onl
 * enables sound if isOnline() returns false.
 *
 * @param songs an array of up to three (3) RTTTL songs as string values to play
 */
void speakerPlayArray(const char * * songs);
/**
 * 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 i
 * undefined.
 *
 * The VEX speaker is not thread safe; it can only be used from one task at a time. Using th
 * VEX speaker may impact robot performance. Teams may benefit from an if statement that onl
 * enables sound if isOnline() returns false.
 *
 * @param song the RTTTL song as a string value to play
 */
void speakerPlayRtttl(const char *song);
/**
 * 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.
 */
void speakerShutdown();

// -------------------- VEX sensor control functions --------------------

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

/**
```

```
 * Initializes all IMEs.
 *
 * IMEs are assigned sequential incrementing addresses, beginning with the first IME on the
 * chain (closest to the VEX Cortex I2C port). Therefore, a given configuration of IMEs will
 * always have the same ID assigned to each encoder. The addresses range from 0 to
 * IME_ADDR_MAX, so the first encoder gets 0, the second gets 1, ...
 *
 * This function should most likely be used in initialize(). Do not use it in initializeIO()
 * 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
 * expected.
 *
 * This function, unlike the other IME functions, is not thread safe. If using imeInitialize
 * to re-initialize encoders, calls to other IME functions might behave unpredictably during
 * this function's execution.
 *
 * @return the number of IMEs successfully initialized.
 */
unsigned int imeInitializeAll();
/**
 * 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 d
 * different number of counts per revolution:
 *
 * * \c 240.448 for the 269 IME
 * * \c 627.2 for the 393 IME in high torque mode (factory default)
 * * \c 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.
 *
 * @param address the IME address to fetch from 0 to IME_ADDR_MAX
 * @param value a pointer to the location where the value will be stored (obtained using the
 * "&" operator on the target variable name e.g. <code>imeGet(2, &counts)</code>)
 * @return true if the count was successfully read and the value stored in *value is valid;
 * false otherwise
 */
bool imeGet(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:
 *
 * * \c 30.056 for the 269 IME
 * * \c 39.2 for the 393 IME in high torque mode (factory default)
 * * \c 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.
 *
 * @param address the IME address to fetch from 0 to IME_ADDR_MAX
 * @param value a pointer to the location where the value will be stored (obtained using the
 * "&" operator on the target variable name e.g. <code>imeGetVelocity(2, &counts)</code>)
 * @return true if the velocity was successfully read and the value stored in *value is valid
 * false otherwise
 */
bool imeGetVelocity(unsigned char address, int *value);
/**
 * Resets the specified IME's counters to zero.
 *
 * This method can be used while the IME is rotating.
 *
 * @param address the IME address to reset from 0 to IME_ADDR_MAX
 * @return true if the reset succeeded; false otherwise
 */
bool imeReset(unsigned char address);
/**
 * 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 agai
 */
void imeShutdown();

/**
 * 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.
 */
typedef void * Gyro;

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

```
 * There are 360 degrees in a circle.
 *
 * @param gyro the Gyro object from gyroInit() to read
 * @return the signed and cumulative number of degrees rotated around the gyro's vertical ax
 * since the last start or reset
 */
int gyroGet(Gyro gyro);
/**
 * 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
 * gyro implicitly calibrates it and resets its count. Do not move the robot while the gyro
 * being calibrated. It is suggested to call this function in initialize() and to place the
 * robot in its final position before powering it on.
 *
 * The multiplier parameter can tune the gyro to adapt to specific sensors. The default valu
 * at this time is 196; higher values will increase the number of degrees reported for a fix
 * 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.
 *
 * @param port the analog port to use from 1-8
 * @param multiplier an optional constant to tune the gyro readings; use 0 for the default
 * value
 * @return a Gyro object to be stored and used for later calls to gyro functions
 */
Gyro gyroInit(unsigned char port, unsigned short multiplier);
/**
 * 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.
 *
 * @param gyro the Gyro object from gyroInit() to reset
 */
void gyroReset(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.
 *
 * @param gyro the Gyro object from gyroInit() to stop
 */
void gyroShutdown(Gyro gyro);
```

```
/**
 * 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.
 */
typedef void * Encoder;
/**
 * Gets the number of ticks recorded by the encoder.
 *
 * There are 360 ticks in one revolution.
 *
 * @param enc the Encoder object from encoderInit() to read
 * @return the signed and cumulative number of counts since the last start or reset
 */
int encoderGet(Encoder enc);
/**
 * 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 implicit
 * resets its count.
 *
 * @param portTop the "top" wire from the encoder sensor with the removable cover side UP
 * @param portBottom the "bottom" wire from the encoder sensor
 * @param reverse if "true", the sensor will count in the opposite direction
 * @return an Encoder object to be stored and used for later calls to encoder functions
 */
Encoder encoderInit(unsigned char portTop, unsigned char portBottom, bool reverse);
/**
 * Resets the encoder to zero.
 *
 * It is safe to use this method while an encoder is enabled. It is not necessary to call th
 * method before stopping or starting an encoder.
 *
 * @param enc the Encoder object from encoderInit() to reset
 */
void encoderReset(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.
 *
 * @param enc the Encoder object from encoderInit() to stop
 */
```

```
void encoderShutdown(Encoder enc);

/**
 * This value is returned if the sensor cannot find a reasonable value to return.
 */
#define ULTRA_BAD_RESPONSE -1

/**
 * Reference type for an initialized ultrasonic sensor.
 *
 * Ultrasonic information is stored as an opaque pointer to a structure in memory; as this
 * pointer type, it can be safely passed or stored by value.
 */
typedef void * Ultrasonic;
/**
 * 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 wai
 * for a response, -1 (ULTRA_BAD_RESPONSE) is returned.
 * If the ultrasonic sensor was never started, the return value is undefined. Round and flu
 * objects can cause inaccurate values to be returned.
 *
 * @param ult the Ultrasonic object from ultrasonicInit() to read
 * @return the distance to the nearest object in centimeters
 */
int ultrasonicGet(Ultrasonic ult);
/**
 * 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.
 *
 * @param portEcho the port connected to the orange cable from 1-9,11-12
 * @param portPing the port connected to the yellow cable from 1-12
 * @return an Ultrasonic object to be stored and used for later calls to ultrasonic functio
 */
Ultrasonic ultrasonicInit(unsigned char portEcho, unsigned char portPing);
/**
 * Stops and disables the ultrasonic sensor.
 *
 * The last distance it had before stopping will be retained. One more ping operation may o
 * before the sensor is fully disabled.
 *
 * @param ult the Ultrasonic object from ultrasonicInit() to stop
 */
```

```
void ultrasonicShutdown(Ultrasonic ult);

// --------------------- Custom sensor control functions ---------------------

// ---- I2C port control ----
/**
 * i2cRead - Reads the specified number of data bytes from the specified 7-bit I2C address.
 * 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.
 */
bool i2cRead(uint8_t addr, uint8_t *data, 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 s
 * 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
 * separate registers into a larger read using this function whenever possible to improve co
 * reliability, even if a few intermediate values need to be thrown away.
 */
bool i2cReadRegister(uint8_t addr, uint8_t reg, uint8_t *value, 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 f
 * 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.
 */
bool i2cWrite(uint8_t addr, uint8_t *data, uint16_t count);
/**
 * 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 fo
 * the vast majority of I2C operations, writing multiple bytes requires the i2cWrite method
 */
bool i2cWriteRegister(uint8_t addr, uint8_t reg, uint16_t value);

/**
 * 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 th
 * actually nothing there.
 */
typedef int PROS_FILE;


#ifndef 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 incorpo
 * into PROS projects. If you're not using C++/newlib, you can disregard this and just use P
 */
#define FILE PROS_FILE
#endif

/**
 * Bit mask for usartInit() for 8 data bits (typical)
 */
#define SERIAL_DATABITS_8 0x0000
/**
 * Bit mask for usartInit() for 9 data bits
 */
#define SERIAL_DATABITS_9 0x1000
/**
 * Bit mask for usartInit() for 1 stop bit (typical)
 */
#define SERIAL_STOPBITS_1 0x0000
/**
 * Bit mask for usartInit() for 2 stop bits
 */
#define SERIAL_STOPBITS_2 0x2000
/**
 * Bit mask for usartInit() for No parity (typical)
 */
#define SERIAL_PARITY_NONE 0x0000
/**
```

```
 * Bit mask for usartInit() for Even parity
 */
#define SERIAL_PARITY_EVEN 0x0400
/**
 * Bit mask for usartInit() for Odd parity
 */
#define SERIAL_PARITY_ODD 0x0600
/**
 * Specifies the default serial settings when used in usartInit()
 */
#define SERIAL_8N1 0x0000

/**
 * Initialize the specified serial interface with the given connection parameters.
 *
 * I/O to the port is accomplished using the "standard" I/O functions such as fputs(),
 * fprintf(), and fputc().
 *
 * Re-initializing an open port may cause loss of data in the buffers. This routine may be
 * safely called from initializeIO() or when the scheduler is paused. If I/O is attempted on
 * serial port which has never been opened, the behavior will be the same as if the port had
 * been disabled.
 *
 * @param usart the port to open, either "uart1" or "uart2"
 * @param baud the baud rate to use from 2400 to 1000000 baud
 * @param flags a bit mask combination of the SERIAL_* flags specifying parity, stop, and da
 * bits
 */
void usartInit(PROS_FILE *usart, unsigned int baud, unsigned int flags);
/**
 * 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.
 *
 * @param usart the port to close, either "uart1" or "uart2"
 */
void usartShutdown(PROS_FILE *usart);

// -------------------- Character input and output --------------------

/**
 * The standard output stream uses the PC debug terminal.
 */
#define stdout ((PROS_FILE *)3)
```

```
/**
 * The standard input stream uses the PC debug terminal.
 */
#define stdin ((PROS_FILE *)3)
/**
 * UART 1 on the Cortex; must be opened first using usartInit().
 */
#define uart1 ((PROS_FILE *)1)
/**
 * UART 2 on the Cortex; must be opened first using usartInit().
 */
#define uart2 ((PROS_FILE *)2)

#ifndef EOF
/**
 * EOF is a value evaluating to -1.
 */
#define EOF ((int)-1)
#endif

#ifndef SEEK_SET
/**
 * SEEK_SET is used in fseek() to denote an absolute position in bytes from the start of the
 * file.
 */
#define        SEEK_SET 0
#endif
#ifndef SEEK_CUR
/**
 * SEEK_CUR is used in fseek() to denote an relative position in bytes from the current file
 * location.
 */
#define        SEEK_CUR 1
#endif
#ifndef SEEK_END
/**
 * SEEK_END is used in fseek() to denote an absolute position in bytes from the end of the
 * file. The offset will most likely be negative in this case.
 */
#define        SEEK_END 2
#endif

/**
 * Closes the specified file descriptor. This function does not work on communication ports,
 * use usartShutdown() instead.
 *
```

```
 * @param stream the file descriptor to close from fopen()
 */
void fclose(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
 * 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 whi
 * meet this criterion.
 *
 * @param stream the stream to read (stdin, uart1, uart2, or an open file in Read mode)
 * @return the number of characters which meet this criterion; if this number cannot be
 * determined, returns 0
 */
int fcount(PROS_FILE *stream);
/**
 * 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
 * required to purge deleted files and free their allocated space for new files to be writt
 * Deleted files are still considered inaccessible to fopen() in Read mode.
 *
 * @param file the file name to erase
 * @return 0 if the file was deleted, or 1 if the file could not be found
 */
int fdelete(const char *file);
/**
 * Checks to see if the specified stream is at its end. This only works for communication p
 * and files in Read mode; for files in Write mode, 1 is always returned.
 *
 * @param stream the channel to check (stdin, uart1, uart2, or an open file in Read mode)
 * @return 0 if the stream is not at EOF, or 1 otherwise.
 */
int feof(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 flush
 * quickly as possible by the kernel.
 *
 * Successful completion of an fflush function on a file in Write mode cannot guarantee tha
 * the file is vaild until fclose() is used on that file descriptor.
 *
 * @param stream the channel to flush (an open file in Write mode)
 * @return 0 if the data was successfully flushed, EOF otherwise
 */
```

```
int fflush(PROS_FILE *stream);
/**
 * Reads and returns one character from the specified stream, blocking until complete.
 *
 * Do not use fgetc() on a VEX LCD port; deadlock may occur.
 *
 * @param stream the stream to read (stdin, uart1, uart2, or an open file in Read mode)
 * @return the next character from 0 to 255, or -1 if no character can be read
 */
int fgetc(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.
 *
 * @param str the location where the characters read will be stored
 * @param num the maximum number of characters to store; at most (num - 1) characters will b
 * read, with a null terminator ('\0') automatically appended
 * @param stream the channel to read (stdin, uart1, uart2, or an open file in Read mode)
 * @return str, or NULL if zero characters could be read
 */
char* fgets(char *str, int num, PROS_FILE *stream);
/**
 * Opens the given file in the specified mode. The file name is truncated to eight characte
 * Only four files can be in use simultaneously in any given time, with at most one of thos
 * files in Write mode. This function does not work on communication ports; use usartInit()
 * instead.
 *
 * mode can be "r" or "w". Due to the nature of the VEX Cortex memory, the "r+", "w+", and
 * 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
 * file in Write mode will create it if there is space. Opening a file that already exists
 * 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. Read
 * from files is safe, but writing to files should only be performed when robot actuators ha
 * been stopped. PROS will attempt to continue to handle events during file writes, but mos
 * user tasks cannot execute during file writing. Powering down the VEX Cortex mid-write ma
 * cause file system corruption.
 *
 * @param file the file name
 * @param mode the file mode
```

```
 * @return a file descriptor pointing to the new file, or NULL if the file could not be open
 */
PROS_FILE * fopen(const char *file, const char *mode);
/**
 * Prints the simple string to the specified stream.
 *
 * This method is much, much faster than fprintf() and does not add a new line like fputs().
 * Do not use fprint() on a VEX LCD port. Use lcdSetText() instead.
 *
 * @param string the string to write
 * @param stream the stream to write (stdout, uart1, uart2, or an open file in Write mode)
 */
void fprint(const char *string, PROS_FILE *stream);
/**
 * Writes one character to the specified stream.
 *
 * Do not use fputc() on a VEX LCD port. Use lcdSetText() instead.
 *
 * @param value the character to write (a value of type "char" can be used)
 * @param stream the stream to write (stdout, uart1, uart2, or an open file in Write mode)
 * @return the character written
 */
int fputc(int value, PROS_FILE *stream);
/**
 * Behaves the same as the "fprint" function, and appends a trailing newline ("\n").
 *
 * Do not use fputs() on a VEX LCD port. Use lcdSetText() instead.
 *
 * @param string the string to write
 * @param stream the stream to write (stdout, uart1, uart2, or an open file in Write mode)
 * @return the number of characters written, excluding the new line
 */
int fputs(const char *string, 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.
 *
 * @param ptr a pointer to where the data will be stored
 * @param size the size of each data element to read in bytes
 * @param count the number of data elements to read
 * @param stream the stream to read (stdout, uart1, uart2, or an open file in Read mode)
 * @return the number of bytes successfully read
 */
size_t fread(void *ptr, size_t size, size_t count, PROS_FILE *stream);
/**
```

```
 * Seeks within a file open in Read mode. This function will fail when used on a file in Wr
 * mode or on any communications port.
 *
 * @param stream the stream to seek within
 * @param offset the location within the stream to seek
 * @param origin the reference location for offset: SEEK_CUR, SEEK_SET, or SEEK_END
 * @return 0 if the seek was successful, or 1 otherwise
 */
int fseek(PROS_FILE *stream, long int offset, int origin);
/**
 * Returns the current position of the stream. This function works on files in either Read
 * Write mode, but will fail on communications ports.
 *
 * @param stream the stream to check
 * @return the offset of the stream, or -1 if the offset could not be determined
 */
long int ftell(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.
 *
 * @param ptr a pointer to the data to write
 * @param size the size of each data element to write in bytes
 * @param count the number of data elements to write
 * @param stream the stream to write (stdout, uart1, uart2, or an open file in Write mode)
 * @return the number of bytes successfully written
 */
size_t fwrite(const void *ptr, size_t size, size_t count, PROS_FILE *stream);
/**
 * Reads and returns one character from "stdin", which is the PC debug terminal.
 *
 * @return the next character from 0 to 255, or -1 if no character can be read
 */
int getchar();
/**
 * Prints the simple string to the debug terminal without formatting.
 *
 * This method is much, much faster than printf().
 *
 * @param string the string to write
 */
void print(const char *string);
/**
 * 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.
 *
 * @param value the character to write (a value of type "char" can be used)
 * @return the character written
 */
int putchar(int value);
/**
 * Behaves the same as the "print" function, and appends a trailing newline ("\n").
 *
 * @param string the string to write
 * @return the number of characters written, excluding the new line
 */
int puts(const char *string);

/**
 * Prints the formatted string to the specified output stream.
 *
 * The specifiers supported by this minimalistic printf() function are:
 * * @c \%d: Signed integer in base 10 (int)
 * * @c \%u: Unsigned integer in base 10 (unsigned int)
 * * @c \%x, @c \%X: Integer in base 16 (unsigned int, int)
 * * @c \%p: Pointer (void *, int *, ...)
 * * @c \%c: Character (char)
 * * @c \%s: Null-terminated string (char *)
 * * @c \%%: Single literal percent sign
 * * @c \%f: Floating-point number
 *
 * Specifiers can be modified with:
 * * @c 0: Zero-pad, instead of space-pad
 * * @c a.b: Make the field at least "a" characters wide. If "b" is specified for "%f", char
 *           number of digits after the decimal point
 * * @c -: Left-align, instead of right-align
 * * @c +: Always display the sign character (displays a leading "+" for positive numbers)
 * * @c l: Ignored for compatibility
 *
 * Invalid format specifiers, or mismatched parameters to specifiers, cause undefined behav
 * Other characters are written out verbatim. Do not use fprintf() on a VEX LCD port.
 * Use lcdPrint() instead.
 *
 * @param stream the stream to write (stdout, uart1, or uart2)
 * @param formatString the format string as specified above
 * @return the number of characters written
 */
int fprintf(PROS_FILE *stream, const char *formatString, ...);
```

```
/**
 * Prints the formatted string to the debug stream (the PC terminal).
 *
 * @param formatString the format string as specified in fprintf()
 * @return the number of characters written
 */
int printf(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 nu
 * terminator in all cases.
 *
 * @param buffer the string buffer where characters can be placed
 * @param limit the maximum number of characters to write
 * @param formatString the format string as specified in fprintf()
 * @return the number of characters stored
 */
int snprintf(char *buffer, size_t limit, 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 behat
 * occurs. See snprintf() for a safer version of this function.
 *
 * @param buffer the string buffer where characters can be placed
 * @param formatString the format string as specified in fprintf()
 * @return the number of characters stored
 */
int sprintf(char *buffer, const char *formatString, ...);

/**
 * LEFT button on LCD for use with lcdReadButtons()
 */
#define LCD_BTN_LEFT 1
/**
 * CENTER button on LCD for use with lcdReadButtons()
 */
#define LCD_BTN_CENTER 2
/**
 * RIGHT button on LCD for use with lcdReadButtons()
 */
#define LCD_BTN_RIGHT 4

/**
```

```
 * Clears the LCD screen on the specified port.
 *
 * Printing to a line implicitly overwrites the contents, so clearing should only be requir
 * at startup.
 *
 * @param lcdPort the LCD to clear, either uart1 or uart2
 */
void lcdClear(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 undefine
 * The port will not be usable with standard serial port functions until the LCD is stopped
 *
 * @param lcdPort the LCD to initialize, either uart1 or uart2
 */
void lcdInit(PROS_FILE *lcdPort);
/**
 * Prints the formatted string to the attached LCD.
 *
 * The output string will be truncated as necessary to fit on the LCD screen, 16 characters
 * wide. It is probably better to generate the string in a local buffer and use lcdSetText(
 * but this method is provided for convenience.
 *
 * @param lcdPort the LCD to write, either uart1 or uart2
 * @param line the LCD line to write, either 1 or 2
 * @param formatString the format string as specified in fprintf()
 */
#ifdef DOXYGEN
void lcdPrint(PROS_FILE *lcdPort, unsigned char line, const char *formatString, ...);
#else
void __attribute__ ((format (printf, 3, 4))) lcdPrint(PROS_FILE *lcdPort, unsigned char line
        const char *formatString, ...);
#endif
/**
 * 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.
 *
 * @param lcdPort the LCD to poll, either uart1 or uart2
 * @return the buttons pressed as a bit mask
 */
unsigned int lcdReadButtons(PROS_FILE *lcdPort);
/**
 * 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.
 *
 * @param lcdPort the LCD to adjust, either uart1 or uart2
 * @param backlight true to turn the backlight on, or false to turn it off
 */
void lcdSetBacklight(PROS_FILE *lcdPort, bool backlight);
/**
 * Prints the string buffer to the attached LCD.
 *
 * The output string will be truncated as necessary to fit on the LCD screen, 16 characters
 * wide. This function, like fprint(), is much, much faster than a formatted routine such as
 * lcdPrint() and consumes less memory.
 *
 * @param lcdPort the LCD to write, either uart1 or uart2
 * @param line the LCD line to write, either 1 or 2
 * @param buffer the string to write
 */
void lcdSetText(PROS_FILE *lcdPort, unsigned char line, const char *buffer);
/**
 * Shut down the specified LCD port.
 *
 * @param lcdPort the LCD to stop, either uart1 or uart2
 */
void lcdShutdown(PROS_FILE *lcdPort);

// -------------------- Real-time scheduler functions --------------------
/**
 * Only this many tasks can exist at once. Attempts to create further tasks will not succeed
 * until tasks end or are destroyed, AND the idle task cleans them up.
 *
 * Changing this value will not change the limit without a kernel recompile. The idle task
 * and VEX daemon task count against the limit. The user autonomous() or teleop() also count
 * against the limit, so 12 tasks usually remain for other uses.
 */
#define TASK_MAX 16
/**
 * 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.
 */
#define TASK_MAX_PRIORITIES 6
/**
 * The lowest priority that can be assigned to a task, which puts it on a level with the idl
 * task. This may cause severe performance problems and is generally not recommended.
 */
```

```
#define TASK_PRIORITY_LOWEST 0
/**
 * The default task priority, which should be used for most tasks.
 *
 * Default tasks such as autonomous() inherit this priority.
 */
#define TASK_PRIORITY_DEFAULT 2
/**
 * 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.
 */
#define TASK_PRIORITY_HIGHEST (TASK_MAX_PRIORITIES - 1)
/**
 * The recommended stack size for a new task that does an average amount of work. This stack
 * size is used for default tasks such as autonomous().
 *
 * This is probably OK for 4-5 levels of function calls and the use of printf() with several
 * arguments. Tasks requiring deep recursion or large local buffers will need a bigger stack
 */
#define TASK_DEFAULT_STACK_SIZE 512
/**
 * The minimum stack depth for a task. Scheduler state is stored on the stack, so even if th
 * 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.
 */
#define TASK_MINIMAL_STACK_SIZE        64

/**
 * Constant returned from taskGetState() when the task is dead or nonexistant.
 */
#define TASK_DEAD 0
/**
 * Constant returned from taskGetState() when the task is actively executing.
 */
#define TASK_RUNNING 1
/**
 * Constant returned from taskGetState() when the task is exists and is available to run, bu
 * not currently running.
 */
#define TASK_RUNNABLE 2
/**
 * Constant returned from taskGetState() when the task is delayed or blocked waiting for a
 * semaphore, mutex, or I/O operation.
 */
```

```
#define TASK_SLEEPING 3
/**
 * Constant returned from taskGetState() when the task is suspended using taskSuspend().
 */
#define TASK_SUSPENDED 4

/**
 * Type by which tasks are referenced.
 *
 * As this is a pointer type, it can be safely passed or stored by value.
 */
typedef void * TaskHandle;
/**
 * Type by which mutexes are referenced.
 *
 * As this is a pointer type, it can be safely passed or stored by value.
 */
typedef void * Mutex;
/**
 * Type by which semaphores are referenced.
 *
 * As this is a pointer type, it can be safely passed or stored by value.
 */
typedef void * Semaphore;
/**
 * 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);
 * }
 */
typedef void (*TaskCode)(void *);

/**
 * Creates a new task and add it to the list of tasks that are ready to run.
 *
 * @param taskCode the function to execute in its own task
 * @param stackDepth the number of variables available on the stack (4 * stackDepth bytes w
 * be allocated on the Cortex)
 * @param parameters an argument passed to the taskCode function
 * @param priority a value from TASK_PRIORITY_LOWEST to TASK_PRIORITY_HIGHEST determining th
 * initial priority of the task
 * @return a handle to the created task, or NULL if an error occurred
```

```
 */
TaskHandle taskCreate(TaskCode taskCode, const unsigned int stackDepth, void *parameters,
        const unsigned int priority);
/**
 * 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
 * scheduled if available. The calling task will still be available for immediate reschedul
 * 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().
 *
 * @param msToDelay the number of milliseconds to wait, with 1000 milliseconds per second
 */
void taskDelay(const unsigned long msToDelay);
/**
 * Delays the current task until a specified time. The task will be unblocked
 * at the time *previousWakeTime + cycleTime, and *previousWakeTime will be changed to refl
 * the time at which the task will unblock.
 *
 * If the target time is in the past, no delay occurs, but a reschedule is forced, as if
 * taskDelay() was called with an argument of zero. If the sum of cycleTime and
 * *previousWakeTime overflows or underflows, undefined behavior occurs.
 *
 * This function should be used by cyclical tasks to ensure a constant execution frequency.
 * While taskDelay() specifies a wake time relative to the time at which the function is
 * called, taskDelayUntil() specifies the absolute future time at which it wishes to unbloc
 * Calling taskDelayUntil with the same cycleTime parameter value in a loop, with
 * previousWakeTime referring to a local variable initialized to millis(), will cause the
 * loop to execute with a fixed period.
 *
 * @param previousWakeTime a pointer to the location storing the last unblock time, obtaine
 * by using the "&" operator on a variable (e.g. "taskDelayUntil(&now, 50);")
 * @param cycleTime the number of milliseconds to wait, with 1000 milliseconds per second
 */
void taskDelayUntil(unsigned long *previousWakeTime, const unsigned long cycleTime);
/**
 * 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 th
```

```
 * 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
 * freed before the task is deleted.
 *
 * @param taskToDelete the task to kill; passing NULL kills the current task
 */
void taskDelete(TaskHandle taskToDelete);
/**
 * 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 no
 * yet freed by the idle task will also be included in the count. Tasks recently created may
 * take one context switch to be counted.
 *
 * @return the number of tasks that are currently running, waiting, or suspended
 */
unsigned int taskGetCount();
/**
 * Retrieves the state of the specified task. Note that the state of tasks which have died r
 * be re-used for future tasks, causing the value returned by this function to reflect a
 * different task than possibly intended in this case.
 *
 * @param 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)
 *
 * @return A value reflecting the task's status, one of the constants TASK_DEAD, TASK_RUNNI
 * TASK_RUNNABLE, TASK_SLEEPING, or TASK_SUSPENDED
 */
unsigned int taskGetState(TaskHandle task);
/**
 * Obtains the priority of the specified task.
 *
 * @param task the task to check; passing NULL checks the current task
 * @return the priority of that task from 0 to TASK_MAX_PRIORITIES
 */
unsigned int taskPriorityGet(const TaskHandle task);
/**
 * Sets the priority of the specified task.
 *
 * A context switch may occur before the function returns if the priority being set is highe
 * than the currently executing task and the task being mutated is available to be schedule
 *
 * @param task the task to change; passing NULL changes the current task
 * @param newPriority a value between TASK_PRIORITY_LOWEST and TASK_PRIORITY_HIGHEST inclus
 * indicating the new task priority
 */
```

```
void taskPrioritySet(TaskHandle task, const unsigned int newPriority);
/**
 * Resumes the specified task.
 *
 * A task that has been suspended by one or more calls to taskSuspend() will be made availal
 * for scheduling again by a call to taskResume(). If the task was not suspended at the time
 * of the call to taskResume(), undefined behavior occurs.
 *
 * @param 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)
 */
void taskResume(TaskHandle taskToResume);
/**
 * 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 defa
 * 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 rob
 * is disabled or when the robot mode is switched.
 *
 * @param fn the function to call in this loop
 * @param increment the delay between successive calls in milliseconds; the taskDelayUntil()
 * function is used for accurate cycle timing
 * @return a handle to the task, or NULL if an error occurred
 */
TaskHandle taskRunLoop(void (*fn)(void), const unsigned long increment);
/**
 * Suspends the specified task.
 *
 * When suspended a task will not be scheduled, regardless of whether it might be otherwise
 * available to run.
 *
 * @param taskToSuspend the task to suspend; passing NULL suspends the current task
 */
void taskSuspend(TaskHandle taskToSuspend);

/**
 * Creates a semaphore intended for synchronizing tasks. To prevent some critical code from
 * simultaneously modifying a shared resource, use mutexes instead.
 *
 * Semaphores created using this function can be accessed using the semaphoreTake() and
 * semaphoreGive() functions. The mutex functions must not be used on objects of this type.
 *
 * This type of object does not need to have balanced take and give calls, so priority
```

```
 * inheritance is not used. Semaphores can be signalled by an interrupt routine.
 *
 * @return a handle to the created semaphore
 */
Semaphore semaphoreCreate();
/**
 * Signals a semaphore. Tasks waiting for a signal using semaphoreTake() will be unblocked b
 * 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.
 *
 * @param semaphore the semaphore to signal
 * @return true if the semaphore was successfully given, or false if the semaphore was not
 * taken since the last give
 */
bool semaphoreGive(Semaphore semaphore);
/**
 * 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.
 *
 * @param semaphore the semaphore to wait
 * @param blockTime the maximum time to wait for the semaphore to be given, where -1
 * specifies an infinite timeout
 * @return true if the semaphore was successfully taken, or false if the timeout expired
 */
bool semaphoreTake(Semaphore semaphore, const unsigned long blockTime);
/**
 * Deletes the specified semaphore. This function can be dangerous; deleting semaphores bein
 * waited on by a task may cause deadlock or a crash.
 *
 * @param semaphore the semaphore to destroy
 */
void semaphoreDelete(Semaphore semaphore);

/**
 * Creates a mutex intended to allow only one task to use a resource at a time. For signalli
 * and synchronization, try using semaphores.
 *
 * Mutexes created using this function can be accessed using the mutexTake() and mutexGive()
 * functions. The semaphore functions must not be used on objects of this type.
 *
 * This type of object uses a priority inheritance mechanism so a task 'taking' a mutex MUST
 * ALWAYS 'give' the mutex back once the mutex is no longer required.
 *
 * @return a handle to the created mutex
```

```
 */
Mutex mutexCreate();
/**
 * Relinquishes a mutex so that other tasks can use the resource it guards. The mutex must b
 * held by the current task using a corresponding call to mutexTake.
 *
 * @param mutex the mutex to release
 * @return true if the mutex was released, or false if the mutex was not already held
 */
bool mutexGive(Mutex mutex);
/**
 * 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.
 *
 * @param mutex the mutex to request
 * @param blockTime the maximum time to wait for the mutex to be available, where -1
 * specifies an infinite timeout
 * @return true if the mutex was successfully taken, or false if the timeout expired
 */
bool mutexTake(Mutex mutex, const unsigned long blockTime);
/**
 * Deletes the specified mutex. This function can be dangerous; deleting semaphores being
 * waited on by a task may cause deadlock or a crash.
 *
 * @param mutex the mutex to destroy
 */
void mutexDelete(Mutex mutex);

/**
 * Wiring-compatible alias of taskDelay().
 *
 * @param time the duration of the delay in milliseconds (1 000 milliseconds per second)
 */
void delay(const unsigned long time);
/**
 * 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 mo
 * more depending on CPU load. In general, this function is less reliable than delay(). Usi
 * this function in a loop may hog processing time from other tasks.
 *
 * @param us the duration of the delay in microseconds (1 000 000 microseconds per second)
 */
```

```
void delayMicroseconds(const unsigned long us);
/**
 * Returns the number of microseconds since Cortex power-up. There are 10^6 microseconds in
 * second, so as a 32-bit integer, this will overflow and wrap back to zero every two hours
 * so.
 *
 * This function is Wiring-compatible.
 *
 * @return the number of microseconds since the Cortex was turned on or the last overflow
 */
unsigned long micros();
/**
 * Returns the number of milliseconds since Cortex power-up. There are 1000 milliseconds in
 * second, so as a 32-bit integer, this will not overflow for 50 days.
 *
 * This function is Wiring-compatible.
 *
 * @return the number of milliseconds since the Cortex was turned on
 */
unsigned long millis();
/**
 * Alias of taskDelay() intended to help EasyC users.
 *
 * @param time the duration of the delay in milliseconds (1 000 milliseconds per second)
 */
void wait(const unsigned long time);
/**
 * Alias of taskDelayUntil() intended to help EasyC users.
 *
 * @param previousWakeTime a pointer to the last wakeup time
 * @param time the duration of the delay in milliseconds (1 000 milliseconds per second)
 */
void waitUntil(unsigned long *previousWakeTime, const unsigned long time);
/**
 * Enables IWDG watchdog timer which will reset the cortex if it locks up due to static sho
 * 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 manuall
 *
 * This function should only be called once in initializeIO()
 */
void watchdogInit();
/**
 * Enables the Cortex to run the op control task in a standalone mode- no VEXnet connection
 *
 * This function should only be called once in initializeIO()
 */
```

```
void standaloneModeEnable();

// End C++ extern to C
#ifdef __cplusplus
}
#endif

#endif
```