RobotOpen Documentation

Release 1.0

Eric Barch

Sep 27, 2017

Contents

Arduino Library Classes:

CHAPTER 1

RobotOpen

The RobotOpen class is the primary class that manages connectivity, loop scheduling, and general operation of the robot. A single instance of this class is created automatically and can be accessed statically.

Functions

`void RobotOpen.setIP(IPAddress newIP)`
: This should be called only once in the setup function of the robot code. It changes the robot's default IP of 10.0.0.22 to the given IP address. The IPAddress format is given as 4 comma delimited values. For example: RobotOpen.setIP(192,168,1,22)

`void RobotOpen.setTimeout(int timeout_in_ms)`
: By the default the robot will consider itself disabled when a packet has not been received in a period greater than 200 milliseconds. This can be overridden using this function by defining the number of milliseconds until the robot is disabled.

`void RobotOpen.begin(*enabledCallback, *disabledCallback, *timedtasksCallback)`
: This function should be called once in the setup function of the robot code with a reference to an enable function, disable function, and timed tasks function. The enabled function is only called when the robot is enabled. The disabled function is only called when the robot is disabled. Timed tasks will execute continuously while the robot is powered on.

`void RobotOpen.syncDS()`
: This is used internally by the RobotOpen library to process data and keep the robot operational. This is the only call that should be in your Arduino loop() function. It should not be called anywhere else.

`int RobotOpen.numJoysticks()`
: Returns the number of joysticks currently being sent from the driver station.

CHAPTER 2

ROAnalog

ROAnalog gives you access to the analog inputs on your RobotOpen board.

Constructor

`ROAnalog(uint8_t channel)`
Creates an ROAnalog instance with the given analog channel (the first channel typically being 0).

Functions

int **`read`**`()`
Read the samples value that can range from 0-1023 (typically proportional to 0-5V).

Examples

```
// create an ROAnalog instance on analog channel 1
ROAnalog steerPOT(1);

// get current reading (0-1023)
int read();
```

CHAPTER 3

ROBlackBox

ROBlackBox offers SD card based logging at regular intervals for later analysis. NOTE: This class is only available on the Sasquatch robot controller.

Functions

`void ROBlackBox.log(String data)`
Log the given string to the SD card.

Examples

```
// log "some info" to the SD card
ROBlackBox.log("some info");
```

CHAPTER 4

RODashboard

RODashboard allows for arbitrary data transmission to the dashboard to de displayed, graphed, or saved in real time. Variables can either be sent over to be added to the list of values or debug messages can be displayed in the virtual console.

Functions

`void RODashboard.debug(String data)`
Log the given string to the virtual console.

`void RODashboard.publish(String label, byte val)`
Transmit the provided byte with the specified label to the dashboard.

`void RODashboard.publish(String label, int val)`
Transmit the provided int with the specified label to the dashboard.

`void RODashboard.publish(String label, long val)`
Transmit the provided long with the specified label to the dashboard.

`void RODashboard.publish(String label, float val)`
Transmit the provided float with the specified label to the dashboard.

Examples

```
// send over the answer
int var = 42;
RODashboard.publish("name", var);

// log to the console
RODashboard.debug("this is a debug statement");
```

CHAPTER 5

RODigitalIO

RODigitalIO provides access to the digital pins on your RobotOpen board. Pins can either be configured as inputs or outputs. When a pin is configured as an input, it will be read as either a high or low state. When configured as an output, the pin can be set either high or low. A low state will generate a voltage of 0V. A high state will generate a 5V signal on that pin.

Constructor

`RODigitalIO(uint8_t channel, uint8_t mode)`
: Creates an RODigitalIO instance with the given digital channel (the first channel typically being 0). The second parameter can either be INPUT or OUTPUT.

Functions

void **`on`**()
: OUTPUT only. Toggle the OUTPUT high.

void **`off`**()
: OUTPUT only. Toggle the OUTPUT low.

boolean **`read`**()
: INPUT only. Returns true if the input read is high, otherwise returns false.

void **`pullUp`**()
: INPUT only. Uses the Arduino's internal pull-up resistor to drive the line to 5V by default.

void **`allowFloat`**()
: INPUT only. Disables the pullup resistor, if enabled.

void **`setMode`**(uint8_t *mode*)
: Change the mode of the digital pin. Can be either INPUT or OUTPUT.

Examples

```
// create some objects
RODigitalIO redLED(3, OUTPUT);
RODigitalIO blueButton(2, INPUT);

// output calls
redLED.on();

// input calls
blueButton.pullUp();
bool is_held = blueButton.read();
blueButton.allowFloat();

// make blueButton an output
blueButton.setMode(OUTPUT);
```

CHAPTER 6

ROEncoder

ROEncoder provides access to the quadrature encoder inputs on your RobotOpen board. Using ROEncoder, the number of interrupts on each quad input can be read.

Constructor

`ROEncoder(uint8_t channel)`
Creates an ROEncoder instance with the given encoder channel (the first channel typically being 0).

Functions

long **`read`**()
Returns the current 32 bit value of the encoder count.

long **`write`**(int32_t *new_value*)
NOTE: Sasquatch boards only. Sets the encoder count to a new value.

void **`reset`**()
NOTE: Gorgon boards only. Resets the encoder count to 0.

float **`readCPS`**()
NOTE: Gorgon boards only. Returns the current number of counts per second.

void **`setSensitivity`**(uint16_t *sensitivity*)
NOTE: Gorgon boards only. Used to adjust how often the coprocessor recalculates encoder counts and CPS. By default this is set to 4 samples. If you have an encoder that generates a very large number of counts per rotation (greater than a couple hundred), you may want to experiment with raising this number.

void **`setCPSSamplesToAverage`**(uint8_t *samples*)
NOTE: Gorgon boards only. Sets the number of samples used in the averaging of the CPS value. Larger numbers will react less quickly, while lower numbers will be jumpier. The default is set to 9.

Examples

```
// create an encoder object
ROEncoder leftDriveEncoder(1);

// read the encoder count
long mycount = leftDriveEncoder.read();
```

CHAPTER 7

ROJoystick

ROJoystick provides access to any controller or joystick data transmitted by the driver station.

Constructor

`ROJoystick(uint8_t joystick_number)`
Creates an ROJoystick instance with the given joystick number. This can range from 1 to 4 and is mapped in the driver station.

Functions

byte **`leftX`**()
Returns a value from 0-255 representing the position of the left joystick's X axis.

byte **`leftY`**()
Returns a value from 0-255 representing the position of the left joystick's Y axis.

byte **`rightX`**()
Returns a value from 0-255 representing the position of the right joystick's X axis.

byte **`rightY`**()
Returns a value from 0-255 representing the position of the right joystick's Y axis.

boolean **`btnA`**()
Returns true if button A is held. False otherwise.

boolean **`btnB`**()
Returns true if button B is held. False otherwise.

boolean **`btnX`**()
Returns true if button X is held. False otherwise.

boolean **btnY**()
Returns true if button Y is held. False otherwise.

boolean **btnLShoulder**()
Returns true if the left shoulder is held. False otherwise.

boolean **btnRShoulder**()
Returns true if the right shoulder is held. False otherwise.

byte **lTrigger**()
Returns a value from 0-255 (hardware permitting) representing the pressure on the left trigger.

byte **rTrigger**()
Returns a value from 0-255 (hardware permitting) representing the pressure on the right trigger.

boolean **btnSelect**()
Returns true if the select button is held. False otherwise.

boolean **btnStart**()
Returns true if the start button is held. False otherwise.

boolean **btnLStick**()
Returns true if the left stick is pushed down. False otherwise.

boolean **btnRStick**()
Returns true if the right stick is pushed down. False otherwise.

boolean **dPadUp**()
Returns true if the D-Pad up button is held. False otherwise.

boolean **dPadDown**()
Returns true if the D-Pad down button is held. False otherwise.

boolean **dPadLeft**()
Returns true if the D-Pad left button is held. False otherwise.

boolean **dPadRight**()
Returns true if the D-Pad right button is held. False otherwise.

byte **auxOne**()
Returns a value from 0-255 for the aux 1 channel. Note: This allows for future expansion within the RobotOpen protocol. Custom joystick implementations could send this.

byte **auxTwo**()
Returns a value from 0-255 for the aux 2 channel. Note: This allows for future expansion within the RobotOpen protocol. Custom joystick implementations could send this.

byte **auxThree**()
Returns a value from 0-255 for the aux 3 channel. Note: This allows for future expansion within the RobotOpen protocol. Custom joystick implementations could send this.

byte **auxFour**()
Returns a value from 0-255 for the aux 4 channel. Note: This allows for future expansion within the RobotOpen protocol. Custom joystick implementations could send this.

Examples

```
// create a reference to joystick #1
ROJoystick usb1(1);
```

```
// get the position of the left joystick's x axis
byte = usb1.leftX();

// get the button status of button A
bool btnA = usb1.btnA();
```

CHAPTER 8

ROParameter

ROParameter allows for on-the-fly configuration of your RobotOpen controller from the dashboard. ROParameters are variables that can be modified straight from the driver station without recompiling and flashing robot code. Supported classes: ROBoolParameter, ROCharParameter, ROIntParameter, ROLongParameter, ROFloatParameter. Note that your robot must be disabled for this values to be updated from the driver station.

Constructors

`ROCharParameter(String label, uint8_t id)`
Creates an ROCharParameter instance with the given id. Note that no two parameters may have the same id.

`ROIntParameter(String label, uint8_t id)`
Creates an ROIntParameter instance with the given id. Note that no two parameters may have the same id.

`ROLongParameter(String label, uint8_t id)`
Creates an ROLongParameter instance with the given id. Note that no two parameters may have the same id.

`ROFloatParameter(String label, uint8_t id)`
Creates an ROFloatParameter instance with the given id. Note that no two parameters may have the same id.

Functions

`[char, int, long, float] read()`
Returns the current stored value for the parameter, depending upon the class used.

Examples

```
// create parameter objects
ROFloatParameter pConstant("pconstant", 0);
ROFloatParameter iConstant("iconstant", 1);
ROFloatParameter dConstant("dconstant", 2);

// use parameters as PID constants
void loop() {
        runPID(pConstant.get(), iConstant.get(), dConstant.get());
}
```

CHAPTER 9

ROPWM

ROPWM allows for control of the PWM outputs on the RobotOpen board. Pulse width modulation is frequently used for electronic speed controllers (ESC) or servos. Valid range is between 0-255, 127 being neutral.

Constructor

`ROPWM(uint8_t channel)`
Creates an ROPWM instance with the given PWM channel (the first channel typically being 0).

Functions

void **`write`**(uint8_t *setpoint*)
Set the PWM channel to the given value. Automatically attaches a PWM channel if detached.

void **`detach`**()
NOTE: Supported on Sasquatch and Gorgon only. Detaches a PWM channel after being attached. This will neutral out most speed controllers and disable servos.

void **`attach`**()
NOTE: Supported on Sasquatch and Gorgon only. Re-attaches a PWM channel after being detached.

Examples

```
// create a left drive ROPWM object on channel 2
ROPWM leftDrive(2);

// set it to 200
leftDrive.write(200);
```

CHAPTER 10

ROSolenoid

ROSolenoid allows for control of the solenoid outputs on the RobotOpen board. Solenoids are either 12V or 24V and can be set in either a high or low state.

Constructor

`ROSolenoid(uint8_t channel)`
Creates an ROSolenoid instance with the given solenoid channel (the first channel typically being 0).

Functions

void **`on`**`()`
Applies power to the specified solenoid.

void **`off`**`()`
Disables power to the specified solenoid.

Examples

```
// create an ROSolenoid object on channel 2
ROSolenoid giantArm(2);

// turn the giant arm on!
giantArm.on();
```

CHAPTER 11

ROStatus

ROStatus provides current robot status such as enabled state, power level, and driver station connection information.

Functions

`float ROStatus.batteryReading()`
: On supported boards, returns the main battery voltage when the BMC jumper is selected.

`boolean ROStatus.isEnabled()`
: Returns true if the robot is enabled. False otherwise.

`int ROStatus.uptimeSeconds()`
: Returns the number of seconds that the RobotOpen board has been powered up for.

CHAPTER 12

ROTimer

ROTimer allows for simple time-based loop execution and delays. By utilizing ROTimer, a certain portion of code can be run at a specific speed (processor time permitting) or "ready'd" for execution.

Constructor

class `ROTimer`
Creates an ROTimer instance.

Functions

void **`queue`**(uint16_t *milliseconds*)
Schedule the timer to expire within the number of milliseconds specified.

boolean **`ready`**()
Returns true if the timer has expired, false otherwise.

Examples

```
// create timers
ROTimer step1;
ROTimer step2;
ROTimer step3;
ROTimer repeatingLoop;

void setup() {
        // schedule timers to fire immediately
        step1.queue(0);
        repeatingLoop.queue(0);
```

```
}


void loop() {
        if (step1.ready()) {
                // do something
                step2.queue(1000);
        }
        if (step2.ready()) {
                // do something else
                step3.queue(2000);
        }
        if (step3.ready()) {
                // last step of the process, now repeat
                step1.queue(0);
        }

        if (repeatingLoop.ready()) {
                // do something every 1000ms
                repeatingLoop.queue(1000);
        }
}
```

RobotOpen Protocol

CHAPTER 13

RobotOpen Protocol

The creation of the RobotOpen Protocol is an effort to design an open, standardized protocol for communication and control of hobbyist robots over an IP based network. It is designed for simple and lightweight two-way communication. This version of the RobotOpen protocol supersedes any prior documentation and is used in all current RobotOpen software.

Basics

By default, any robot controller implementing the RobotOpen protocol should be disabled and inactive upon powerup. At any point that no packets are received for a timeout period (typically 100-200 milliseconds), the robot should also disable itself. All RobotOpen protocol data is currently intended to traverse an IP based network, specifically utilizing UDP on the transport layer. The RobotOpen protocol is stateless. Every packet should be treated individually and does not rely on any specific ordering.

Parameters

Parameters are semi-permanent values intended to be stored in non-volatile EEPROM memory onboard the Robot. These values can be read from the robot and updated from the Driver Station when the robot controller is disabled. Currently RobotOpen protocol supports the following types listed below. Each parameter has a unique address assigned to it (a number between 0 and 99). Parameters are always transmitted as 4 bytes, even if their size is less than 4. Additional bytes should be set to 0. For instance a boolean that is true would be transmitted as 0xFF000000 in a parameter packet. Values are transmitted in big endian format (most significant byte first).

Parameter Types

Bool - 1 byte (0x00 false, 0xFF true)
Char - 1 byte (signed)
Int - 2 bytes (signed)
Long - 4 bytes (signed)

Float - 4 bytes (signed)

Driver Station to Robot Packets

Currently the RobotOpen protocol supports up to four control devices (each having a fixed size of 24 bytes, totalling 96 bytes of control data) per packet. These 24 bytes have specified names but arbitrary data can be inserted and interpreted on the robot controller.

Joystick/Control Naming

Byte 0 - **Analog Left X Axis** (0x00-0xFF)
Byte 1 - **Analog Left Y Axis** (0x00-0xFF)
Byte 2 - **Analog Right X Axis** (0x00-0xFF)
Byte 3 - **Analog Right Y Axis** (0x00-0xFF)
Byte 4 - **Button A** (0x00 off, 0xFF on)
Byte 5 - **Button B** (0x00 off, 0xFF on)
Byte 6 - **Button X** (0x00 off, 0xFF on)
Byte 7 - **Button Y** (0x00 off, 0xFF on)
Byte 8 - **Left Shoulder** (0x00 off, 0xFF on)
Byte 9 - **Right Shoulder** (0x00 off, 0xFF on)
Byte 10 - **Left Trigger** (0x00-0xFF)
Byte 11 - **Right Trigger** (0x00-0xFF)
Byte 12 - **Select** (0x00 off, 0xFF on)
Byte 13 - **Start** (0x00 off, 0xFF on)
Byte 14 - **Left Stick Button** (0x00 off, 0xFF on)
Byte 15 - **Right Stick Button** (0x00 off, 0xFF on)
Byte 16 - **D-Pad Up** (0x00 off, 0xFF on)
Byte 17 - **D-Pad Down** (0x00 off, 0xFF on)
Byte 18 - **D-Pad Left** (0x00 off, 0xFF on)
Byte 19 - **D-Pad Right** (0x00 off, 0xFF on)
Byte 20 - **Aux 1** (0x00-0xFF)
Byte 21 - **Aux 2** (0x00-0xFF)
Byte 22 - **Aux 3** (0x00-0xFF)
Byte 23 - **Aux 4** (0x00-0xFF)

Control (Enable) Packet

A control packet must contain at least one controller's worth of data. This means that a control packet can be 27, 51, 75, or 99 bytes long. Control packets with no joystick data will be dropped on the robot side. A CRC-16 of the packet is calculated and appended on the end. When a valid control packet is received on the robot side, the robot will enable itself.

Byte 0 - 0x63
(Controller 1 Bytes 0-23)
(Controller 2 Bytes 0-23) - optional

(Controller 3 Bytes 0-23) - optional
(Controller 4 Bytes 0-23) - optional
2nd to Last Byte - CRC-16 Low Byte
Last Byte - CRC-16 High Byte

Heartbeat (Disable) Packet

Heartbeat packets are used to disable the robot but keep communications active. All heartbeat packets are 3 fixed bytes, show below.

Byte 0 - 0x68
Byte 1 - 0xEE
Byte 2 - 0x01

Set Parameter Packet

The robot must be disabled for the parameters to be successfully set. Multiple parameters may be transmitted in each set parameter packet. Each parameter will consume 5 bytes (address + 4 data bytes).

Byte 0 - 0x73
...
(Parameter Address (0x00-0x63)
Val1
Val2
Val3
Val4
...
2nd to Last Byte - CRC-16 Low Byte
Last Byte - CRC-16 High Byte

Get Parameters Packet

A get parameters packet requests the robot controller to transmit all current parameters and their values back to the Driver Station. All get parameters packets are 3 fixed bytes, shown below.

Byte 0 - 0x67
Byte 1 - 0xEA
Byte 2 - 0x41

Robot to Driver Station Packets

Robot to Driver Station packets can be transmitted at any time. While a robot controller is disabled (heartbeat packets being sent), it should continue to receive debug and dashboard packets regularly. Parameter packets will be received

any time requested.

Debug Packet

Debug packets allow for arbitrary ASCII data to be sent to the Driver Station (a sort of remote serial console). Debug packets begin with the byte 0x70 followed by a variable number of ascii bytes.

Byte 0 - 0x70
(ascii bytes of variable length)

Dashboard Packet

A dashboard packet contains values that a user has published from their robot code so that they can monitor them remotely on their Driver Station. The supported types can be seen above. Unlike parameters, dashboard values are exactly their specified length in the packet. Every dashboard value in a dashboard packet begins with a length (the length of that dashboard value including the length byte). The type will be one of the type codes listed below. The appropriate number of bytes will follow based on the value's type. The ID is a variable length ascii name for the value.

Byte 0 - 0x64
...
Length
Type
Val1
Val2 (optional)
Val3 (optional)
Val4 (optional)
ID (variable length)
...

Dashboard Types

Char - 1 byte (unsigned) - (type code: 0x63)
Int - 2 bytes (signed) - (type code: 0x69)
Long - 4 bytes (signed) - (type code: 0x6C)
Float - 4 bytes (signed) - (type code: 0x66)

Parameters Packet

A parameters packet is sent back upon the reception of a get parameters packet. Each parameter in the parameters packet begins with a length (the length of the parameter including the length byte). This is followed by the unique parameter address (0-99). The type follows that (reference the parameter type codes below). 4 bytes will follow, however based on the type the non-used bytes will be zeroes. The ID is a variable length ascii name for the parameter.

Byte 0 - 0x72

...
Length
Parameter Address (0x00-0x63)
Type
Val1
Val2
Val3
Val4
ID (variable length)
...

Parameter Type Codes

Bool - 0x62
Char - 0x63
Int - 0x69
Long - 0x6C
Float - 0x66

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Index