wpi-32u4-library

Generated by Doxygen 1.9.3

Chapter 1

wpi-32u4-library

Version: 3.0.0
Release date: 2021-12-12
Forked from www.pololu.com

1.1 Summary

This is a C++ library for the Arduino IDE that helps access the on-board hardware of the Romi 32U4 Control Board.

The Romi 32U4 Control Board turns the Romi chassis into a programmable, Arduino-compatible robot. It has an integrated AVR ATmega32U4 microcontroller, motor drivers, encoders, buzzer, buttons, and an LSM6DS33 accelerometer and gyro.

This library includes code for accessing the LSM6DS33, forked from the separate LSM6 library.

1.2 Installing the library (platformio)

Add the following lines to your platformio.ini file:

```
lib_deps =
     Wire
     wpi-32u4-library
```

You're done.

(You may or may not need to add Wire – it can't hurt if you do.)

1.3 Classes and functions

The main classes and functions provided by the library are listed below:

- Romi32U4ButtonA
- Romi32U4ButtonB
- Romi32U4ButtonC
- ~~Romi32U4Buzzer~~
- Romi32U4Encoders
- ~~Romi32U4LCD~~
- Romi32U4Motors
- ledRed()
- ledGreen()
- ledYellow()
- usbPowerPresent()
- readBatteryMillivolts()

1.4 Component libraries

This library also includes copies of several other Arduino libraries inside it which are used to help implement the classes and functions above.

- `FastGPIO`
- ~~[PololuBuzzer]~~(`https://github.com/pololu/pololu-buzzer-arduino`)
- ~~[PololuHD44780]~~(`https://github.com/pololu/pololu-hd44780-arduino`)
- `Pushbutton`
- `QTRSensors`
- `USBPause`

You can use these libraries in your sketch automatically without any extra installation steps and without needing to add any extra `#include` lines to your sketch.

You should avoid adding extra `#include` lines such as `#include <Pushbutton.h>` because then the Arduino IDE might try to use the standalone Pushbutton library (if you previously installed it), and it would conflict with the copy of the Pushbutton code included in this library.

1.5 Documentation

Documentation found at https://wpiroboticsengineering.github.io/wpi-32u4-library/index.html. We are working on updating documentation for the changes to the forked Pololu library. The biggest change is to the motor class, where
`Romi32U4Motors::setSpeeds()`

has been changed to
`Romi32U4Motors::setEfforts()`

since that better represents the behavior of that function.

Some other library files (LCD, buzzer) have been removed, since they conflict with some of the changes in the background (mostly to timers).

1.6 Version history

- 2.6.0 (2021-04-04): Added in a library and example for the IR Positionn sensor from DF Robot. Also includes improvements to some libraries. Now includes a Timer class for software timers.
- 2.5.0 (2021-01-30): Previous versions add IR remote with interrupts handled automatically (so long as you use either an external interrupt pin or PCINT). IMU updates.
- 2.1.0 (2020-09-06): Primary release for WPI courses.
- 1.0.2 (2017-07-17): Fixed a bug that caused errors for the right encoder to be reported as errors for the left encoder.
- 1.0.1 (2017-02-23):
  - Changed the internal `Romi32U4Motors::maxSpeed` variable to be an `int16_t` so it can be compared to other `int16_t` variables without warnings.
  - Fixed the InterialSensors and Demo examples to not use a compass.
  - Fixed some comments.
- 1.0.0 (2017-02-06): Original release.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

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

Chapter 5

Class Documentation

5.1 A Class Reference

5.1.1 Detailed Description

controller.

If errorBound is non-zero, the integral will be capped at that value.

to control a servo on pin 5.

Servo32U4 uses output compare on Timer3 to control the pulse to the servo. The 16-bit Timer3 is set up with a pre-scaler of 8, TOP of 39999 + 1 => 20 ms interval.

OCR3A controls the pulse on pin 5 – THE SERVO MUST BE ON PIN 5!

Defaults to a range of 1000 - 2000 us, but can be customized.

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

- src/PIDcontroller.h

5.2 Chassis Class Reference

```
#include <Chassis.h>
```

Public Member Functions

- Chassis (float wheelDiam, float ticksPerRevolution, float wheelTrack)

  Chassis constructor.
- void init (void)

  Initializes the chassis. Must be called in setup().
- void **setMotorPIDcoeffs** (float kp, float ki)

  Sets PID coefficients for the motors. Not independent.
- void idle (void)

  Idles chassis. Motors will stop.
- void setMotorEfforts (int leftEffort, int rightEffort)

  Sets motor efforts. Max speed is 420.
- void setWheelSpeeds (float leftSpeed, float rightSpeed)

  Sets target wheel speeds in cm/sec.
- void setTwist (float forwardSpeed, float turningSpeed)

  Sets target motion for the chassis.
- void driveFor (float forwardDistance, float forwardSpeed)

  Commands the robot to drive at a distance and speed.
- void turnFor (float turnAngle, float turningSpeed)

  Commands the chassis to turn a set angle.
- bool checkMotionComplete (void)

  Checks if the motion commanded by driveFor() or turnFor() is done.
- void **printSpeeds** (void)
- void **updateEncoderDeltas** ()

Public Attributes

- LeftMotor **leftMotor**
- RightMotor **rightMotor**

Protected Attributes

- const float **cmPerEncoderTick**
- const float **robotRadius**
- const uint16_t **ctrlIntervalMS** = 16

5.2.1 Detailed Description

The Chassis class manages the motors and encoders.

Chassis sets up a hardware-based timer on a 16ms interval. At each interrupt, it reads the current encoder counts and, if controlling for speed, calculates the effort using a PID controller for each motor, which can be adjusted by the user.

The encoders are attached automatically and the encoders will count regardless of the state of the robot.

Several methods are provided for low level control to commands for driving or turning.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Chassis()

```
Chassis::Chassis (
            float wheelDiam,
            float ticksPerRevolution,
            float wheelTrack )  [inline]
```

Chassis constructor.

Parameters

wheelDiam	Wheel diameter in cm.
ticksPerRevolution	Enccoder ticks per *wheel* revolution.
wheelTrack	Distance between wheels in cm.

5.2.3 Member Function Documentation

5.2.3.1 checkMotionComplete()

```
bool Chassis::checkMotionComplete (
            void  )
```

Checks if the motion commanded by driveFor() or turnFor() is done.

Returns

Returns true if the motion is complete.

5.2.3.2 driveFor()

```
void Chassis::driveFor (
            float forwardDistance,
            float forwardSpeed )
```

Commands the robot to drive at a distance and speed.

The chassis will stop when the distance is reached.

Parameters

forwardDistance	Target distance in cm
forwardSpeed	Target speed rate in cm/sec

5.2.3.3 idle()

```
void Chassis::idle (
            void  )
```

Idles chassis. Motors will stop.

Stops the motors. It calls setMotorEfforts() so that the wheels won't lock. Use setSpeeds() if you want the wheels to 'lock' in place.

5.2.3.4 init()

```
void Chassis::init (
            void  )
```

Initializes the chassis. Must be called in setup().

Call init() in your setup() routine. It sets up some internal timers so that the speed controllers for the wheels will work properly.

Here's how it works: Motor::init() starts a hardware timer on a 16 ms loop. Every time the timer "rolls over," an interrupt service routine (ISR) is called that updates the motor speeds and sets a flag to notify Chassis that it is time to calculate the control inputs.

When set up this way, pins 6, 12, and 13 cannot be used with analogWrite()

5.2.3.5 setMotorEfforts()

```
void Chassis::setMotorEfforts (
            int leftEffort,
            int rightEffort )
```

Sets motor efforts. Max speed is 420.

Parameters

leftEffort	Effort for left motor
rightEffort	Effort for right motor

Sets the motor *efforts*.

5.2.3.6 setTwist()

```
void Chassis::setTwist (
            float forwardSpeed,
            float turningSpeed )
```

Sets target motion for the chassis.

Parameters

forwardSpeed	Target forward speed in cm/sec
rightSpeed	Target spin rate in deg/sec

5.2.3.7 setWheelSpeeds()

```
void Chassis::setWheelSpeeds (
            float leftSpeed,
            float rightSpeed )
```

Sets target wheel speeds in cm/sec.

Parameters

leftSpeed	Target speed for left wheel in cm/sec
rightSpeed	Target speed for right wheel in cm/sec

5.2.3.8 turnFor()

```
void Chassis::turnFor (
            float turnAngle,
            float turningSpeed )
```

Commands the chassis to turn a set angle.

Parameters

turnAngle	Target angle to turn in degrees
turningSpeed	Target spin rate in deg/sec

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

- src/Chassis.h
- src/Chassis.cpp

5.3 IRDecoder Class Reference

Public Member Functions

- **IRDecoder** (uint8_t p)
- void **init** (void)
- void **handleIRsensor** (void)
- uint32_t **getCode** (void)
- int16_t **getKeyCode** (bool acceptRepeat=false)

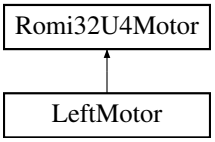
The documentation for this class was generated from the following files:

- src/IRdecoder.h
- src/IRdecoder.cpp

5.4 LeftMotor Class Reference

`#include <Romi32U4Motors.h>`

Inheritance diagram for LeftMotor:



Public Member Functions

- void **setMotorEffort** (int16_t effort)

Protected Member Functions

- void setEffort (int16_t effort)

Additional Inherited Members

5.4.1 Detailed Description

Two derived classes, one for each motor. With the way Pololu controls the speeds, this avoids ugly lookup tables (though it's not beautiful itself). Friended to Chassis so that setEffort() can't be called from main(). Use Chassis::setMotorEfforts() to set efforts, as that will adjust the control mode properly.

5.4.2 Member Function Documentation

5.4.2.1 setEffort()

```
void LeftMotor::setEffort (
            int16_t effort )  [protected], [virtual]
```

Because the Pololu library is based on the FastGPIO library, we don't/can't use analogWrite. Instead, we set the duty cycle directly at the register level.

We also have to have separate classes for the left and right motors to avoid the complex mapping of speeds to registers.

Implements Romi32U4Motor.

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

- src/Romi32U4Motors.h
- src/Romi32U4Motors.cpp

5.5 LSM6 Class Reference

Classes

- struct vector

Public Types

- enum **deviceType** { **device_DS33** , **device_auto** }
- enum **sa0State** { **sa0_low** , **sa0_high** , **sa0_auto** }
- enum **ACC_FS** { **ACC_FS2** , **ACC_FS4** , **ACC_FS8** , **ACC_FS16** }
- enum **GYRO_FS** { **GYRO_FS245** , **GYRO_FS500** , **GYRO_FS1000** , **GYRO_FS2000** }
- enum **ODR** {
  ODR13 = 0x1 , **ODR26** = 0x2 , **ODR52** = 0x3 , **ODR104** = 0x4 ,
  ODR208 = 0x5 , **ODR416** = 0x6 , **ODR833** = 0x7 , **ODR166k** = 0x8 }
- enum **regAddr** {
  FUNC_CFG_ACCESS = 0x01 , **FIFO_CTRL1** = 0x06 , **FIFO_CTRL2** = 0x07 , **FIFO_CTRL3** = 0x08 ,
  FIFO_CTRL4 = 0x09 , **FIFO_CTRL5** = 0x0A , **ORIENT_CFG_G** = 0x0B , **INT1_CTRL** = 0x0D ,
  INT2_CTRL = 0x0E , **WHO_AM_I** = 0x0F , **CTRL1_XL** = 0x10 , **CTRL2_G** = 0x11 ,
  CTRL3_C = 0x12 , **CTRL4_C** = 0x13 , **CTRL5_C** = 0x14 , **CTRL6_C** = 0x15 ,
  CTRL7_G = 0x16 , **CTRL8_XL** = 0x17 , **CTRL9_XL** = 0x18 , **CTRL10_C** = 0x19 ,
  WAKE_UP_SRC = 0x1B , **TAP_SRC** = 0x1C , **D6D_SRC** = 0x1D , **STATUS_REG** = 0x1E ,
  OUT_TEMP_L = 0x20 , **OUT_TEMP_H** = 0x21 , **OUTX_L_G** = 0x22 , **OUTX_H_G** = 0x23 ,
  OUTY_L_G = 0x24 , **OUTY_H_G** = 0x25 , **OUTZ_L_G** = 0x26 , **OUTZ_H_G** = 0x27 ,
  OUTX_L_XL = 0x28 , **OUTX_H_XL** = 0x29 , **OUTY_L_XL** = 0x2A , **OUTY_H_XL** = 0x2B ,
  OUTZ_L_XL = 0x2C , **OUTZ_H_XL** = 0x2D , **FIFO_STATUS1** = 0x3A , **FIFO_STATUS2** = 0x3B ,
  FIFO_STATUS3 = 0x3C , **FIFO_STATUS4** = 0x3D , **FIFO_DATA_OUT_L** = 0x3E , **FIFO_DATA_OUT_H** = 0x3F ,
  TIMESTAMP0_REG = 0x40 , **TIMESTAMP1_REG** = 0x41 , **TIMESTAMP2_REG** = 0x42 , **STEP_↩ TIMESTAMP_L** = 0x49 ,
  STEP_TIMESTAMP_H = 0x4A , **STEP_COUNTER_L** = 0x4B , **STEP_COUNTER_H** = 0x4C , **FUNC_SRC** = 0x53 ,
  TAP_CFG = 0x58 , **TAP_THS_6D** = 0x59 , **INT_DUR2** = 0x5A , **WAKE_UP_THS** = 0x5B ,
  WAKE_UP_DUR = 0x5C , **FREE_FALL** = 0x5D , **MD1_CFG** = 0x5E , **MD2_CFG** = 0x5F }

Public Member Functions

- bool **init** (deviceType device=device_auto, sa0State sa0=sa0_auto)
- deviceType **getDeviceType** (void)
- void **enableDefault** (void)
- void **writeReg** (uint8_t reg, uint8_t value)
- uint8_t **readReg** (uint8_t reg)
- void **readAcc** (void)
- void **readGyro** (void)
- void **read** (void)
- void **setFullScaleGyro** (GYRO_FS gfs)
- void **setFullScaleAcc** (ACC_FS afs)
- void **setGyroDataOutputRate** (ODR)
- void **setAccDataOutputRate** (ODR)
- void **setTimeout** (uint16_t timeout)
- uint16_t **getTimeout** (void)
- bool **timeoutOccurred** (void)
- uint8_t **getStatus** (void)

Public Attributes

- vector< int16_t > **a**
- vector< int16_t > **g**
- vector< float > **dps**
- float **mdps** = 0
- float **mg** = 0
- uint8_t **last_status**

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

- src/LSM6.h
- src/LSM6.cpp

5.6 PIDController Class Reference

Public Member Functions

- **PIDController** (float p, float i=0, float d=0, float bound=0)
- float calcEffort (float error)

  Used to calculate the effort from the error.
- float **setKp** (float k)
- float **setKi** (float k)
- float **setKd** (float k)
- float **setCap** (float cap)
- void **resetSum** (void)

Protected Attributes

- float **Kp**
- float **Ki**
- float **Kd**
- float **currError** = 0
- float **prevError** = 0
- float **sumError** = 0
- float **errorBound** = 0
- float **deltaT** = 0
- float **currEffort** = 0

5.6.1 Member Function Documentation

5.6.1.1 calcEffort()

```
float PIDController::calcEffort (
            float error )
```

Used to calculate the effort from the error.

Parameters

error	The current error (calculated in the calling code).

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

- src/PIDcontroller.h
- src/PIDcontroller.cpp

5.7 FastGPIO::Pin< pin > Class Template Reference

```
#include <FastGPIO.h>
```

Static Public Member Functions

- static void setOutputLow () __attribute__((always_inline))

  Configures the pin to be an output driving low.
- static void setOutputHigh () __attribute__((always_inline))

  Configures the pin to be an output driving high.
- static void **setOutputToggle** () __attribute__((always_inline))

  Configures the pin to be an output and toggles it.
- static void setOutput (bool value) __attribute__((always_inline))

  Sets the pin as an output.

- static void setOutputValueLow () __attribute__((always_inline))

  Sets the output value of the pin to 0.
- static void setOutputValueHigh () __attribute__((always_inline))

  Sets the output value of the pin to 1.
- static void setOutputValueToggle () __attribute__((always_inline))

  Toggles the output value of the pin.
- static void setOutputValue (bool value) __attribute__((always_inline))

  Sets the output value of the pin.
- static void **setInput** () __attribute__((always_inline))

  Sets a pin to be a digital input with the internal pull-up resistor disabled.
- static void **setInputPulledUp** () __attribute__((always_inline))

  Sets a pin to be a digital input with the internal pull-up resistor enabled.
- static bool isInputHigh () __attribute__((always_inline))

  Reads the input value of the pin.
- static bool isOutput () __attribute__((always_inline))

  Returns 1 if the pin is configured as an output.
- static bool isOutputValueHigh () __attribute__((always_inline))

  Returns the output value of the pin.
- static uint8_t getState ()

  Returns the full 2-bit state of the pin.
- static void setState (uint8_t state)

  Sets the full 2-bit state of the pin.

5.7.1 Detailed Description

template<uint8_t pin>
class FastGPIO::Pin< pin >

Template Parameters

pin	The pin number

The FastGPIO::Pin class provides static functions for manipulating pins. This class can only be used if the pin number is known at compile time, which means it does not come from a variable that might change and it does not come from the result of a complicated calculation.

Here is some example code showing how to use this class to blink an LED:

```
#include <FastGPIO.h>
#define LED_PIN 13
void setup() {
}
void loop() {
  FastGPIO::Pin<LED_PIN>::setOutput(0);
  delay(500);
  FastGPIO::Pin<LED_PIN>::setOutput(1);
  delay(500);
}
```

5.7.2 Member Function Documentation

5.7.2.1 getState()

```
template<uint8_t pin>
static uint8_t FastGPIO::Pin< pin >::getState ( )  [inline], [static]
```

Returns the full 2-bit state of the pin.

Bit 0 of this function's return value is the pin's output value. Bit 1 of the return value is the pin direction; a value of 1 means output. All the other bits are zero.

5.7.2.2 isInputHigh()

```
template<uint8_t pin>
static bool FastGPIO::Pin< pin >::isInputHigh ( )  [inline], [static]
```

Reads the input value of the pin.

Returns

0 if the pin is low, 1 if the pin is high.

5.7.2.3 isOutput()

```
template<uint8_t pin>
static bool FastGPIO::Pin< pin >::isOutput ( )  [inline], [static]
```

Returns 1 if the pin is configured as an output.

Returns

1 if the pin is an output, 0 if it is an input.

5.7.2.4 isOutputValueHigh()

```
template<uint8_t pin>
static bool FastGPIO::Pin< pin >::isOutputValueHigh ( )  [inline], [static]
```

Returns the output value of the pin.

This is mainly intended to be called on pins that have been configured an an output. If it is called on an input pin, the return value indicates whether the pull-up resistor is enabled or not.

5.7.2.5 setOutput()

```
template<uint8_t pin>
static void FastGPIO::Pin< pin >::setOutput (
            bool value )  [inline], [static]
```

Sets the pin as an output.

Parameters

value	Should be 0, LOW, or false to drive the pin low. Should be 1, HIGH, or true to drive the pin high.

The PORT bit is set before the DDR bit to ensure that the output is not accidentally driven to the wrong value during the transition.

5.7.2.6 setOutputHigh()

```
template<uint8_t pin>
static void FastGPIO::Pin< pin >::setOutputHigh ( )  [inline], [static]
```

Configures the pin to be an output driving high.

This is equivalent to calling setOutput with an argument of 1, but it has a simpler implementation which means it is more likely to be compiled down to just 2 assembly instructions.

5.7.2.7 setOutputLow()

```
template<uint8_t pin>
static void FastGPIO::Pin< pin >::setOutputLow ( )  [inline], [static]
```

Configures the pin to be an output driving low.

This is equivalent to calling setOutput with an argument of 0, but it has a simpler implementation which means it is more likely to be compiled down to just 2 assembly instructions.

5.7.2.8 setOutputValue()

```
template<uint8_t pin>
static void FastGPIO::Pin< pin >::setOutputValue (
            bool value )  [inline], [static]
```

Sets the output value of the pin.

Parameters

value	Should be 0, LOW, or false to drive the pin low. Should be 1, HIGH, or true to drive the pin high.

This is mainly intended to be used on pins that have already been configured as an output.

If this function is used on an input pin, it has the effect of toggling setting the state of the input pin's pull-up resistor.

5.7.2.9 setOutputValueHigh()

```
template<uint8_t pin>
static void FastGPIO::Pin< pin >::setOutputValueHigh ( )  [inline], [static]
```

Sets the output value of the pin to 1.

This is mainly intended to be used on pins that have already been configured as an output in order to make the output drive low.

If this is used on an input pin, it has the effect of enabling the input pin's pull-up resistor.

5.7.2.10 setOutputValueLow()

```
template<uint8_t pin>
static void FastGPIO::Pin< pin >::setOutputValueLow ( )  [inline], [static]
```

Sets the output value of the pin to 0.

This is mainly intended to be used on pins that have already been configured as an output in order to make the output drive low.

If this is used on an input pin, it has the effect of disabling the input pin's pull-up resistor.

5.7.2.11 setOutputValueToggle()

```
template<uint8_t pin>
static void FastGPIO::Pin< pin >::setOutputValueToggle ( )  [inline], [static]
```

Toggles the output value of the pin.

This is mainly intended to be used on pins that have already been configured as an output. If the pin was driving low, this function changes it to drive high. If the pin was driving high, this function changes it to drive low.

If this function is used on an input pin, it has the effect of toggling the state of the input pin's pull-up resistor.

5.7.2.12 setState()

```
template<uint8_t pin>
static void FastGPIO::Pin< pin >::setState (
            uint8_t state )  [inline], [static]
```

Sets the full 2-bit state of the pin.

Parameters

state	The state of the pin, as returns from getState. All bits other than bits 0 and 1 are ignored.

Sometimes this function will need to change both the PORT bit (which specifies the output value) and the DDR bit (which specifies whether the pin is an output). If the DDR bit is getting set to 0, this function changes DDR first, and if it is getting set to 1, this function changes DDR last. This guarantees that the intermediate pin state is always an input state.

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

- src/FastGPIO.h

5.8 FastGPIO::PinLoan< pin > Class Template Reference

`#include <FastGPIO.h>`

Public Attributes

- uint8_t **state**

  The state of the pin as returned from FastGPIO::Pin::getState.

5.8.1 Detailed Description

template<uint8_t pin>
class FastGPIO::PinLoan< pin >

This class saves the state of the specified pin in its constructor when it is created, and restores the pin to that state in its destructor. This can be very useful if a pin is being used for multiple purposes. It allows you to write code that temporarily changes the state of the pin and is guaranteed to restore the state later.

For example, if you were controlling both a button and an LED using a single pin and you wanted to see if the button was pressed without affecting the LED, you could write:

```
bool buttonIsPressed()
{
    FastGPIO::PinLoan<IO_D5> loan;
    FastGPIO::Pin<IO_D5>::setInputPulledUp();
    _delay_us(10);
    return !FastGPIO::Pin<IO_D5>::isInputHigh();
}
```

This is equivalent to:

```
bool buttonIsPressed()
{
    uint8_t state = FastGPIO::Pin<IO_D5>::getState();
    FastGPIO::Pin<IO_D5>::setInputPulledUp();
    _delay_us(10);
    bool value = !FastGPIO::Pin<IO_D5>::isInputHigh();
    FastGPIO::Pin<IO_D5>::setState(state);
    return value;
}
```

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

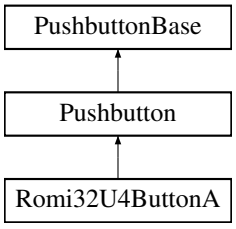
- src/FastGPIO.h

5.9 Pushbutton Class Reference

Main class for interfacing with pushbuttons.

`#include <Pushbutton.h>`

Inheritance diagram for Pushbutton:



Public Member Functions

- Pushbutton (uint8_t pin, uint8_t pullUp=PULL_UP_ENABLED, uint8_t defaultState=DEFAULT_STATE_HIGH)
- virtual bool isPressed ()

  indicates whether button is currently pressed without any debouncing.

5.9.1 Detailed Description

Main class for interfacing with pushbuttons.

This class can interface with any pushbutton whose state can be read with the `digitalRead` function, which is part of the Arduino core.

See `https://github.com/pololu/pushbutton-arduino` for an overview of the different ways to use this class.

5.9.2 Constructor & Destructor Documentation

5.9.2.1 Pushbutton()

```
Pushbutton::Pushbutton (
            uint8_t pin,
            uint8_t pullUp = PULL_UP_ENABLED,
            uint8_t defaultState = DEFAULT_STATE_HIGH )
```

Constructs a new instance of Pushbutton.

Parameters

pin	The pin number of the pin. This is used as an argument to `pinMode` and `digitalRead`.
pullUp	Specifies whether the pin's internal pull-up resistor should be enabled. This should be either PULL_UP_ENABLED (which is the default if the argument is omitted) or PULL_UP_DISABLED.
defaultState	Specifies the voltage level that corresponds to the button's default (released) state. This should be either DEFAULT_STATE_HIGH (which is the default if this argument is omitted) or DEFAULT_STATE_LOW.

5.9.3 Member Function Documentation

5.9.3.1 isPressed()

```
bool Pushbutton::isPressed ( )  [virtual]
```

indicates whether button is currently pressed without any debouncing.

Returns

1 if the button is pressed right now, 0 if it is not.

This function must be implemented in a subclass of PushbuttonBase, such as Pushbutton.

Implements PushbuttonBase.

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

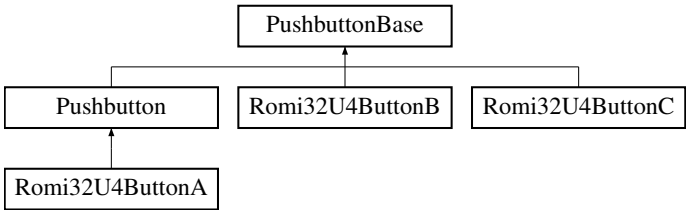
- src/Pushbutton.h
- src/Pushbutton.cpp

5.10 PushbuttonBase Class Reference

General pushbutton class that handles debouncing.

`#include <Pushbutton.h>`

Inheritance diagram for PushbuttonBase:



Public Member Functions

- void waitForPress ()

  Waits until the button is pressed and takes care of debouncing.
- void waitForRelease ()

  Waits until the button is released and takes care of debouncing.
- void waitForButton ()

  Waits until the button is pressed and then waits until the button is released, taking care of debouncing.
- bool getSingleDebouncedPress ()

  Uses a state machine to return true once after each time it detects the button moving from the released state to the pressed state.
- bool getSingleDebouncedRelease ()

  Uses a state machine to return true once after each time it detects the button moving from the pressed state to the released state.
- virtual bool isPressed ()=0

  indicates whether button is currently pressed without any debouncing.

5.10.1 Detailed Description

General pushbutton class that handles debouncing.

/*!

This is an abstract class used for interfacing with pushbuttons. It knows about debouncing, but it knows nothing about how to read the current state of the button. The functions in this class get the current state of the button by calling isPressed(), a virtual function which must be implemented in a subclass of PushbuttonBase, such as Pushbutton.

Most users of this library do not need to directly use PushbuttonBase or even know that it exists. They can use Pushbutton instead.

5.10.2 Member Function Documentation

5.10.2.1 getSingleDebouncedPress()

```
bool PushbuttonBase::getSingleDebouncedPress ( )
```

Uses a state machine to return true once after each time it detects the button moving from the released state to the pressed state.

This is a non-blocking function that is meant to be called repeatedly in a loop. Each time it is called, it updates a state machine that monitors the state of the button. When it detects the button changing from the released state to the pressed state, with debouncing, it returns true.

5.10.2.2 getSingleDebouncedRelease()

```
bool PushbuttonBase::getSingleDebouncedRelease ( )
```

Uses a state machine to return true once after each time it detects the button moving from the pressed state to the released state.

This is just like getSingleDebouncedPress() except it has a separate state machine and it watches for when the button goes from the pressed state to the released state.

There is no strict guarantee that every debounced button press event returned by getSingleDebouncedPress() will have a corresponding button release event returned by getSingleDebouncedRelease(); the two functions use independent state machines and sample the button at different times.

5.10.2.3 isPressed()

```
virtual bool PushbuttonBase::isPressed ( )  [pure virtual]
```

indicates whether button is currently pressed without any debouncing.

Returns

1 if the button is pressed right now, 0 if it is not.

This function must be implemented in a subclass of PushbuttonBase, such as Pushbutton.

Implemented in Pushbutton, Romi32U4ButtonB, and Romi32U4ButtonC.

5.10.2.4 waitForButton()

```
void PushbuttonBase::waitForButton ( )
```

Waits until the button is pressed and then waits until the button is released, taking care of debouncing.

This is equivalent to calling waitForPress() and then waitForRelease().

5.10.2.5 waitForPress()

```
void PushbuttonBase::waitForPress ( )
```

Waits until the button is pressed and takes care of debouncing.

This function waits until the button is in the pressed state and then returns. Note that if the button is already pressed when you call this function, it will return quickly (in 10 ms).

5.10.2.6 waitForRelease()

```
void PushbuttonBase::waitForRelease ( )
```

Waits until the button is released and takes care of debouncing.

This function waits until the button is in the released state and then returns. Note that if the button is already released when you call this function, it will return quickly (in 10 ms).

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

- src/Pushbutton.h
- src/Pushbutton.cpp

5.11 Rangefinder Class Reference

```
#include <Rangefinder.h>
```

Public Member Functions

- **Rangefinder** (uint8_t echo, uint8_t trig)
- void **init** (void)
- uint8_t checkPingTimer (void)
- uint16_t **checkEcho** (void)
- float **getDistance** (void)

  Returns the last recorded distance.
- void **ISR_echo** (void)

Protected Attributes

- volatile uint8_t **state** = 0
- uint8_t **echoPin** = -1
- uint8_t **trigPin** = -1
- uint32_t **lastPing** = 0
- uint32_t **pingInterval** = 10
- volatile uint32_t **pulseStart** = 0
- volatile uint32_t **pulseEnd** = 0
- float **distance** = 99

5.11.1 Detailed Description

A class to manage an ultrasonic rangefinder.

Uses a TRIG and ECHO pin to send chirps and detect round trip time.

5.11.2 Member Function Documentation

5.11.2.1 checkPingTimer()

```
uint8_t Rangefinder::checkPingTimer (
            void  )
```

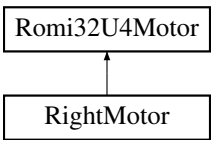
checkPingTimer check to see if it's time to send a new ping.

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

- src/Rangefinder.h
- src/Rangefinder.cpp

5.12 RightMotor Class Reference

Inheritance diagram for RightMotor:



Public Member Functions

- void **setMotorEffort** (int16_t effort)

Protected Member Functions

- void setEffort (int16_t effort)

  Sets the effort for the motor directly. Overloaded for the left and right motors. Use Chassis::setEfforts() to control motors.

Additional Inherited Members

5.12.1 Member Function Documentation

5.12.1.1 setEffort()

```
void RightMotor::setEffort (
            int16_t effort )  [protected], [virtual]
```

Sets the effort for the motor directly. Overloaded for the left and right motors. Use Chassis::setEfforts() to control motors.

Parameters

effort	A number from -300 to 300 representing the effort and direction of the left motor. Values of -300 or less result in full effort reverse, and values of 300 or more result in full effort forward.

Implements Romi32U4Motor.

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

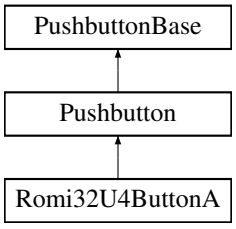
- src/Romi32U4Motors.h
- src/Romi32U4Motors.cpp

5.13 Romi32U4ButtonA Class Reference

Interfaces with button A on the Romi 32U4.

```
#include <Romi32U4Buttons.h>
```

Inheritance diagram for Romi32U4ButtonA:



Additional Inherited Members

5.13.1 Detailed Description

Interfaces with button A on the Romi 32U4.

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

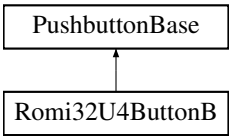
- src/Romi32U4Buttons.h

5.14 Romi32U4ButtonB Class Reference

Interfaces with button B on the Romi 32U4.

`#include <Romi32U4Buttons.h>`

Inheritance diagram for Romi32U4ButtonB:



Public Member Functions

- virtual bool isPressed ()

  indicates whether button is currently pressed without any debouncing.

5.14.1 Detailed Description

Interfaces with button B on the Romi 32U4.

The pin used for button B is also used for the TX LED.

This class temporarily disables USB interrupts because the Arduino core code has USB interrupts enabled that sometimes write to the pin this button is on.

This class temporarily sets the pin to be an input without a pull-up resistor. The pull-up resistor is not needed because of the resistors on the board.

5.14.2 Member Function Documentation

5.14.2.1 isPressed()

```
virtual bool Romi32U4ButtonB::isPressed ( )  [inline], [virtual]
```

indicates whether button is currently pressed without any debouncing.

Returns

1 if the button is pressed right now, 0 if it is not.

This function must be implemented in a subclass of PushbuttonBase, such as Pushbutton.

Implements PushbuttonBase.

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

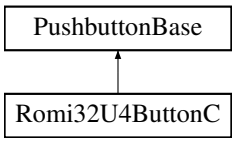
- src/Romi32U4Buttons.h

5.15 Romi32U4ButtonC Class Reference

Interfaces with button C on the Romi 32U4.

```
#include <Romi32U4Buttons.h>
```

Inheritance diagram for Romi32U4ButtonC:



Public Member Functions

- virtual bool isPressed ()
  indicates whether button is currently pressed without any debouncing.

5.15.1 Detailed Description

Interfaces with button C on the Romi 32U4.

The pin used for button C is also used for the RX LED.

This class temporarily disables USB interrupts because the Arduino core code has USB interrupts enabled that sometimes write to the pin this button is on.

This class temporarily sets the pin to be an input without a pull-up resistor. The pull-up resistor is not needed because of the resistors on the board.

5.15.2 Member Function Documentation

5.15.2.1 isPressed()

```
virtual bool Romi32U4ButtonC::isPressed ( )  [inline], [virtual]
```

indicates whether button is currently pressed without any debouncing.

Returns

1 if the button is pressed right now, 0 if it is not.

This function must be implemented in a subclass of PushbuttonBase, such as Pushbutton.

Implements PushbuttonBase.

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

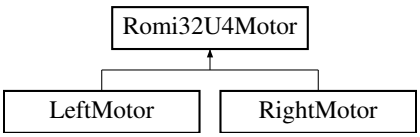
- src/Romi32U4Buttons.h

5.16 Romi32U4Motor Class Reference

Controls motor effort and direction on the Romi 32U4.

```
#include <Romi32U4Motors.h>
```

Inheritance diagram for Romi32U4Motor:



Public Member Functions

- void **setPIDCoeffients** (float kp, float ki)
- void allowTurbo (bool turbo)

  Turns turbo mode on or off.
- void handleISR (bool newA, bool newB)

Static Public Member Functions

- static void init ()

Protected Types

- enum **CTRL_MODE** : uint8_t { **CTRL_DIRECT** , **CTRL_SPEED** , **CTRL_POS** }

Protected Member Functions

- virtual void setEffort (int16_t effort)=0

  Sets the effort for the motor directly. Overloaded for the left and right motors. Use Chassis::setEfforts() to control motors.
- int16_t getCount (void)
- int16_t getAndResetCount (void)
- void setTargetSpeed (int16_t targetSpeed)
- void moveFor (int16_t amount)
- bool **checkComplete** (void)
- void update (void)
- void calcEncoderDelta (void)

Static Protected Member Functions

- static void initMotors ()
- static void initEncoders ()

Protected Attributes

- volatile CTRL_MODE **ctrlMode** = CTRL_DIRECT
- volatile int16_t **speed** = 0
- int16_t **targetSpeed** = 0
- int16_t **targetPos** = 0
- int16_t **maxEffort** = 300
- volatile int16_t **prevCount** = 0
- volatile int16_t **count** = 0
- volatile int16_t **lastA** = 0
- volatile int16_t **lastB** = 0
- PIDController **pidCtrl**

Friends

- class **Chassis**

5.16.1 Detailed Description

Controls motor effort and direction on the Romi 32U4.

This library uses Timer 1, so it will conflict with any other libraries using that timer.

Also reads counts from the encoders on the Romi 32U4.

This class allows you to read counts from the encoders on the Romi 32U4, which lets you tell how much each motor has turned and in what direction.

The encoders are monitored in the background using interrupts, so your code can perform other tasks without missing encoder counts.

5.16.2 Member Function Documentation

5.16.2.1 allowTurbo()

```
void Romi32U4Motor::allowTurbo (
            bool turbo )
```

Turns turbo mode on or off.

By default turbo mode is off. When turbo mode is on, the range of speeds accepted by the other functions in this library becomes -400 to 400 (instead of -300 to 300). Turning turbo mode on allows the Romi to move faster but could decrease the lifetime of the motors.

This function does not have any immediate effect on the speed of the motors; it just changes the behavior of the other functions in this library.

Parameters

turbo	If true, turns turbo mode on. If false, turns turbo mode off.

Top speed is limited to 300/420 by default. This allow you to go faster. Be careful.

5.16.2.2 calcEncoderDelta()

```
void Romi32U4Motor::calcEncoderDelta (
            void ) [protected]
```

calcEncoderDelta() is called automatically by an ISR. It takes a 'snapshot of the encoders and stores the change since the last call in speed, which has units of "encoder ticks/16 ms interval"

Because it is called from within an ISR, interrupts don't need to be disabled.

5.16.2.3 getAndResetCount()

```
int16_t Romi32U4Motor::getAndResetCount (
            void ) [protected]
```

Resets the encoder count and returns the last count.

5.16.2.4 getCount()

```
int16_t Romi32U4Motor::getCount (
            void ) [protected]
```

Returns the number of counts that have been detected from the left-side encoder. These counts start at 0. Positive counts correspond to forward movement of the left side of the Romi, while negative counts correspond to backwards movement.

The count is returned as a signed 16-bit integer. When the count goes over 32767, it will overflow down to -32768. When the count goes below -32768, it will overflow up to 32767.

Returns the current encoder count.

5.16.2.5 handleISR()

```
void Romi32U4Motor::handleISR (
            bool newA,
            bool newB )  [inline]
```

Service function for the ISR

Calculates the encoder counter increment/decrement due to an encoder transition. Pololu sets up their encoders in an interesting way with some logic chips, so first we have to deconvolute the encoder signals (in the ISR); then, we call this function to update the counter.

More details are found here:

https://www.pololu.com/docs/0J69/3.3

This function is called from the ISR, which does the actual deconvolution for each motor.

5.16.2.6 init()

```
static void Romi32U4Motor::init (
            void  )  [inline], [static]
```

Must be called near the beginning of the program [usually in Chassis::init()]

5.16.2.7 initEncoders()

```
void Romi32U4Motor::initEncoders (
            void  )  [static], [protected]
```

Set up the encoder 'machinery'. Call it near the beginning of the program.

Do not edit this function.

5.16.2.8 initMotors()

```
void Romi32U4Motor::initMotors ( )  [static], [protected]
```

initMotors() should be called near the beginning of the program (usually in Chassis::init()). It sets up Timer4 to run at 38 kHz, which is used to both drive the PWM signal for the motors and (tangentially) allow for a 38 kHz signal on pin 11, which can be used, say, to drive an IR LED at a common rate.

Timer 1 has the following configuration: prescaler of 1 outputs enabled on channels A (pin 9), B (pin 10) and C (pin 11) fast PWM mode top of 420, which will be the max speed frequency is then: 16 MHz / [1 (prescaler) / (420 + 1)] = 38.005 kHz

5.16.2.9 moveFor()

```
void Romi32U4Motor::moveFor (
            int16_t amount )  [protected]
```

Sets the (delta) target position in "encoder ticks" and a speed to drive to get there in "encoder ticks/16 ms interval"

5.16.2.10 setEffort()

```
virtual void Romi32U4Motor::setEffort (
            int16_t effort )  [protected], [pure virtual]
```

Sets the effort for the motor directly. Overloaded for the left and right motors. Use Chassis::setEfforts() to control motors.

Parameters

effort	A number from -300 to 300 representing the effort and direction of the left motor. Values of -300 or less result in full effort reverse, and values of 300 or more result in full effort forward.

Implemented in LeftMotor, and RightMotor.

5.16.2.11 setTargetSpeed()

```
void Romi32U4Motor::setTargetSpeed (
             int16_t target )  [protected]
```

Sets the target speed in "encoder ticks/16 ms interval"

5.16.2.12 update()

```
void Romi32U4Motor::update (
             void  )  [protected]
```

update() must be called regularly to update the control signals sent to the motors.

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

- src/Romi32U4Motors.h
- src/Romi32U4Encoders.cpp
- src/Romi32U4Motors.cpp

5.17 Servo32U4 Class Reference

Public Member Functions

- void **attach** (void)
- void **detach** (void)
- void **writeMicroseconds** (uint16_t microseconds)
- uint16_t **setMinMaxMicroseconds** (uint16_t min, uint16_t max)

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

- src/servo32u4.h
- src/servo32u4.cpp

5.18 Timer Class Reference

Public Member Functions

- Timer (unsigned long interval)
- bool isExpired ()
- void reset ()
- void reset (unsigned long newInterval)

5.18.1 Constructor & Destructor Documentation

5.18.1.1 Timer()

```
Timer::Timer (
            unsigned long interval )
```

Create a timer that will expire every "interval"

5.18.2 Member Function Documentation

5.18.2.1 isExpired()

```
bool Timer::isExpired ( )
```

Check if the timer is expired, that is the current time is past the expired time.

5.18.2.2 reset() [1/2]

```
void Timer::reset ( )
```

Reset the timer to that the expired time is the current time + interval

5.18.2.3 reset() [2/2]

```
void Timer::reset (
            unsigned long newInterval )
```

Change the timer interval to "NewInterval" then reset the timer to that the expired time is the current time + interval

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

- src/Timer.h
- src/Timer.cpp

5.19 USBPause Class Reference

`#include <USBPause.h>`

5.19.1 Detailed Description

This class disables USB interrupts in its constructor when it is created and restores them to their previous state in its destructor when it is destroyed. This class is tailored to the behavior of the Arduino core USB code, so it might have to change if that code changes.

This class assumes that the only USB interrupts enabled are general device interrupts and endpoint 0 interrupts.

It also assumes that the endpoint 0 interrupts will not enable or disable any of the general device interrupts.

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

- src/USBPause.h

5.20 LSM6::vector< T > Struct Template Reference

Public Attributes

- T **x**
- T **y**
- T **z**

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

- src/LSM6.h

Chapter 6

File Documentation

6.1 Chassis.h

```
1 #pragma once
2
3 #include <Arduino.h>
4 #include <Romi32U4Motors.h>
5
18 class Chassis
19 {
20 public:
21     LeftMotor leftMotor;
22     RightMotor rightMotor;
23
24 protected:
25     const float cmPerEncoderTick;
26     const float robotRadius;
27     const uint16_t ctrlIntervalMS = 16;
28
29 public:
36     Chassis(float wheelDiam, float ticksPerRevolution, float wheelTrack)
37         : cmPerEncoderTick(wheelDiam * M_PI / ticksPerRevolution), robotRadius(wheelTrack / 2.0)
38         {}
39
42     void init(void);
43
46     void setMotorPIDcoeffs(float kp, float ki);
47
50     void idle(void);
51
57     void setMotorEfforts(int leftEffort, int rightEffort);
58
64     void setWheelSpeeds(float leftSpeed, float rightSpeed);
65
71     void setTwist(float forwardSpeed, float turningSpeed);
72
80     void driveFor(float forwardDistance, float forwardSpeed);
81
87     void turnFor(float turnAngle, float turningSpeed);
88
93     bool checkMotionComplete(void);
94
95     void printSpeeds(void);
96
97     inline void updateEncoderDeltas();
98 };
99
100 extern Chassis chassis;
```

6.2 src/FastGPIO.h File Reference

```
#include <avr/io.h>
#include <stdint.h>
```

Classes

- class FastGPIO::Pin< pin >
- class FastGPIO::PinLoan< pin >

6.2.1 Detailed Description

FastGPIO is a C++ header library for efficient AVR I/O.

For an overview of the features of this library, see https://github.com/pololu/fastgpio-arduino. That is the main repository for this library, though copies may exist in other repositories.

The FastGPIO::Pin class provides static functions for manipulating pins. See its class reference for more information.

6.3 FastGPIO.h

Go to the documentation of this file.

```
1 // Copyright Pololu Corporation.  For more information, see http://www.pololu.com/
2
44 #pragma once
45 #include <avr/io.h>
46 #include <stdint.h>
47
49 #define _FG_SBI(mem_addr, bit) asm volatile("sbi %0, %1\n" : \
50     : "I" (mem_addr - __SFR_OFFSET), "I" (bit))
51 #define _FG_CBI(mem_addr, bit) asm volatile("cbi %0, %1\n" : \
52     : "I" (mem_addr - __SFR_OFFSET), "I" (bit))
53 #define _FG_PIN(port, bit) { _SFR_MEM_ADDR(PIN##port), _SFR_MEM_ADDR(PORT##port), \
54      _SFR_MEM_ADDR(DDR##port), bit }
57 namespace FastGPIO
58 {
65     typedef struct IOStruct
66     {
67         uint8_t pinAddr;
68         uint8_t portAddr;
69         uint8_t ddrAddr;
70         uint8_t bit;
71
72         volatile uint8_t * pin() const
73         {
74             return (volatile uint8_t *)(uint16_t)pinAddr;
75         }
76
77         volatile uint8_t * port() const
78         {
79             return (volatile uint8_t *)(uint16_t)portAddr;
80         }
81
82         volatile uint8_t * ddr() const
83         {
84             return (volatile uint8_t *)(uint16_t)ddrAddr;
85         }
86     } IOStruct;
89 #if defined(__AVR_ATmega168__) || defined(__AVR_ATmega168P__) || defined(__AVR_ATmega328__) ||
      defined(__AVR_ATmega328P__)
90
91     const IOStruct pinStructs[] = {
92         _FG_PIN(D, 0),
93         _FG_PIN(D, 1),
94         _FG_PIN(D, 2),
95         _FG_PIN(D, 3),
96         _FG_PIN(D, 4),
97         _FG_PIN(D, 5),
98         _FG_PIN(D, 6),
99         _FG_PIN(D, 7),
100         _FG_PIN(B, 0),
101         _FG_PIN(B, 1),
102         _FG_PIN(B, 2),
103         _FG_PIN(B, 3),
104         _FG_PIN(B, 4),
105         _FG_PIN(B, 5),
```

```
106         _FG_PIN(C, 0),
107         _FG_PIN(C, 1),
108         _FG_PIN(C, 2),
109         _FG_PIN(C, 3),
110         _FG_PIN(C, 4),
111         _FG_PIN(C, 5),
112         _FG_PIN(C, 6),
113         _FG_PIN(C, 7), // Null pin (IO_NONE)
114     };
115
116 #define IO_D0 0
117 #define IO_D1 1
118 #define IO_D2 2
119 #define IO_D3 3
120 #define IO_D4 4
121 #define IO_D5 5
122 #define IO_D6 6
123 #define IO_D7 7
124 #define IO_B0 8
125 #define IO_B1 9
126 #define IO_B2 10
127 #define IO_B3 11
128 #define IO_B4 12
129 #define IO_B5 13
130 #define IO_C0 14
131 #define IO_C1 15
132 #define IO_C2 16
133 #define IO_C3 17
134 #define IO_C4 18
135 #define IO_C5 19
136 #define IO_C6 20
137 #define IO_NONE 21
138
139 #elif defined(__AVR_ATmega32U4__)
140
141     const IOStruct pinStructs[] = {
142         _FG_PIN(D, 2),
143         _FG_PIN(D, 3),
144         _FG_PIN(D, 1),
145         _FG_PIN(D, 0),
146         _FG_PIN(D, 4),
147         _FG_PIN(C, 6),
148         _FG_PIN(D, 7),
149         _FG_PIN(E, 6),
150
151         _FG_PIN(B, 4),
152         _FG_PIN(B, 5),
153         _FG_PIN(B, 6),
154         _FG_PIN(B, 7),
155         _FG_PIN(D, 6),
156         _FG_PIN(C, 7),
157
158         _FG_PIN(B, 3),
159         _FG_PIN(B, 1),
160         _FG_PIN(B, 2),
161         _FG_PIN(B, 0),
162
163         _FG_PIN(F, 7),
164         _FG_PIN(F, 6),
165         _FG_PIN(F, 5),
166         _FG_PIN(F, 4),
167         _FG_PIN(F, 1),
168         _FG_PIN(F, 0),
169
170         _FG_PIN(D, 4),
171         _FG_PIN(D, 7),
172         _FG_PIN(B, 4),
173         _FG_PIN(B, 5),
174         _FG_PIN(B, 6),
175         _FG_PIN(D, 6),
176
177         // Extra pins added by this library and not supported by the
178         // Arduino GPIO functions:
179         _FG_PIN(D, 5),
180         _FG_PIN(E, 2),
181
182         _FG_PIN(E, 0)  // Null pin (IO_NONE)
183     };
184
185 #define IO_D2 0
186 #define IO_D3 1
187 #define IO_D1 2
188 #define IO_D0 3
189 #define IO_D4 4
190 #define IO_C6 5
191 #define IO_D7 6
192 #define IO_E6 7
```

```
193 #define IO_B4 8
194 #define IO_B5 9
195 #define IO_B6 10
196 #define IO_B7 11
197 #define IO_D6 12
198 #define IO_C7 13
199 #define IO_B3 14
200 #define IO_B1 15
201 #define IO_B2 16
202 #define IO_B0 17
203 #define IO_F7 18
204 #define IO_F6 19
205 #define IO_F5 20
206 #define IO_F4 21
207 #define IO_F1 22
208 #define IO_F0 23
209 #define IO_D5 30
210 #define IO_E2 31
211 #define IO_NONE 32
212
213 #else
214 #error FastGPIO does not support this board.
215 #endif
216
217     template<uint8_t pin> class Pin
218     {
219     public:
226         static inline void setOutputLow() __attribute__((always_inline))
227         {
228             _FG_CBI(pinStructs[pin].portAddr, pinStructs[pin].bit);
229             _FG_SBI(pinStructs[pin].ddrAddr, pinStructs[pin].bit);
230         }
231
238         static inline void setOutputHigh() __attribute__((always_inline))
239         {
240             _FG_SBI(pinStructs[pin].portAddr, pinStructs[pin].bit);
241             _FG_SBI(pinStructs[pin].ddrAddr, pinStructs[pin].bit);
242         }
243
246         static inline void setOutputToggle() __attribute__((always_inline))
247         {
248             setOutputValueToggle();
249             _FG_SBI(pinStructs[pin].ddrAddr, pinStructs[pin].bit);
250         }
251
260         static inline void setOutput(bool value) __attribute__((always_inline))
261         {
262             setOutputValue(value);
263             _FG_SBI(pinStructs[pin].ddrAddr, pinStructs[pin].bit);
264         }
265
274         static inline void setOutputValueLow() __attribute__((always_inline))
275         {
276             _FG_CBI(pinStructs[pin].portAddr, pinStructs[pin].bit);
277         }
278
287         static inline void setOutputValueHigh() __attribute__((always_inline))
288         {
289             _FG_SBI(pinStructs[pin].portAddr, pinStructs[pin].bit);
290         }
291
302         static inline void setOutputValueToggle() __attribute__((always_inline))
303         {
304             _FG_SBI(pinStructs[pin].pinAddr, pinStructs[pin].bit);
305         }
306
318         static inline void setOutputValue(bool value) __attribute__((always_inline))
319         {
320             if (value)
321             {
322                 _FG_SBI(pinStructs[pin].portAddr, pinStructs[pin].bit);
323             }
324             else
325             {
326                 _FG_CBI(pinStructs[pin].portAddr, pinStructs[pin].bit);
327             }
328         }
329
333         static inline void setInput() __attribute__((always_inline))
334         {
335             _FG_CBI(pinStructs[pin].ddrAddr, pinStructs[pin].bit);
336             _FG_CBI(pinStructs[pin].portAddr, pinStructs[pin].bit);
337         }
338
342         static inline void setInputPulledUp() __attribute__((always_inline))
343         {
344             _FG_CBI(pinStructs[pin].ddrAddr, pinStructs[pin].bit);
```

```
345             _FG_SBI(pinStructs[pin].portAddr, pinStructs[pin].bit);
346         }
347 
352         static inline bool isInputHigh() __attribute__((always_inline))
353         {
354             return *pinStructs[pin].pin() » pinStructs[pin].bit & 1;
355 
356             /* This is another option but it is less efficient in code
357                like "if (isInputHigh()) { ... }":
358             bool value;
359             asm volatile(
360                 "ldi %0, 0\n"
361                 "sbic %2, %1\n"
362                 "ldi %0, 1\n"
363                 : "=d" (value)
364                 : "I" (pinStructs[pin].bit),
365                   "I" (pinStructs[pin].pinAddr - __SFR_OFFSET));
366             return value;
367             */
368         }
369 
374         static inline bool isOutput() __attribute__((always_inline))
375         {
376             return *pinStructs[pin].ddr() » pinStructs[pin].bit & 1;
377         }
378 
385         static inline bool isOutputValueHigh() __attribute__((always_inline))
386         {
387             return *pinStructs[pin].port() » pinStructs[pin].bit & 1;
388         }
389 
396         static uint8_t getState()
397         {
398             uint8_t state;
399             asm volatile(
400                 "ldi  %0, 0\n"
401                 "sbic %2, %1\n"
402                 "ori  %0, 1\n"   // Set state bit 0 to 1 if PORT bit is set.
403                 "sbic %3, %1\n"
404                 "ori  %0, 2\n"   // Set state bit 1 to 1 if DDR bit is set.
405                 : "=a" (state)
406                 : "I" (pinStructs[pin].bit),
407                   "I" (pinStructs[pin].portAddr - __SFR_OFFSET),
408                   "I" (pinStructs[pin].ddrAddr - __SFR_OFFSET));
409             return state;
410 
411             /* Equivalent C++ code:
412               return isOutput() « 1 | isOutputValueHigh();
413             */
414         }
415 
428         static void setState(uint8_t state)
429         {
430             asm volatile(
431                 "bst  %0, 1\n"   // Set DDR to 0 if needed
432                 "brts .+2\n"
433                 "cbi  %3, %1\n"
434                 "bst  %0, 0\n"   // Copy state bit 0 to PORT bit
435                 "brts .+2\n"
436                 "cbi  %2, %1\n"
437                 "brtc .+2\n"
438                 "sbi  %2, %1\n"
439                 "bst  %0, 1\n"   // Set DDR to 1 if needed
440                 "brtc .+2\n"
441                 "sbi  %3, %1\n"
442                 :
443                 : "a" (state),
444                   "I" (pinStructs[pin].bit),
445                   "I" (pinStructs[pin].portAddr - __SFR_OFFSET),
446                   "I" (pinStructs[pin].ddrAddr - __SFR_OFFSET));
447         }
448     };
449 
484     template<uint8_t pin> class PinLoan
485     {
486     public:
488         uint8_t state;
489 
490         PinLoan()
491         {
492             state = Pin<pin>::getState();
493         }
494 
495         ~PinLoan()
496         {
497             Pin<pin>::setState(state);
498         }
```

```
499     };
500 };
501
502 #undef _FG_PIN
503 #undef _FG_CBI
504 #undef _FG_SBI
```

6.4 ir_codes.h

```
1 #pragma once
2
3 #define VOLminus    0
4 #define PLAY_PAUSE  1
5 #define VOLplus     2
6
7 #define SETUP_BTN   4
8 #define UP_ARROW    5
9 #define STOP_MODE   6
10
11 #define LEFT_ARROW  8
12 #define ENTER_SAVE  9
13 #define RIGHT_ARROW 10
14
15 #define NUM_0_10    12
16 #define DOWN_ARROW  13
17 #define REWIND      14
18
19 #define NUM_1       16
20 #define NUM_2       17
21 #define NUM_3       18
22
23 #define NUM_4       20
24 #define NUM_5       21
25 #define NUM_6       22
26
27 #define NUM_7       24
28 #define NUM_8       25
29 #define NUM_9       26
```

6.5 IRdecoder.h

```
1 #include <Arduino.h>
2
3 /*
4  * A class to interpret IR remotes with NEC encoding. NEC encoding sends four bytes:
5  *
6  * [device ID, ~divice ID, key code, ~key code]
7  *
8  * Sending the inverse allow for easy error checking (and reduces saturation in the receiver).
9  *
10  * Codes are send in little endian; this library reverses upon reception, so the first bit received
11  * is in the LSB of currCode. That means that the key code is found in bits [23..16] of currCode
12  *
13  * https://techdocs.altium.com/display/FPGA/NEC+Infrared+Transmission+Protocol
14  *
15  * This does not interpret the codes into which key was pressed. That needs to be
16  * mapped on a remote by remote basis.
17  */
18
19 class IRDecoder
20 {
21 private:
22   uint8_t pin = -1;
23
24   enum IR_STATE
25   {
26     IR_READY,    //idle, returns to this state after you request a code
27     IR_PREAMBLE, //received the start burst, waiting for first bit
28     IR_REPEAT,   //received repeat code (part of NEC protocol); last code will be returned
29     IR_ACTIVE,   //have some bits, but not yet complete
30     IR_COMPLETE, //a valid code has been received
31     IR_ERROR     //an error occurred; won't return a valid code
32   };
33
34   IR_STATE state = IR_READY; //a simple state machine for managing reception
35
36   volatile uint32_t lastReceiveTime = 0; //not really used -- could be used to sunset codes
37
38   volatile uint32_t currCode = 0; //the most recently received valid code
```

```
39   volatile uint8_t index = 0;     //for tracking which bit we're on
40
41   volatile uint32_t fallingEdge = 0;
42   volatile uint32_t risingEdge = 0;
43
44   volatile uint32_t lastRisingEdge = 0; //used for tracking spacing between rising edges, i.e., bit value
45 public:
46   //volatile uint16_t bits[32];  //I used this for debugging; obsolete
47
48 public:
49   IRDecoder(uint8_t p) : pin(p) {}
50   void init(void);           //call this in the setup()
51   void handleIRsensor(void); //ISR
52
53   uint32_t getCode(void) //returns the most recent valid code; returns zero if there was an error
54   {
55     if (state == IR_COMPLETE || state == IR_REPEAT)
56     {
57       state = IR_READY;
58       return currCode;
59     }
60     else
61       return 0;
62   }
63
64   int16_t getKeyCode(bool acceptRepeat = false) //returns the most recent key code; returns -1 on error
       (not sure if 0 can be a code or not!!!)
65   {
66     if (state == IR_COMPLETE || (acceptRepeat == true && state == IR_REPEAT))
67     {
68       state = IR_READY;
69       return (currCode » 16) & 0x0ff;
70     }
71     else
72       return -1;
73   }
74 };
75
76 extern IRDecoder decoder;
```

6.6 LSM6.h

```
1 #ifndef LSM6_h
2 #define LSM6_h
3
4 #include <Arduino.h>
5
6 class LSM6
7 {
8   public:
9     template <typename T> struct vector
10    {
11      T x, y, z;
12    };
13
14    enum deviceType { device_DS33, device_auto };
15    enum sa0State { sa0_low, sa0_high, sa0_auto };
16
17    enum ACC_FS {ACC_FS2, ACC_FS4, ACC_FS8, ACC_FS16};
18    enum GYRO_FS {GYRO_FS245, GYRO_FS500, GYRO_FS1000, GYRO_FS2000};
19    enum ODR
20    {
21      ODR13 = 0x1,
22      ODR26 = 0x2,
23      ODR52 = 0x3,
24      ODR104 = 0x4,
25      ODR208 = 0x5,
26      ODR416 = 0x6,
27      ODR833 = 0x7,
28      ODR166k = 0x8
29    };
30
31    // register addresses
32    enum regAddr
33    {
34      FUNC_CFG_ACCESS   = 0x01,
35
36      FIFO_CTRL1        = 0x06,
37      FIFO_CTRL2        = 0x07,
38      FIFO_CTRL3        = 0x08,
39      FIFO_CTRL4        = 0x09,
40      FIFO_CTRL5        = 0x0A,
41      ORIENT_CFG_G      = 0x0B,
```

```
42
43        INT1_CTRL          = 0x0D,
44        INT2_CTRL          = 0x0E,
45        WHO_AM_I           = 0x0F,
46        CTRL1_XL           = 0x10,
47        CTRL2_G            = 0x11,
48        CTRL3_C            = 0x12,
49        CTRL4_C            = 0x13,
50        CTRL5_C            = 0x14,
51        CTRL6_C            = 0x15,
52        CTRL7_G            = 0x16,
53        CTRL8_XL           = 0x17,
54        CTRL9_XL           = 0x18,
55        CTRL10_C           = 0x19,
56
57        WAKE_UP_SRC        = 0x1B,
58        TAP_SRC            = 0x1C,
59        D6D_SRC            = 0x1D,
60        STATUS_REG         = 0x1E,
61
62        OUT_TEMP_L         = 0x20,
63        OUT_TEMP_H         = 0x21,
64        OUTX_L_G           = 0x22,
65        OUTX_H_G           = 0x23,
66        OUTY_L_G           = 0x24,
67        OUTY_H_G           = 0x25,
68        OUTZ_L_G           = 0x26,
69        OUTZ_H_G           = 0x27,
70        OUTX_L_XL          = 0x28,
71        OUTX_H_XL          = 0x29,
72        OUTY_L_XL          = 0x2A,
73        OUTY_H_XL          = 0x2B,
74        OUTZ_L_XL          = 0x2C,
75        OUTZ_H_XL          = 0x2D,
76
77        FIFO_STATUS1       = 0x3A,
78        FIFO_STATUS2       = 0x3B,
79        FIFO_STATUS3       = 0x3C,
80        FIFO_STATUS4       = 0x3D,
81        FIFO_DATA_OUT_L    = 0x3E,
82        FIFO_DATA_OUT_H    = 0x3F,
83        TIMESTAMP0_REG     = 0x40,
84        TIMESTAMP1_REG     = 0x41,
85        TIMESTAMP2_REG     = 0x42,
86
87        STEP_TIMESTAMP_L   = 0x49,
88        STEP_TIMESTAMP_H   = 0x4A,
89        STEP_COUNTER_L     = 0x4B,
90        STEP_COUNTER_H     = 0x4C,
91
92        FUNC_SRC           = 0x53,
93
94        TAP_CFG            = 0x58,
95        TAP_THS_6D         = 0x59,
96        INT_DUR2           = 0x5A,
97        WAKE_UP_THS        = 0x5B,
98        WAKE_UP_DUR        = 0x5C,
99        FREE_FALL          = 0x5D,
100        MD1_CFG            = 0x5E,
101        MD2_CFG            = 0x5F,
102     };
103
104     vector<int16_t> a; // accelerometer readings
105     vector<int16_t> g; // gyro readings
106     vector<float> dps;
107
108     //conversion factors
109     float mdps = 0;
110     float mg = 0;
111     //float odrGyro = 0;
112
113     uint8_t last_status; // status of last I2C transmission
114
115     LSM6(void);
116
117     bool init(deviceType device = device_auto, sa0State sa0 = sa0_auto);
118     deviceType getDeviceType(void) { return _device; }
119
120     void enableDefault(void);
121
122 public:
123     void writeReg(uint8_t reg, uint8_t value);
124     uint8_t readReg(uint8_t reg);
125
126     void readAcc(void);
127     void readGyro(void);
128     void read(void);
```

```
129
130     void setFullScaleGyro(GYRO_FS gfs);
131     void setFullScaleAcc(ACC_FS afs);
132
133     void setGyroDataOutputRate(ODR);
134     void setAccDataOutputRate(ODR);
135
136     void setTimeout(uint16_t timeout);
137     uint16_t getTimeout(void);
138     bool timeoutOccurred(void);
139
140     uint8_t getStatus(void) {return readReg(LSM6::STATUS_REG);}
141
142     // vector functions
143     //template <typename Ta, typename Tb, typename To> static void vector_cross(const vector<Ta> *a,
      const vector<Tb> *b, vector<To> *out);
144     //template <typename Ta, typename Tb> static float vector_dot(const vector<Ta> *a, const vector<Tb>
      *b);
145     //static void vector_normalize(vector<float> *a);
146
147   private:
148     deviceType _device; // chip type
149     uint8_t address;
150
151     uint16_t io_timeout;
152     bool did_timeout;
153
154     int16_t testReg(uint8_t address, regAddr reg);
155 };
156
157
158 // template <typename Ta, typename Tb, typename To> void LSM6::vector_cross(const vector<Ta> *a, const
      vector<Tb> *b, vector<To> *out)
159 // {
160 //   out->x = (a->y * b->z) - (a->z * b->y);
161 //   out->y = (a->z * b->x) - (a->x * b->z);
162 //   out->z = (a->x * b->y) - (a->y * b->x);
163 // }
164
165 // template <typename Ta, typename Tb> float LSM6::vector_dot(const vector<Ta> *a, const vector<Tb> *b)
166 // {
167 //   return (a->x * b->x) + (a->y * b->y) + (a->z * b->z);
168 // }
169
170 #endif
```

6.7 pcint.h

```
1 #include <Arduino.h>
2
9 void attachPCInt(uint8_t pcInt, void (*pcisr)(void));
10
11 uint8_t digitalPinToPCInterrupt(uint8_t pin);
```

6.8 PIDcontroller.h

```
1 #pragma once
2
3 #include <Arduino.h>
4
9 class PIDController
10 {
11 protected:
12     float Kp, Ki, Kd;
13     float currError = 0;
14     float prevError = 0;
15
16     float sumError = 0;
17     float errorBound = 0;
18
19     float deltaT = 0; //not used for now; could be useful
20
21     float currEffort = 0;
22
23 public:
24     PIDController(float p, float i = 0, float d = 0, float bound = 0) : Kp(p), Ki(i), Kd(d),
      errorBound(bound) {}
25
30     float calcEffort(float error);
```

```
31
32      float setKp(float k) {return Kp = k;}
33      float setKi(float k) {sumError = 0; return Ki = k;}
34      float setKd(float k) {return Kd = k;}
35      float setCap(float cap) {return errorBound = cap;}
36      void resetSum(void) {sumError = 0;}
37  };
```

6.9 src/Pushbutton.h File Reference

```
#include <Arduino.h>
```

Classes

- class PushbuttonBase

  General pushbutton class that handles debouncing.
- class Pushbutton

  Main class for interfacing with pushbuttons.

Macros

- #define PULL_UP_DISABLED 0
- #define PULL_UP_ENABLED 1
- #define DEFAULT_STATE_LOW 0
- #define DEFAULT_STATE_HIGH 1
- #define ZUMO_BUTTON 12

6.9.1 Detailed Description

This is the main header file for the Pushbutton library.

For an overview of the library's features, see https://github.com/pololu/pushbutton-arduino. That is the main repository for the library, though copies of the library may exist in other repositories.

6.9.2 Macro Definition Documentation

6.9.2.1 DEFAULT_STATE_HIGH

```
#define DEFAULT_STATE_HIGH 1
```

Indicates that the default (released) state of the button is when the I/O line reads high.

6.9.2.2 DEFAULT_STATE_LOW

```
#define DEFAULT_STATE_LOW 0
```

Indicates that the default (released) state of the button is when the I/O line reads low.

6.9.2.3 PULL_UP_DISABLED

```
#define PULL_UP_DISABLED 0
```

Indicates the that pull-up resistor should be disabled.

6.9.2.4 PULL_UP_ENABLED

```
#define PULL_UP_ENABLED 1
```

Indicates the that pull-up resistor should be enabled.

6.9.2.5 ZUMO_BUTTON

```
#define ZUMO_BUTTON 12
```

The pin used for the button on the Zumo Shield for Arduino.

This does not really belong here in this general pushbutton library and will probably be removed in the future.

6.10 Pushbutton.h

Go to the documentation of this file.

```
1 // Copyright Pololu Corporation.  For more information, see http://www.pololu.com/
2
12 #pragma once
13
14 #include <Arduino.h>
15
17 #define PULL_UP_DISABLED    0
18
20 #define PULL_UP_ENABLED     1
21
24 #define DEFAULT_STATE_LOW   0
25
28 #define DEFAULT_STATE_HIGH  1
29
35 #define ZUMO_BUTTON 12
36
37 // \cond
43 class PushbuttonStateMachine
44 {
45 public:
46
47   PushbuttonStateMachine();
48
51   bool getSingleDebouncedRisingEdge(bool value);
52
53 private:
54
55   uint8_t state;
56   uint16_t prevTimeMillis;
57 };
58 // \endcond
59
```

```
70 class PushbuttonBase
71 {
72 public:
73
79   void waitForPress();
80
86   void waitForRelease();
87
92   void waitForButton();
93
101   bool getSingleDebouncedPress();
102
114   bool getSingleDebouncedRelease();
115
122   virtual bool isPressed() = 0;
123
124 private:
125
126   PushbuttonStateMachine pressState;
127   PushbuttonStateMachine releaseState;
128 };
129
137 class Pushbutton : public PushbuttonBase
138 {
139 public:
140
154   Pushbutton(uint8_t pin, uint8_t pullUp = PULL_UP_ENABLED,
155       uint8_t defaultState = DEFAULT_STATE_HIGH);
156
157   virtual bool isPressed();
158
159 private:
160
161   void init()
162   {
163     if (!initialized)
164     {
165       initialized = true;
166       init2();
167     }
168   }
169
170   void init2();
171
172   bool initialized;
173   uint8_t _pin;
174   bool _pullUp;
175   bool _defaultState;
176 };
```

6.11 Rangefinder.h

```
1 #pragma once
2
3 #include <Arduino.h>
4
10 class Rangefinder
11 {
12 protected:
13     volatile uint8_t state = 0;
14
15     uint8_t echoPin = -1;
16     uint8_t trigPin = -1;
17
18     // for keeping track of ping intervals
19     uint32_t lastPing = 0;
20
21     // we set the pingInterval to 10 ms, but it won't actually ping that fast
22     // since it _only_ pings if the ECHO pin is low -- that is, it will ping
23     // in 10 ms or when the last echo is done, whichever is _longer_
24     uint32_t pingInterval = 10;
25
26     // for keeping track of echo duration
27     volatile uint32_t pulseStart = 0;
28     volatile uint32_t pulseEnd = 0;
29
30     // holds the last recorded distance
31     float distance = 99;
32
33 public:
34     Rangefinder(uint8_t echo, uint8_t trig);
35
36     // must call init() to set up pins and interrupts
```

```
37     void init(void);
38
39     // checks to see if it's time to emit a ping
40     uint8_t checkPingTimer(void);
41
42     // checks to see if an echo is complete
43     uint16_t checkEcho(void);
44
47     float getDistance(void);
48
49     // ISR for the echo pin
50     void ISR_echo(void);
51 };
52
53 // ISR for the echo
54 void ISR_Rangefinder(void);
55
56 // we declare as extern so we can refer to it in the ISR
57 extern Rangefinder rangefinder;
```

6.12 src/Romi32U4.h File Reference

Main header file for the Romi32U4 library.

```
#include <FastGPIO.h>
#include <Romi32U4Buttons.h>
#include <Romi32U4Encoders.h>
#include <Romi32U4Motors.h>
#include <pcint.h>
#include <Wire.h>
#include <LSM6.h>
```

Functions

- void ledRed (bool on)

  Turns the red user LED (RX) on or off.
- void ledYellow (bool on)

  Turns the yellow user LED on pin 13 on or off.
- void ledGreen (bool on)

  Turns the green user LED (TX) on or off.
- bool usbPowerPresent ()

  Returns true if USB power is detected.
- uint16_t readBatteryMillivolts ()

  Reads the battery voltage and returns it in millivolts.

6.12.1 Detailed Description

Main header file for the Romi32U4 library.

This file includes all the other headers files provided by the library.

6.12.2 Function Documentation

6.12.2.1 ledGreen()

```
void ledGreen (
            bool on )  [inline]
```

Turns the green user LED (TX) on or off.

Parameters

on	1 to turn on the LED, 0 to turn it off.

The green user LED is pin PD5, which is also known as TXLED. The Arduino core code uses this LED to indicate when it receives data over USB, so it might be hard to control this LED when USB is connected.

6.12.2.2 ledRed()

```
void ledRed (
             bool on )  [inline]
```

Turns the red user LED (RX) on or off.

Parameters

on	1 to turn on the LED, 0 to turn it off.

The red user LED is on pin 17, which is also known as PB0, SS, and RXLED. The Arduino core code uses this LED to indicate when it receives data over USB, so it might be hard to control this LED when USB is connected.

6.12.2.3 ledYellow()

```
void ledYellow (
             bool on )  [inline]
```

Turns the yellow user LED on pin 13 on or off.

Parameters

on	1 to turn on the LED, 0 to turn it off.

6.12.2.4 readBatteryMillivolts()

```
uint16_t readBatteryMillivolts ( )  [inline]
```

Reads the battery voltage and returns it in millivolts.

If this function returns a number below 5500, the actual battery voltage might be significantly lower than the value returned.

6.12.2.5 usbPowerPresent()

```
bool usbPowerPresent ( )  [inline]
```

Returns true if USB power is detected.

This function returns true if power is detected on the board's USB port and returns false otherwise. It uses the ATmega32U4's VBUS line, which is directly connected to the power pin of the USB connector.

See also

A method for detecting whether the board's virtual COM port is open: http://arduino.cc/en/↩Serial/IfSerial

6.13 Romi32U4.h

Go to the documentation of this file.

```
1 // Copyright Pololu Corporation.  For more information, see http://www.pololu.com/
2
10 #pragma once
11
12 #ifndef __AVR_ATmega32U4__
13 #error "This library only supports the ATmega32U4.  Try selecting A-Star 32U4 in the Boards menu."
14 #endif
15
16 #include <FastGPIO.h>
17 #include <Romi32U4Buttons.h>
18 #include <Romi32U4Encoders.h>
19 #include <Romi32U4Motors.h>
20 #include <pcint.h>
21
22 #include <Wire.h> //not used in many codes, but platformio balks without it, since it scans the IMU
       library
23 #include <LSM6.h>
24
32 inline void ledRed(bool on)
33 {
34     FastGPIO::Pin<17>::setOutput(!on);
35 }
36
40 inline void ledYellow(bool on)
41 {
42     FastGPIO::Pin<13>::setOutput(on);
43 }
44
52 inline void ledGreen(bool on)
53 {
54     FastGPIO::Pin<IO_D5>::setOutput(!on);
55 }
56
65 inline bool usbPowerPresent()
66 {
67     return USBSTA » VBUS & 1;
68 }
69
74 inline uint16_t readBatteryMillivolts()
75 {
76     const uint8_t sampleCount = 8;
77     uint16_t sum = 0;
78     for (uint8_t i = 0; i < sampleCount; i++)
79     {
80         sum += analogRead(A1);
81     }
82
83     // VBAT = 3 * millivolt reading = 3 * raw * 5000/1024
84     //      = raw * 1875 / 128
85     // The correction term below makes it so that we round to the
86     // nearest whole number instead of always rounding down.
87     const uint32_t correction = 64 * sampleCount - 1;
88     return ((uint32_t)sum * 1875 + correction) / (128 * sampleCount);
89 }
```

6.14 src/Romi32U4Buttons.h File Reference

```
#include <Pushbutton.h>
#include <FastGPIO.h>
#include <USBPause.h>
#include <util/delay.h>
```

Classes

- class Romi32U4ButtonA

  Interfaces with button A on the Romi 32U4.
- class Romi32U4ButtonB

  Interfaces with button B on the Romi 32U4.
- class Romi32U4ButtonC

  Interfaces with button C on the Romi 32U4.

Macros

- #define ROMI_32U4_BUTTON_A 14
- #define ROMI_32U4_BUTTON_B IO_D5
- #define ROMI_32U4_BUTTON_C 17

6.14.1 Macro Definition Documentation

6.14.1.1 ROMI_32U4_BUTTON_A

```
#define ROMI_32U4_BUTTON_A 14
```

The pin number for the pin connected to button A on the Romi 32U4.

6.14.1.2 ROMI_32U4_BUTTON_B

```
#define ROMI_32U4_BUTTON_B IO_D5
```

The pin number for the pin connected to button B on the Romi 32U4. Note that this is not an official Arduino pin number so it cannot be used with functions like digitalRead, but it can be used with the FastGPIO library.

6.14.1.3 ROMI_32U4_BUTTON_C

```
#define ROMI_32U4_BUTTON_C 17
```

The pin number for the pin conencted to button C on the Romi 32U4.

6.15 Romi32U4Buttons.h

Go to the documentation of this file.

```
1 // Copyright Pololu Corporation.  For more information, see http://www.pololu.com/
2
5 #pragma once
6
7 #include <Pushbutton.h>
8 #include <FastGPIO.h>
9 #include <USBPause.h>
10 #include <util/delay.h>
11
13 #define ROMI_32U4_BUTTON_A 14
14
18 #define ROMI_32U4_BUTTON_B IO_D5
19
21 #define ROMI_32U4_BUTTON_C 17
22
24 class Romi32U4ButtonA : public Pushbutton
25 {
26 public:
27     Romi32U4ButtonA() : Pushbutton(ROMI_32U4_BUTTON_A)
28     {
29     }
30 };
31
42 class Romi32U4ButtonB : public PushbuttonBase
43 {
44 public:
45
46     virtual bool isPressed()
47     {
48         USBPause usbPause;
49         FastGPIO::PinLoan<ROMI_32U4_BUTTON_B> loan;
50         FastGPIO::Pin<ROMI_32U4_BUTTON_B>::setInputPulledUp();
51         _delay_us(3);
52         return !FastGPIO::Pin<ROMI_32U4_BUTTON_B>::isInputHigh();
53     }
54 };
55
66 class Romi32U4ButtonC : public PushbuttonBase
67 {
68 public:
69
70     virtual bool isPressed()
71     {
72         USBPause usbPause;
73         FastGPIO::PinLoan<ROMI_32U4_BUTTON_C> loan;
74         FastGPIO::Pin<ROMI_32U4_BUTTON_C>::setInputPulledUp();
75         _delay_us(3);
76         return !FastGPIO::Pin<ROMI_32U4_BUTTON_C>::isInputHigh();
77     }
78 };
```

6.16 src/Romi32U4Encoders.h File Reference

6.17 Romi32U4Encoders.h

Go to the documentation of this file.

```
1 // Copyright Pololu Corporation.  For more information, see http://www.pololu.com/
2
5 #pragma once
6
```

6.18 src/Romi32U4Motors.h File Reference

```
#include <Arduino.h>
#include <stdint.h>
#include <PIDController.h>
```

Classes

- class Romi32U4Motor

  Controls motor effort and direction on the Romi 32U4.
- class LeftMotor
- class RightMotor

6.19 Romi32U4Motors.h

Go to the documentation of this file.

```
1 // Adapted from a library by Pololu Corporation.  For more information, see http://www.pololu.com/
2
5 #pragma once
6
7 #include <Arduino.h>
8 #include <stdint.h>
9 #include <PIDController.h>
10
24 class Romi32U4Motor
25 {
26 protected:
27    // Used to control the motors in different ways
28    enum CTRL_MODE : uint8_t {CTRL_DIRECT, CTRL_SPEED, CTRL_POS};
29    volatile CTRL_MODE ctrlMode = CTRL_DIRECT;
30
31    // this is the 'speed' of the motor, in "encoder counts / 16 ms interval"
32    volatile int16_t speed = 0;
33
34    // used to set target speed or position (or both)
35    int16_t targetSpeed = 0;
36    int16_t targetPos = 0;
37
38    // Maximum effort (to protect the gear boxes). Can be changed by setting turbo mode
39    int16_t maxEffort = 300;
40
41    // keeps track of encoder changes
42    volatile int16_t prevCount = 0;
43    volatile int16_t count = 0;
44    volatile int16_t lastA = 0;
45    volatile int16_t lastB = 0;
46
47    // We build a PID controller into the object for controlling speed
48    PIDController pidCtrl;
49
50    friend class Chassis;
51
52 public:
53    Romi32U4Motor(void) : pidCtrl(1, 0) {}
54
58    static void init()
59    {
60       initMotors();
61       initEncoders();
62    }
63
64    void setPIDCoeffients(float kp, float ki)
65    {
66       pidCtrl.setKp(kp);
67       pidCtrl.setKi(ki);
68    }
69
70 protected:
71    // Used to set up motors and encoders. Do not call directly; they are called from init(), which
72    // is called from Chassis::init()
73    static void initMotors();
74    static void initEncoders();
75
82   virtual void setEffort(int16_t effort) = 0;
83
92    int16_t getCount(void);
93    int16_t getAndResetCount(void);
94
95    void setTargetSpeed(int16_t targetSpeed);
96    void moveFor(int16_t amount);//, int16_t speed);
97    bool checkComplete(void) {return ctrlMode == CTRL_DIRECT;}
98
99    void update(void);
100    void calcEncoderDelta(void);
```

```
101
102 public:
116     void allowTurbo(bool turbo);
117
121     inline void handleISR(bool newA, bool newB);
122 };
123
130 class LeftMotor : public Romi32U4Motor
131 {
132 protected:
133     void setEffort(int16_t effort);
134
135 public:
136     void setMotorEffort(int16_t effort)
137     {
138         ctrlMode = CTRL_DIRECT;
139         setEffort(effort);
140     }
141 };
142
143 class RightMotor : public Romi32U4Motor
144 {
145 protected:
146     void setEffort(int16_t effort);
147
148 public:
149     void setMotorEffort(int16_t effort)
150     {
151         ctrlMode = CTRL_DIRECT;
152         setEffort(effort);
153     }
154 };
```

6.20 servo32u4.h

```
1 #pragma once
2
3 #include <Arduino.h>
4
14 class Servo32U4
15 {
16 private:
17     uint16_t usMin = 1000;
18     uint16_t usMax = 2000;
19
20     uint8_t feedbackPin = -1;
21     bool isAttached = false;
22
23 public:
24     void attach(void);
25     void detach(void);
26     void writeMicroseconds(uint16_t microseconds);
27     uint16_t setMinMaxMicroseconds(uint16_t min, uint16_t max);
28 };
```

6.21 Timer.h

```
1 #pragma once
2
3 class Timer
4 {
5 public:
6     Timer(unsigned long interval);
7     bool isExpired();
8     void reset();
9     void reset(unsigned long newInterval);
10
11 private:
12     unsigned long expiredTime;
13     unsigned long timeInterval;
14 };
```

6.22 src/USBPause.h File Reference

```
#include <avr/io.h>
```

Classes

- class USBPause

6.22.1 Detailed Description

This is the main file for the USBPause library.

For an overview of this library, see https://github.com/pololu/usb-pause-arduino. That is the main repository for this library, though copies may exist in other repositories.

6.23 USBPause.h

Go to the documentation of this file.

```
1 // Copyright Pololu Corporation.  For more information, see https://www.pololu.com/
2
11 #pragma once
12
13 #include <avr/io.h>
14
26 class USBPause
27 {
29     uint8_t savedUDIEN;
30
32     uint8_t savedUENUM;
33
35     uint8_t savedUEIENX0;
36
37 public:
38
39     USBPause()
40     {
41         // Disable the general USB interrupt.  This must be done
42         // first, because the general USB interrupt might change the
43         // state of the EP0 interrupt, but not the other way around.
44         savedUDIEN = UDIEN;
45         UDIEN = 0;
46
47         // Select endpoint 0.
48         savedUENUM = UENUM;
49         UENUM = 0;
50
51         // Disable endpoint 0 interrupts.
52         savedUEIENX0 = UEIENX;
53         UEIENX = 0;
54     }
55
56     ~USBPause()
57     {
58         // Restore endpoint 0 interrupts.
59         UENUM = 0;
60         UEIENX = savedUEIENX0;
61
62         // Restore endpoint selection.
63         UENUM = savedUENUM;
64
65         // Restore general device interrupt.
66         UDIEN = savedUDIEN;
67     }
68
69 };
```

6.24 wpi-32u4-lib.h

```
1 #pragma once
2
3 #include <Romi32U4Motors.h>
4 #include <Romi32U4Buttons.h>
5
6 // line sensor definitions
7 #define  RIGHT_LINE_SENSE A4
8 #define  LEFT_LINE_SENSE A3
```