bV5

May 25, 2019

Contents:

1 Getting Started **1**

1.1 Getting Hold of bV5 1

1.2 Introduction to bV5 1

1.3 Where Next? 2

2 The bV5 Command Line **3**

2.1 Installing 3

2.2 Updating Firmare 3

2.3 Building Your Project 3

2.4 Uploading Your Project 4

2.5 Cool V5 Utilities 4

3 API Reference **5**

3.1 Global Variables 5

3.2 Robot Management 5

3.3 Controllers 6

3.4 Motors 7

3.5 Motor Reporting 11

3.6 ADI Interface 12

3.7 Robot Battery 13

3.8 Graphics 14

4 Task Management **17**

5 Examples **19**

6 Indices and tables **21**

CHAPTER 1

Getting Started

1.1 Getting Hold of bV5

Right now there are no public repositories of binaries of bV5. If you found this site, either you found a random site, or it was linked to you. In the former case, welcome, stranger. In the, more likely, latter, there should have been a template archive provided to you in the same place. Take that archive, and un-archive it. And now you're done. .

1.2 Introduction to bV5

bV5 is a fun little library I wrote for V5. Why? Because I wanted to have some fun, and I figured I might as well do something productive.

When you open your first project, you have a few files. The ones you're going to be interested in initially are `include/main.h` and the ones in `src/`.

Here's a sample from `user.c`:

```
#include "main.h"

void driver() {
    while (1) {
        /* [ SNIP  */

        /* Make sure to call delay so we can do other stuff */
        delay(20);
    }
}
```

If we have a look around, as well as `driver`, there are `auton`, `setup`, `init`, and `disabled`.

If you've been following along, you've probably noticed the all have a little comment saying what they do. Here's what they do again, anyway:

- `driver`: This is the code that is run when you are in driver control mode.

- `auton`: And this is the code for when you're in autonomous mode.
- `disabled`: This code is run when the robot is disabled.
- `setup`: This function is called before entering any of the previous three functions.
- `init`: This function is called once when the program starts. This is a good place to do things like startup background tasks, initialize sensors like ultrasonics, etc.

It's important to note that any time you have a loop that will run for a long (or infinite) time, you will need to call *`delay()`* somewhere so that the rest of the system can tick along nicely.

`main.h` is a globally included header file. This is a nice place to define things you want to be global.

1.3 Where Next?

Anywhere, really. If you want to dive right in, the API reference is just for you, if you want to learn more about how to use the command line, that section'll help, and if you're looking to run some extra tasks or want to know a little about them, then have a look at Task Management.

This is the part where I'd usually refer you over to the examples, but they're somewhat lacking because I've just spend 4 hours writing all these pages. I don't really feel like making any examples. When I do, I'll replace this.

CHAPTER 2

The bV5 Command Line

bV5 makes use of a handy handy command line program called... *bv5*! It manages all the interactions with the brain so you don't have to. It also has project management and updating.. maybe. I've not really tested that part enough, so I'm not going to suggest using it and expecting it to work :P. It's under `bv5 packages` if you want to have a peek though.

2.1 Installing

```
// TODO: This. Maybe.
```

Right now the best solution is to `git clone <the repo>` then use `pip install --user -e .` to get setup. A proper solution will probably come later.

2.2 Updating Firmare

The `bv5 dfu` command will open a firmware updater for you to use to, you guessed it, update firmware. It doesn't do anything fancy, and doesn't have any bells and whistles, but it's a darn sight smaller than the official one so is a nice way to save installing a massive firmware updater from VEX.

2.3 Building Your Project

By way of a `Makefile`, you can just run the command `make`. If you are on Windows, that command probably doesn't exist. Using `bv5 make`, it will search for a copy of `make`, then if it fails to find it, it will attempt to use the Windows Subsystem for Linux.

If you need a copy of make, or instructions for how to setup the WSL, Google is your friend.

bV5 also has a command named `bv5 make watch`, which will monitor your project source, and then re-compile it when anything changes. It can also optionally upload the new binary to the brain. Use `--help` for more information on that.

2.4 Uploading Your Project

So you've compiled your program, ey? `bv5 upload`! As above, use `bv5 upload --help` for more information about the options you can pass to this command.

2.5 Cool V5 Utilities

There's a lot of cool stuff in `bv5 v5`. Here are some of them:

- `erase`: Clear your brain
- `name [robot name]`: Set or get the robot's name
- `team [team name]`: Set or get the robot's team
- `tree`: List all the files on the robot.
- `speedtest`: Test the connection speed. This is useful for wireless uploading (when you plug into the controller instead of the brain)
- `term`: Open a serial terminal to the brain. You can use `printf()` and `getchar()` to communicate from the brain.
- `set <variable> <value>`/`get <variable>`: Modify system variables. These are general purpose storage spaces on the brain. Fun fact: They persist even after a factory reset of the brain!
- `read <file>`: Download a file off the brain. It's like `upload`, but cooler!
- `last`: Ever wondered when the last time you used bV5 on a robot brain was? Of course not, but hey, there's a command for it.

`bv5 v5` is where I usually put random things, so it's worth running `bv5 v5 --help` if you get bored, to see if I've added anything fun :).

CHAPTER 3

API Reference

3.1 Global Variables

uint32_t **systemVersion**
The current version of VEXos

uint32_t **stdlibVersion**
The current version of the stdlib that is being used

uint32_t **sdkVersion**
The current version of the sdk that is being used

uint64_t **startupTime**
The time that the system was powered on

3.2 Robot Management

void **delay** (uint32_t *ms*)
Delay the currently running task for a period of time

Parameters

- **ms** – The number of milliseconds to delay (or 0 for as short as possible)

bool **isDriver** ()
Check if we're in driver control or not

bool **isAuton** ()
Check if we're in auton mode or not

bool **isCompetition** ()
This will be true when we are in competiton mode (either field control or a testing controller are connected).

bool **fieldConnected** ()
This will be true if we are connected to a field, but false if we are at inspection or using a testing controller.

Warning: This **clearly** has a lot of room for abuse. Do what you will with it, but it's not my fault if you do something stupid and get DQ'd.

void **`exit`** (int *code*)

Quit the program.

Parameters

- **`code`** – The exit code. This isn't actually used, just added for compatability with the normal `exit` function.

uint32_t **`getMillis`** ()

Get the current time in milliseconds

uint64_t **`getMicros`** ()

Get the current time in microseconds

uint32_t **`getUsbStatus`** ()

Get the USB connection status. The return value is a bit field, as described below:

- Bit 1 (1): Is there a cabled plugged into the brain?
- Bit 2 (2): Is there an established connection from the brain cable to a host computer?
- Bit 3 (4): Is there a cabled plugged into the controller?
- Bit 4 (8): Is there an established connection from the controller cable to a host computer?

Bits 5 (16) and 6 (32) are possibly used like bit 3 and 4 but for the partner controller, however I lack a second controller. It's worth noting that a connection from the controller **and** the brain at the same time is possible.

3.3 Controllers

int32_t **`controllerGet`** (ControllerId *id*, ControllerChan *channel*)

Query a "channel" from a controller. This usually just means getting a joystick value or a button, but you can also use it to get the current battery level of the controller, the power button in the middle of the controller, etc.

Parameters

- **`id`** – The controller to query. Either `Master` or `Partner`.
- **`channel`** – The controller channel to query. This is one of:
  - `LeftX`
  - `LeftY`
  - `RightX`
  - `RightY`
  - `BtnL1`
  - `BtnL2`
  - `BtnR1`
  - `BtnR2`
  - `BtnUp`
  - `BtnDown`
  - `BtnLeft`
  - `BtnRight`
  - `BtnX`
  - `BtnY`
  - `BtnA`

  - `BtnB`
  - `BtnSel`
  - `BattLevel`
  - `BtnAll`
  - `Flags`
  - `BattCapacity`

ControllerStatus **`controllerStatus`** (ControllerId *id*)

Get the status of a controller. The return value is either:

- `Offline`
- `Tethered`
- `VEXnet`

This function does not differentaite between a VEXnet connection, or a Bluetooth connection from the robot radio.

Parameters

- **`id`** – The controller to query. Either `Master` or `Partner`.

bool **`setControllerText`** (ControllerId *id*, uint32_t *line*, uint32_t *col*, const char **str*)

Set a line of text on the controller. The return value indicates success.

Warning: This function must never be called more than once every 50ms due to limitations in how the controller works.

Parameters

- **`id`** – The controller to query. Either `Master` or `Partner`.
- **`line`** – The line on the controller to set.
- **`col`** – The column to start writing at.
- **`str`** – A pointer to a char array, containing the string.

bool **`clearControllerLine`** (ControllerId *id*, uint8_t *line*)

This is a helper function to fill a line of text on the controller with spaces, effectively clearing that line.

Parameters

- **`id`** – The controller to query. Either `Master` or `Partner`.
- **`line`** – The line on the controller to clear.

bool **`controllerRumble`** (ControllerId *id*, const char **pattern*)

Make the controller V I B R A T E!!

Parameters

- **`id`** – The controller to query. Either `Master` or `Partner`.
- **`pattern`** – A pointer to a string containing the vibration pattern.

3.4 Motors

Note: All of these functions assume that a motor has been plugged into the specified port. If something other than a motor is plugged in unexpected magic™ might happed!

void **`velocitySet`** (uint32_t *port*, int32_t *velocity*)
Set the velocity of a motor. This will reset the motor's internal PID tracking, and is probably what you want.

Parameters

- **`port`** – The port from 1-21 for the motor
- **`velocity`** – The velocity to set the motor to

void **`velocityUpdate`** (uint32_t *port*, int32_t *velocity*)
Update the target velocity for a motor's internal PID without resetting the state.

Parameters

- **`port`** – The port from 1-21 for the motor
- **`velocity`** – The velocity to update the motor to

void **`voltageSet`** (uint32_t *port*, int32_t *voltage*)
Set the voltage to a motor. **When using this function, the internal PID for the motor will be ignored.**

Parameters

- **`port`** – The port from 1-21 for the motor
- **`voltage`** – The velocity to send to the motor (in mV)

int32_t **`velocityGet`** (uint32_t *port*)
Get the current velocity that the motor is attempting to reach.

Parameters

- **`port`** – The port from 1-21 for the motor

int32_t **`directionGet`** (uint32_t *port*)
Get the current direction of travel for the given motor.

Parameters

- **`port`** – The port from 1-21 for the motor

int32_t **`velocityGetReal`** (uint32_t *port*)
Get the real velocity that a given motor is running at.

Parameters

- **`port`** – The port from 1-21 for the motor

void **`pwmSet`** (uint32_t *port*, int32_t *duty*)
Control a motor using a PWM duty cycle instead of voltage or velocity.

Parameters

- **`port`** – The port from 1-21 for the motor
- **`duty`** – The duty cycle the motor should be run at.

int32_t **`pwmGet`** (uint32_t *port*)
Get the current PWM duty cycle that a motors is being driven at.

Parameters

- **`port`** – The port from 1-21 for the motor

void **`currentLimitSet`** (uint32_t *port*, int32_t *limit*)
Set the current draw limit for a motor.

Parameters

- **`port`** – The port from 1-21 for the motor
- **`limit`** – The limit that should be set for the motor.

int32_t **`currentLimitGet`** (uint32_t *port*)
Get the current draw limit for a motor.

Parameters

- **`port`** – The port from 1-21 for the motor

void **`voltageLimitSet`** (uint32_t *port*, int32_t *limit*)
Set the voltage limit for a motor.

Parameters

- **`port`** – The port from 1-21 for the motor
- **`limit`** – The voltage that should be set for the motor.

int32_t **`voltageLimitGet`** (uint32_t *port*)
Get the voltage limit for a motor.

Parameters

- **`port`** – The port from 1-21 for the motor

void **`setEncoderUnits`** (uint32_t *port*, EncoderUnits *units*)
Set the units that should be reported by *`getMotorPos()`*.

Parameters

- **`port`** – The port from 1-21 for the motor
- **`units`** – The units to use. One of:
  - `Degrees`
  - `Rotations`
  - `Counts`

EncoderUnits **`getEncoderUnits`** (uint32_t *port*)
Get the units that are being reported by *`getMotorPos()`*. See also *`setEncoderUnits()`*.

Parameters

- **`port`** – The port from 1-21 for the motor

void **`setBrake`** (uint32_t *port*, BrakeMode *mode*)
Set the brake mode that the motor will use when stopping.

Note: The brake mode is not respected when setting the voltage directly instead of using velocity control.

Parameters

- **`port`** – The port from 1-21 for the motor
- **`mode`** – The brake mode to set the motor to. One of:
  - `BrakeCoast`

    – `BrakeBrake`
    – `BrakeHold`

BrakeMode **`getBrake`** (uint32_t *port*)
Get the brake mode that the motor is currently using.

Parameters

- **`port`** – The port from 1-21 for the motor

void **`setMotorPos`** (uint32_t *port*, double *position*)
Set the position of a given motor.

Parameters

- **`port`** – The port from 1-21 for the motor
- **`position`** – The position the motor should be set to

double **`getMotorPos`** (uint32_t *port*)
Get the position of a given motor.

Parameters

- **`port`** – The port from 1-21 for the motor

void **`resetMotorPos`** (uint32_t *port*)
Reset the position counter of a given motor to 0.

Parameters

- **`port`** – The port from 1-21 for the motor

double **`getTarget`** (uint32_t *port*)
Get the target that the motor PID is trying to achieve.

Parameters

- **`port`** – The port from 1-21 for the motor

void **`setServo`** (uint32_t *port*, double *position*)
Set the position of a given motor, as if it were acting as a servo instead of a continuous motor.

Parameters

- **`port`** – The port from 1-21 for the motor
- **`position`** – The postition that the motor should seek to

void **`targetSetAbs`** (uint32_t *port*, double *position*, int32_t *velocity*)
Set the absolute target of a motor.

Parameters

- **`port`** – The port from 1-21 for the motor
- **`position`** – The absolute position that the motor should target.
- **`velocity`** – The velocity at which the motor should move to that target.

void **`targetSetRel`** (uint32_t *port*, double *position*, int32_t *velocity*)
Set the target of a motor, relative to its current location.

Parameters

- **`port`** – The port from 1-21 for the motor

- **`position`** – The relative position that the motor should target.
- **`velocity`** – The velocity at which the motor should move to that target.

void **`setGears`** (uint32_t *port*, Gearset *gears*)

Set the gear ration that the motor is using, used for when trying to control the motor in terms of rotations or velocity. It is a good idea to reset the known gearing of every motor using this function in the `setup()` or `init()` function of user code.

Parameters

- **`port`** – The port from 1-21 for the motor
- **`gears`** – The gearset to set the motor to. One of:
  - `Gears36`
  - `Gears18`
  - `Gears06`

Gearset **`getGears`** (uint32_t *port*)

Get the current gearset that a given motor is set to. This might not be the actual gearset installed in the motor! This is just the gearset that has been set at some point previously.

Parameters

- **`port`** – The port from 1-21 for the motor

3.5 Motor Reporting

The following functions report some form of status for a motor. They can be useful when diagnosing problems, or maybe you could use them in your workflow.

int32_t **`motorCurrent`** (uint32_t *port*)

Get the current that a motor is currently drawing.

Parameters

- **`port`** – The port from 1-21 for the motor

int32_t **`motorVoltage`** (uint32_t *port*)

Get the current voltage of a given motor.

Parameters

- **`port`** – The port from 1-21 for the motor

double **`motorPower`** (uint32_t *port*)

Get the current power for a given motor.

Parameters

- **`port`** – The port from 1-21 for the motor

double **`motorEfficiency`** (uint32_t *port*)

Get the current efficiency for a given motor.

Parameters

- **`port`** – The port from 1-21 for the motor

double **`motorTemp`** (uint32_t *port*)

Get the current temperature for a given motor.

Parameters

- **`port`** – The port from 1-21 for the motor

bool **`isOverTemp`** (uint32_t *port*)

Checks if a motor is exceeding its safe operating temperature. If a motor is doing this, you might want to cool it down :).

Parameters

- **`port`** – The port from 1-21 for the motor

bool **`isOverCurrent`** (uint32_t *port*)

Checks if a motor is drawing too much current. If this happens, you could increase the maximum, or check if there's a mechanical fault jamming the mechanism.

Parameters

- **`port`** – The port from 1-21 for the motor

uint32_t **`getFaults`** (uint32_t *port*)

Get faults for a particular motor. Your motors shouldn't usually fault, so if they do that's probably bad.

Parameters

- **`port`** – The port from 1-21 for the motor

3.6 ADI Interface

3.6.1 Genric ADI Devices

void **`setupAdi`** (uint32_t *port*, AdiType *type*)

Set the type of device connected to a 3-wire port

Parameters

- **`port`** – The 3-wire from 'a'-'h' (or 1-8)
- **`type`** – The type of device connected. One of:
  - `AnalogIn`
  - `AnalogOut`
  - `DigitalIn`
  - `DigitalOut`
  - `SmartButton`
  - `SmartPot`
  - `LegacyButton`
  - `LegacyPotentiometer`
  - `LegacyLineSensor`
  - `LegacyLightSensor`
  - `LegacyGyro`
  - `LegacyAccelerometer`
  - `LegacyServo`
  - `LegacyPwm`
  - `QuadEncoder`
  - `Sonar`
  - `LegacyPwmSlew`

void **adiSet** (uint32_t *port*, int32_t *value*)
Set the output value of a 3-write port. What this does is dependant on how `setupAdi()` was used.

Parameters

- **port** – The 3-wire from 'a'-'h' (or 1-8)
- **value** – The value to set the port to.

void **adiGet** (uint32_t *port*)
Reads a value from the 3-wire interface. As with `adiSet()`, this is heavily dependant on `setupAdi()`.

Parameters

- **port** – The 3-wire from 'a'-'h' (or 1-8)

3.6.2 Ultrasonics

ultra_t **setupUltra** (uint32_t *ping*, uint32_t *echo*)
Sets up an ultrasonic (sonar) sensor on two ports.

Note: These two ports must be either AB, CD, EF or GH. Any other combination will fail to work.

Parameters

- **ping** – The 3-wire port the ping wire is in
- **echo** – The 3-wire port the echo wire is in

uint32_t **ultraGet** (ultra_t *ultra*)
Read the value from an ultrasonic sensor.

Parameters

- **ultra** – An ultrasonic sensor, as returned by `setupUltra()`.

void **ultraStop** (ultra_t *ultra*)
Stop using a pair of ports as an ultrasonic sensor.

Parameters

- **ultra** – An ultrasonic sensor, as returned by `setupUltra()`.

3.7 Robot Battery

If, for some reason, you are powering your robot brain through means other than the official Robot Battery, these functions will return 0, however this is undefined behaviour and should not be relied on.

int32_t **batteryVoltage** ()
Get the current voltage of the connected robot battery.

int32_t **batteryCurrent** ()
Get the current current draw on the connected robot battery from the brain.

double **betteryTemp** ()
Get the temperature of the connected robot battery.

double **betteryCapacity**()
: Get the capacity of the connected robot battery.

3.8 Graphics

void **foregroundColor**(uint32_t *col*)
: Set the foreground colour to use in the following drawing functions.

  Parameters

  - **col** – The colour to use.

void **backgroundColor**(uint32_t *col*)
: Set the background colour to use in the following drawing functions.

  Parameters

  - **col** – The colour to use.

void **clearDisplay**()
: Clear the display

void **printfDisplay**(int32_t *xpos*, int32_t *ypos*, uint32_t *opacity*, const char **format*, ...)
: Put text onto the display. See also `setFont()`.

  Parameters

  - **xpos** – The X position on the display
  - **ypos** – The Y position on the display
  - **opacity** – The opacity of the text being rendered
  - **format** – A printf-style format string

void **drawLine**(int32_t *x1*, int32_t *y1*, int32_t *x2*, int32_t *y2*)
: Draw a line onto the display.

  Parameters

  - **x1** – The starting X coordinate
  - **y1** – The starting Y coordinate
  - **x2** – The ending X coordinate
  - **y2** – The ending Y coordinate

void **drawRect**(int32_t *x1*, int32_t *y1*, int32_t *x2*, int32_t *y2*)
: Draw a rectangle's outline onto the display.

  Parameters

  - **x1** – The X coordinate of the top left
  - **y1** – The Y coordinate of the top left
  - **x2** – The X coordinate of the bottom right
  - **y2** – The Y coordinate of the bottom right

void **fillRect**(int32_t *x1*, int32_t *y1*, int32_t *x2*, int32_t *y2*)
: Draw a filled rectangle onto the display.

  Parameters

- **`x1`** – The X coordinate of the top left
- **`y1`** – The Y coordinate of the top left
- **`x2`** – The X coordinate of the bottom right
- **`y2`** – The Y coordinate of the bottom right

void **`drawCircle`** (int32_t *xc*, int32_t *yc*, int32_t *radius*)
Draw the outline of a circle onto the display.

Parameters

- **`xc`** – The X coordinate of the centre
- **`yc`** – The Y coordinate of the centre
- **`radius`** – The radius of the circle to draw

void **`fillCircle`** (int32_t *xc*, int32_t *yc*, int32_t *radius*)
Draw a filled circle onto the display.

Parameters

- **`xc`** – The X coordinate of the centre
- **`yc`** – The Y coordinate of the centre
- **`radius`** – The radius of the circle to draw

void **`setAt`** (uint32_t *x*, uint32_t *y*)
Set a single pixel on the display.

Parameters

- **`x`** – The X coordinate
- **`y`** – The Y coordinate

bool **`setFont`** (FontFace *face*)
Set the font that should be used when putting text onto the display.

Parameters

- **`face`** – The font face to use. One of:
  - `Monospace`
  - `Proportional`

CHAPTER 4

Task Management

bV5 currently uses a cooperative scheduler to run tasks. This means you have to **cooperate** with the code. If you never call `delay()`, the system will never get a chance to think, and crash, so make sure you call it!

Mutex **`new_mutex`**()

Create a new mutex that can be locked and unlocked.

void **`take_mutex`**(Mutex *mutex*)

Lock a mutex. This will prevent other tasks that also want to lock it from running until you release it (or the contray, if it's already locked).

Parameters

- **`mutex`** – The mutex that should be locked.

void **`release_mutex`**(Mutex *mutex*)

Unlock a mutex. This allows other tasks to lock the mutex again.

Parameters

- **`mutex`** – The mutex that should be unlocked.

tid_t **`start_new_task`**(void (**taskf*)(void), uint8_t *priority*)

Create a new task, and start it. It's best explained with an example:

```
void my_task() {
    while (1) {
        delay(10);  // Do nothing!
    }
}

tid_t my_task = start_new_task(&my_task, 5);
```

The slightly odd name `tid_t` is just short for "Task ID Type".

Parameters

- **`taskf`** – A pointer to a task function.

- **`priority`** – The task priority. This should probably be between 4 and 10. Lower priority gets precedence.

void **`swap_task`**()

Releases execution to the scheduler. This allows another taks to run. This is essentially the same as using `delay(0);`

void **`stop_current_task`**()

Kills the current task. This function will never return.

void **`stop_task`**(tid_t *task_id*)

Stops and cleans up given task.

Parameters

- **`task_id`** – The `tid_t` returned from *`start_new_task()`*

CHAPTER 5

Examples

Idk. DM me on Discord or smth.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

A

adiGet (*C function*), 13
adiSet (*C function*), 12

B

backgroundColor (*C function*), 14
batteryCurrent (*C function*), 13
batteryVoltage (*C function*), 13
betteryCapacity (*C function*), 13
betteryTemp (*C function*), 13

C

clearControllerLine (*C function*), 7
clearDisplay (*C function*), 14
controllerGet (*C function*), 6
controllerRumble (*C function*), 7
controllerStatus (*C function*), 7
currentLimitGet (*C function*), 9
currentLimitSet (*C function*), 8

D

delay (*C function*), 5
directionGet (*C function*), 8
drawCircle (*C function*), 15
drawLine (*C function*), 14
drawRect (*C function*), 14

E

exit (*C function*), 6

F

fieldConnected (*C function*), 5
fillCircle (*C function*), 15
fillRect (*C function*), 14
foregroundColor (*C function*), 14

G

getBrake (*C function*), 10
getEncoderUnits (*C function*), 9
getFaults (*C function*), 12
getGears (*C function*), 11
getMicros (*C function*), 6
getMillis (*C function*), 6
getMotorPos (*C function*), 10
getTarget (*C function*), 10
getUsbStatus (*C function*), 6

I

isAuton (*C function*), 5
isCompetition (*C function*), 5
isDriver (*C function*), 5
isOverCurrent (*C function*), 12
isOverTemp (*C function*), 12

M

motorCurrent (*C function*), 11
motorEfficiency (*C function*), 11
motorPower (*C function*), 11
motorTemp (*C function*), 11
motorVoltage (*C function*), 11

N

new_mutex (*C function*), 17

P

printfDisplay (*C function*), 14
pwmGet (*C function*), 8
pwmSet (*C function*), 8

R

release_mutex (*C function*), 17
resetMotorPos (*C function*), 10

S

sdkVersion (*C variable*), 5
setAt (*C function*), 15
setBrake (*C function*), 9
setControllerText (*C function*), 7

setEncoderUnits (*C function*), 9
setFont (*C function*), 15
setGears (*C function*), 11
setMotorPos (*C function*), 10
setServo (*C function*), 10
setupAdi (*C function*), 12
setupUltra (*C function*), 13
start_new_task (*C function*), 17
startupTime (*C variable*), 5
stdlibVersion (*C variable*), 5
stop_current_task (*C function*), 18
stop_task (*C function*), 18
swap_task (*C function*), 18
systemVersion (*C variable*), 5

T

take_mutex (*C function*), 17
targetSetAbs (*C function*), 10
targetSetRel (*C function*), 10

U

ultraGet (*C function*), 13
ultraStop (*C function*), 13

V

velocityGet (*C function*), 8
velocityGetReal (*C function*), 8
velocitySet (*C function*), 8
velocityUpdate (*C function*), 8
voltageLimitGet (*C function*), 9
voltageLimitSet (*C function*), 9
voltageSet (*C function*), 8