Arduino Toolkit 0.9.0

a somewhat MATLAB compatible Arduino toolkit for GNU Octave.

John Donoghue

Distribution

The GNU Octave arduino package is *free* software. Free software is a matter of the users' freedom to run, copy, distribute, study, change and improve the software. This means that everyone is free to use it and free to redistribute it on certain conditions. The GNU Octave arduino package is not, however, in the public domain. It is copyrighted and there are restrictions on its distribution, but the restrictions are designed to ensure that others will have the same freedom to use and redistribute Octave that you have. The precise conditions can be found in the GNU General Public License that comes with the GNU Octave arduino package and that also appears in Appendix A [Copying], page 91.

To download a copy of the GNU Octave arduino package, please visit `http://octave.sourceforge.net/arduino/`.

Table of Contents

1 Installing and loading

The Arduino toolkit must be installed and then loaded to be used.

It can be installed in GNU Octave directly from octave-forge, or can be installed in an off-line mode via a downloaded tarball.

NOTE

The toolkit requires the Arduino IDE in order to program Arduino devices.

NOTE

The toolkit has a dependency on the instrument-control package, so it must be installed in order to successfully install the Arduino toolkit

The toolkit must be then be loaded once per each GNU Octave session in order to use its functionality.

1.1 Online Direct install

With an internet connection available, the Arduino package can be installed from octave-forge using the following command within GNU Octave:

```
pkg install -forge arduino
```

The latest released version of the toolkit will be downloaded and installed.

1.2 Off-line install

With the arduino toolkit package already downloaded, and in the current directory when running GNU Octave, the package can be installed using the following command within GNU Octave:

```
pkg install arduino-0.9.0.tar.gz
```

1.3 Loading

Regardless of the method of installing the Arduino toolkit, in order to use its functions, the toolkit must be loaded using the pkg load command:

```
pkg load arduino
```

The toolkit must be loaded on each GNU Octave session.

2 Hardware setup

In order to use the arduino hardware with the toolkit, it must be programmed with special firmware.

2.1 Programming the Arduino

To program the hardware, using a default configuration, run the arduinosetup command:

```
arduinosetup
```

A temporary Arduino project will be created, with the Arduino toolkit files copied to it and the Arduino IDE will open.

Set the board type and port correctly for the connected Arduino and press the upload button on the IDE.

The sources will be compiled and then uploaded to the connected arduino board.

After successful upload the Arduino IDE should be closed.

NOTE

The arduino programming is not compatible with the Matlab arduino library, so must be programmed by the Octave Arduino toolkit to to communicate to the arduino, even if it was previously used to work with Matlab.

NOTE

The toolkit requires the Arduino IDE in order to program the Arduino device. If the toolkit can not find the IDE, run the IDE manually, close it and retry programming the Arduino.

2.2 Known Arduino Board Types

The board type must be known in order to successfully detect and connect to the Arduino board after programming.

Currently, known boards are:

- Arduino Due
- Arduino UNO
- Arduino Mega 2560
- Arduino Nano
- Arduino Nano Every
- Arduino Nano 33 BLE
- Arduino Nano RP2040 Connect
- Arduino Pro/Pro Mini
- Arduino Pro Micro
- Arduino Leonardo
- Arduino Micro
- Arduino MKR1000
- Arduino MKRZero
- Sparkfun SAMD21
- Arduino Lilypad

- Arduino UNO WiFi rev2

  NOTE
  The Arduino servo library code may require modifications to eliminate conflicts between servos and the tone library

- Raspberry Pi Pico

Additional boards can be added easily, however require minor code changes.

3 Connecting to an arduino

To control an arduino device, a connection must be made to it by creating an arduino object.

3.1 Connecting to a single arduino

Assuming a single arduino device is connected to the computer, creating an arduino object with no arguments will find the connected arduino and connect to it:

```
ar = arduino()
```

3.2 Connecting to a specific arduino

Where multiple arduinos may be connected to the computer, a specific board can be connected by specifying the name of the port it is connected to:

```
ar = arduino("/dev/ttyACM0")
```

The port name will be operating system dependent.

3.3 Querying available arduinos

To list the ports of all *programmed* available arduinos, the scanForArduinos function can be used:

```
scanForArduinos
```

It will provide a list of all available boards it can find with the port they are connected to.

NOTE

The scanForArduinos function will only detect boards that have been programmed using the arduinosetup command

4 Basic Input and Output Overview

Basic input and output can be performed on a connected arduino device using by calling the read and write functions for a specific named pin on the arduino.

A list of available pins can get found from the pins property of the connected arduino object and are also displayed as part of the default shown properties:

```
ar = arduino();
% get the pin names
pins = ar.availablepins
```

Pin generally follow a naming scheme of D<number> for digital pins and A<number> for analog pins.

Digital pins can be used to read and write digital data, but can not read analog voltages. Analog pins can perform digital I/O as well as reading voltages.

4.1 Performing Digital I/O

A pin's digital logic value can be true (1) or false (0) and can be set using the writeDigitalPin function.

The following example attempts to set the D2 pin of the connected arduino object "ar" to true, waits 5 seconds and then sets it to false:

```
writeDigitalPin (ar,  "d2", true);
pause 5
writeDigitalPin (ar,  "d2", false);
```

Using the readDigitalPin will read the current logic state of the pin.

```
value = readDigitalPin (ar,  "d2");
```

4.2 Performing Analog Input

For analog pins, the voltage level can be read using a analog to digital conversion and will return a voltage level between 0 and the boards voltage (nominally 5V):

```
value = readVoltage (ar,  "a0");
```

The raw digital value of the pin can also be read instead of a voltage, giving a value between 0 and 2^x where x is the number of bits used by the analog to digital converter.

```
value = readAnalogPin (ar,  "a0");
```

5 Protocol based I/O Overview

The arduino toolkit supports more complex I/O for SPI, I2C, Servo control and more.

5.1 SPI communication

SPI communication can be performed by creating a SPI device object and then calling the writeRead function:

```
spi = device (ar,  "spichipselectpin", "d2");
```

The function call expects a connected arduino object as the first argument, followed by the chip select pin of the SPI device.

After a device is created, a write to device followed by read can can be made using the writeRead function:

```
spi = device (ar,  "spichipselectpin", "d2");
data = writeRead (spi,  100);
```

5.2 I2C communication

I2C communication can be performed by creating an I2C device object for a specific I2C address.

The following example creates an I2C device that will communicate with a I2C device at address 100"

```
i2c = device (ar,  "i2caddress", 100);
```

After creating an I2C device, data can be read and written using read, write, readRegister and writeRegister. The data to send and receive will be device dependent.

5.3 Servo communication

Servo communication can be performed after creating a servo device object to operate on a PWM pin:

```
servoobj = servo(ar, "d9", "minpulseduration", 1.0e-3, ...
  "maxpulseduration", 2e-3);
```

The servo function expects the connected arduino object and the PWM pin that the servo is connected to. Optional properties can be specified to control the setup of device.

In the example, the min and max pulse width values are set.

Using the servo object the current position and be read or set with values ranging between 0 to 1, with 0 being the minimum pulse width and 1 being the maximum.

The following example sets the servo to its middle position.

```
servoobj = servo(ar, "d9", "minpulseduration", 1.0e-3, ...
  "maxpulseduration", 2e-3);

writePosition (servoobj, 0.5);
```

5.4 Shift Registers

A shift register can be controlled by creating a shiftRegister object:

```
registerobj = shiftRegister(ar, '74hc164', "d2", "d3");
```

The parameters required are dependent on the type of shift register created.

Once a register object has been created, it can be read and written to using the read and write functions.

5.5 Rotary Encoders

A rotary encoder can be created by creating a rotaryEncoder object.

```
encoder = rotaryEncoder(ar, "d2", "d3", 180);
```

Using the created object, the rotary encoder value and speed can be read.

5.6 Ultrasonic Sensors

An ultrasonic sensor can be read by creating an ultrasonic object.

```
sensor = ultrasonic(ar, "d9", "d10");
```

Using the created object, the sensor distance and echo time and be read.

5.7 Serial communication

Serial communication can be performed on devices that support multiple serial devices such as the leonardo and mega2560 boards. The communications port to Octave is reserved and can not be used as a user controlled communications port.

Serial communication can be performed by creating a serial device object and then calling the read and write functions:

```
ser = device (ar,  "serial", 1);
```

The function call expects a connected arduino object as the first argument, followed "serial" and serial id.

After a device is created, the device can be read and written:

```
ser = device (ar,  "serial", 1);
write(ser, "hello");
data = read(ser, 100);
```

6 Addons Overview

This chapter provides an overview of the arduino package addon functionality for adding additional addons to arduino.

6.1 Addon Introduction

Addons provide a way of adding additional functionality to the arduino toolkit that provides Matlab access directly to the arduino hardware.

Addons are implemented in two parts.

1. code running on the arduino that implements the required functionality
2. a octave wrapper class that provides the Matlab interface and communication to the code.

Both parts are required to create a plugin.

The arduino toolkit provides a number of pre-created addons. These can be seen using the following command:

```
listArduinoLibraries
```

The command will display all known arduino libraries (addons as well as core libraries), however addons typically use a "foldername/classname" for this naming.

See also: listArduinoLibraries.

6.2 Creating an addon

An addon requires at minimum 3 things:

1. A addon package directory that will contain the addon files
2. A Matlab file within that directory that is a subclass of arduinoio.LibraryBase
3. A arduino source/header file that contains the arduino code to load, sub-classed for LibraryBase

So the addon directory structure at a minimum will be:

```
+arduinoioaddons  (dir) [somewhere in the octave load path]
  +MyAddons (dir)
    MyAddon1.m
    MyAddon1.h
```

6.2.1 Addon package directory

The addon architecture looks for plugins in the octave load path in a package directory called +arduinoioaddons

So this directory must be created somewhere within the paths that octave will check for functions.

In addition, the addon architecture expects plugins to be contained in a sub directory within the +arduinoioaddons folder. The subdirectory must begin with a '+'.

Multiple plugin .m files can be within the same sub directory.

6.2.2 Addon package .m file

The Matlab interface file within the addon directory provides the Matlab interface for the arduino code as well as provides information about the addon.

Class inheritance and required properties

The interface file must be a subclass pf arduinoio.LibraryBase and must contain some constant properties values that provide the information.

A minimum example of required is below:

```
classdef MyAddon1 < arduinoio.LibraryBase
  properties(Access = protected, Constant = true)
    LibraryName = 'MyAddons/MyAddon1';
    CppHeaderFile = fullfile(arduinoio.FilePath(mfilename('fullpath')), 'MyAddon1.h');
    CppClassName = 'MyAddon1';
  endproperties
  .
  .
  .
endclassdef
```

The following constant properties can be set within the addon:

LibraryName
: (Required) The name of the addon. My convention this is usually the directoryname / theclassname

CppHeaderFile
: (Required) The header file for the arduino code

CppSourceFile
: (Optional) The source file (if any) for the arduino code

CppClassName
: (Required) The classname used within the cppheaderfile for the arduino library

DependantLibraries
: (Optional) Any additional addons or cores that are needed for this library to be used

ArduinoLibraryHeaderFiles
: (Optional) Any additional header files that need to be included

Class constructor

The Matlab class constructor will be called from the addon function when creating a instance of the addon and should initialize at least two properties in inherited from arduinoio.LibraryBase:

1. Parent should be set to the first input argument (the arduino class)
2. Pins should be set to a list of pins that are used for the plugin

```
classdef MyAddon1 < arduinoio.LibraryBase
  .
  .
  methods
    function obj = MyAddon1(parentObj, varargin)
      obj.Parent = parentObj;
      # no pins being used
      obj.Pins = [];
      # send any command to the arduino during setup ?
```

```
    endfunction
    .
    .
  endmethods
endclassdef
```

Class functions

The class functions will usually communicate to the arduino and use the response for what is returned to the user.

By convention, the commands sent to the arduino are defined as constants in the class file but do not have to be.

```
classdef MyAddon1 < arduinoio.LibraryBase
  properties(Access = private, Constant = true)
    INIT_COMMAND = hex2dec('00');
    FUNC1_COMMAND = hex2dec('01');
  endproperties
  .
  .
  methods
    function obj = MyAddon1(parentObj, varargin)
      obj.Parent = parentObj;
      # no pins being used
      obj.Pins = [];
      # send any command to the arduino during setup ?
      sendCommand(obj.Parent, obj.LibraryName, obj.INIT_COMMAND, []);
    endfunction

    function retval = func1(obj)
       cmdID = obj.FUNC1_COMMAND;
       retval = sendCommand(obj.Parent, obj.LibraryName, cmdID, []);
    endfunction
    .
    .
  endmethods
endclassdef
```

NOTE

the sendCommand uses the objects parent for the arduino, the objects library name and the command id

See also: sendCommand.

6.2.3 Addon package header file

The header file should contain a class that matches the functionally and information of the matlab file and provides the ability to register the code on the arduino.

The following things should occur in the arduino class files:

1. The class name within the file must be the same as the one set in the .m file CppClassName property.

2. The libName variable must be the same as the LibraryName property.
3. The constructor should call registerLibrary
4. the commandHandler function to act on cmdID values that match the commands that will be sent from .m file and send data back using sendResponseMsg
5. on receiving unknown cmdID values, the commandHandler should use sendUnknownCmdIDMsg

An example, matching the previous .m file code is below:

```
#include "LibraryBase.h"

#define MYADDON1_INIT  0x00
#define MYADDON1_FUNC1 0x01

class MyAddon1 : public LibraryBase
{
  uint8_t cnt;
public:
  MyAddon1(OctaveArduinoClass& a)
  {
    libName = "MyAddons/MyAddon1";
    a.registerLibrary(this);
  }
  void commandHandler(uint8_t cmdID, uint8_t* data, uint8_t datasz)
  {
    switch (cmdID)
      {
      case MYADDON_INIT:
        {
          cnt = 0;
          sendResponseMsg(cmdID, 0,0);
          break;
        }
      case MYADDON_FUNC1:
        {
          // func 1 is just returning a uint8 count of number of times called
          cnt ++;
          sendResponseMsg(cmdID, &cnt, 1);
          break;
        }
      default:
        {
          // notify of invalid cmd
          sendUnknownCmdIDMsg();
        }
      }
  }
}
```

The body of functions can be in the CppSourceFile file is it is defined or within the header file as illustrated above.

6.2.4 Verify octave can see the addon

Use the listArduinoLibaries command to verify that the new addon appears in the list of known libraries.

If it does not, ensure that the +arduinoioaddons directory is within one of the octave class paths, and that the directory structure and inheritance requirements have been met.

6.3 Using addons

6.3.1 Programming the arduino with the addon

To use a addon, the code must be programmed onto the arduino.

Using the libraries command, when creating a arduino object, the arduino can be reprogrammed if the library does not already exist on the arduino.

```
ar = arduino([],[], 'libraries', 'MyAddons/MyAddon1', 'forcebuild', true)
```

The libraries property of the arduino object should list the libraries programmed on the arduino.

Alternatively, the library can be added using the libraries property and arduinosetup

See also: arduino, arduinosetup.

6.3.2 Creating a addon object

An object of the addon type can be created using the addon command.

```
ar = arduino([],[], 'libraries', 'MyAddons/MyAddon1', 'forcebuild', true)
obj = addon(ar, "MyAddons/MyAddon1");
```

7 Sensors Overview

There are two types of sensors available:

1. Matlab compatible(ish) sensors for envorinment and IMU.
2. Additional lightweight wrappers for some chips in a arduinosensor namespace.

7.1 Matlab Compatible Sensors

7.1.1 Overview

Matlab compatible functions are provided for a number of sensors, using a similar function naming as provided by the Maybal arduino package.

7.1.2 Available Sensors

The functions for each sensor is listed in the function reference, Section 9.15 [Sensors], page 74, and provides for:

bme280	BME280 temperature, pressure and humidity sensor
bno005	BNO055 acceleration, angular velocity, orientation and magnetic field sensor
lis3dh	LIS3DH acceleration sensor
lps22hb	LPS22HB temperature and pressure sensor
lsm6dso	LSM6DSO acceleration, angular velocity sensor
mpu6050	MPU-6050 acceleration, angular velocity sensor
SI7021	SI7021 temperature and humidity sensor

7.2 Lightweight Arduino Sensors

7.2.1 Overview

Arduino sensors are a collection of lightweight wrappers around other underlying protocols for providing specific sensor functionality.

For instance a DS1307 chip communicates using I2C protocol and so a DS1307 class exists that provides the conversion/commands in order to communicate to the chip.

Using the class, providing the functionality is very easy:

```
a = arduino()
rtc = arduinosensor.DS1307(a)
# get and display rtc time as a date string
datestr(rtc.clock)
```

It is lightweight compared to the addon functionality, as it only requires a wrapper class rather than add on code, however it is limited to then using available addon and core codes rather than creating new ones.

Currently the are only a small number of sensors available, however this will be built upon in future versions.

7.2.2 Available Sensors

The functions for each sensor is listed in the function reference, Section 9.12 [Arduino Sensors], page 66, and provides for:

DS1307 DS1307 RTC clock using i2c.

MPC3002 MPC3002 ADC using SPI

SI7021 SI7021 temperature and humidity sensor

GUVAS12SD
GUVAS12SD analog UV-B sensor

8 Examples

8.1 Blinking an LED

This example shows blinking the inbuilt LED on the Arduino board. Code is available by running:

```
edit examples/example_blink
```

Hardware setup

This example uses in the builtin LED, so requires only a connection of the Arduino board to computer for communication.

Create an Arduino object

```
ar = arduino ();
```

If you have more than one Arduino board connected, you may need to specify the port in order to connect to the correct device.

Query Device for pins connected to builtin LED

The pin connected to the Arduino UNO built in led if D13.

```
led_pin = "d13";
```

The connected pins can be queried programatically if desired.

```
pins = getLEDTerminals (ar);
```

Connected to a Arduino UNO would return a list pins containing only one item '13'.

The terminal number can be converted to a pin using getPinsFromTerminals:

```
led_pin = getPinsFromTerminals (ar, pins{1});
```

Turn the LED off

Write a 0 value to the pin to turn it off.

```
writeDigitalPin (ar, led_pin, 0);
```

Turn the LED on

Write a 1 value to the pin to turn it on

```
writeDigitalPin (ar, led_pin, 1);
```

Making the LED blink

Add a while loop with a pause between the changes in the pin state to blink.

```
while true
  writeDigitalPin (ar, led_pin, 0);
  pause (0.5)
  writeDigitalPin (ar, led_pin, 1);
  pause (0.5)
endwhile
```

8.2 Using I2C to communicate with an EEPROM

This example shows using I2C to communicate with a EEPROM chip. Code is available by running:

```
edit examples/example_i2c_eeprom
```

Hardware setup

Using an Arduino UNO, the board should be configured with the following connections between the board and a 24XX256 EEPROM chip:

A4	Connected to pin 5 of EEPROM
A5	Connected to pin 6 of EEPROM
5V	Connected to pin 8 of EEPROM
GND	Connected to pin 1,2,3,4 of EEPROM

Create an Arduino object

```
ar = arduino ();
```

If you have more than one Arduino board connected, you may need to specify the port in order to connect to the correct device.

Query I2C pins

Display the I2C terminals of the board:

```
getI2CTerminals(ar)
```

Scan the arduino for the connected device

```
scanI2Cbus(ar)
```

The devices listed should contain 0x50, the address of the EEPROM chip.

Create an I2C object to communicate to the EEPROM

```
eeprom = device (ar, "i2caddress", 0x50)
```

Write data to the EEPROM

The EEPROM expects the first byte to be the page number, the second the offset, followed by data, so to write 1 2 3 4, starting address 0 (page 0, offset 0):

```
write(eeprom, [0 0 1 2 3 4])
```

Reading from the EEPROM

Reading from the EEPROM requires first writing the address to read from, in this case, if we want to read the 3, 4, this would be page 0, offset 2:

```
write(eeprom, [0 2])
```

Next read the 2 bytes:

```
data = read(eeprom, 2)
```

8.3 Using SPI to communicate with a mcp3002 10 bit ADC

This example shows using SPI to communicate with an mcp3002 10 bit ADC. Code is available by running:

```
edit examples/example_spi_mcp3002
```

Hardware setup

Using an Arduino UNO, the board should be configured with the following connections between the board and a mcp3002 chip:

D10 Connected to pin 1 (CS) of MCP3002

D11 Connected to pin 5 (DI) of MCP3002

D12 Connected to pin 6 (DO) of MCP3002

D13 Connected to pin 7 (CLK) MCP3002

VCC Connected to pin 8 (VDD) MCP3002

GND Connected to pin 4 (VSS) MCP3002

Analog input
Connected from pin 2 of the MCP3002 to a LOW (< 5V) voltage to measure

Create an Arduino object

```
ar = arduino ();
```

If you have more than one Arduino board connected, you may need to specify the port in order to connect to the correct device.

Create an SPI object to communicate to the MCP3002

```
adc = device(ar, "spichipselectpin", "d10")
```

The d10 is the chip select pin connected from the Arduino to the MCP3002.

Read the ADC

The MCP3002 expects specific commands in order to read a channel.

For illustration for the command to read chan 0 in single ended mode:

```
command (bits) in MSB mode to device:
[START SGL ODN MSBF X X X X] [ X X X X X X X X ]
   1    1   0    1   1 1 1 1    1 1 1 1 1 1 1 1
      [chan 0 ] MSB
data back:
   X    X  X     X   X 0 D D    D D D D D D D D
```

D is a output data bit

X is a don't care what value is input/output

The first byte contains the command and start of the data read back, the second bytes is written to clock out the rest of the ADC data.

In hex, this corresponds to 0xDF 0xFF,

```
data = writeRead(adc, [hex2dec("DF") hex2dec("FF")])
```

Of the data returned, the last 10 bits is the actual data, so convert data to a 16 bit value:

```
val = uint16(data(1))*256 + uint16(data(2))
```

Then bitand it to remove the non value parts, to get the ADC value:

```
val = bitand (val, hex2dec('3FF'))
```

To make the value correspond to a voltage it needs to be scaled as 0 will be 0 Volts, 1023 will be 5 Volts.

```
volts = double(val) * 5.0 / 1023.0;
```

9 Function Reference

The functions currently available in the Arduino toolkit are described below;

9.1 General Functions

9.1.1 arduinosetup

retval = arduinosetup ()
retval = arduinosetup (*propertyname*, *propertyvalue*)

Open the arduino config / programming tool to program the arduino hardware for usage with the Octave arduino functions.

arduinosetup will create a temporary project using the arduino IDE and allow compiling and programming of the code to an arduino.

Inputs

propertyname, *propertyvalue* - A sequence of property name/value pairs can be given to set defaults while programming.

Currently the following properties can be set:

libraries
: The value should be the name of a library, or string array of libraries to program on the arduino board.

arduinobinary
: The value should be the name/path of the arduino IDE binary for programming. If not specified, the function will attempt to find the binary itself.

Outputs

retval - return 1 if arduino IDE returned without an error

See also: arduino, __arduino_binary__.

9.1.2 isarduino

retval = isarduino (*obj*)

Check if input value is an arduino object

Function is essentially just a call of `retval = isa(obj, "arduino");`

Inputs

obj - The object to check

Outputs

retval is true, if obj is an arduino object, false otherwise.

See also: arduino.

9.1.3 listArduinoLibraries

retval = listArduinoLibraries ()
retval = listArduinoLibraries (*libtypes*)

Retrieve list of all known arduino library modules that are available.

Inputs

libtypes - optional specifier for type of libraries to list.

Options are:

all	List core and addons
core	List core only libraries
addons	List addons only

When no libtypes is specified, all libraries are shown.

Outputs

retval is an cell array of string library names that are available for programming to the arduino.

See also: arduino, arduinosetup.

9.1.4 scanForArduinos

```
retval = scanForArduinos (maxCount)
retval = scanForArduinos ("debug")
retval = scanForArduinos (maxCount, type)
```

Scan system for programmed arduino boards.

scanForArduinos will scan the system for programmed arduino boards and return at most *maxCount* of them as a cell array in *retval*.

Inputs

maxCount - max number of arduino boards to detect. if *maxCount* is not specified, or is a less than 1, the function will return as many arduino boards as it can detect.

type - optional board type to match. If specified, the board type must match for the arduino to be added to the return list.

"*debug*" - if single input parameter is "debug", the scanForArduinos will display debug information as it scans all available ports for arduinos.

Outputs

retval structure cell array of matching detected arduino boards.

Each cell value of the cell array will contain a structure with values of:

port	the serial port the arduino is connected to
board	the board type of the arduino

See also: arduino.

9.2 Arduino Functions

9.2.1 @arduino/arduino

```
retval = arduino ()
retval = arduino (port)
retval = arduino (port, board)
retval = arduino (port, board[, [propname, propvalue]*)
retval = arduino (iaddress)
retval = arduino (ipaddress, board)
```

Create a arduino object with a connection to an arduino board.

Inputs

port - full path of serial port to connect to. For Linux, usually /dev/ttySXXX, for windows COMXX.

board - name of board to connect (default is 'uno').

propname, *propvalue* - property name and value pair for additional properties to pass to the creation of the arduino object.

Currently properties are ignored.

if the arduino function is called without parameters, it will scan for the first available arduino it can find and connect to it.

Outputs

retval - a successfully connected arduino object.

Properties

The arduino object has the following public properties:

name — name assigned to the arduino object

debug — true / false flag for whether debug in turned on

port (read only)
: the communications port the board is connected to.

board (read only)
: The name of the board type that the arduino connected to

libraries (read only)
: The libraries currently programmed onto the board

availablepins
: The pins available for use on the board

analogreference
: The analog voltage reference

See also: scanForArduinos, arduinosetup.

9.2.2 @arduino/checkI2CAddress

`retval = checkI2CAddress (ar, address)`
`retval = checkI2CAddress (ar, address, bus)`

Check that an address of given address responds on the I2C bus

Inputs

ar - arduino object connected to a arduino board.

address - I2C address number to check

bus - bus number to check for I2C device, when multiple buses are available. If the bus is not specified, it will default to 0.

Outputs

retval - boolean value of true if address responds on the I2C bus

Example

```
# create arduino connection.
ar = arduino();
# scan for devices on the I2C bus
checkI2CAddress (ar)
# output if a device using that address is attached
ans =
  1
```

See also: arduino, scanI2Cbus.

9.2.3 @arduino/configurePin

currmode = configurePin (*ar*, *pin*)
configurePin (*ar*, *pin*, *mode*)

Set/Get pin mode for a specified pin on arduino connection.

configurePin (*ar*, *pin*) will get the current mode of the specified pin.

configurePin (*ar*, *pin*, *mode*) will attempt set the pin to the specified mode if the mode is unset.

Inputs

ar - the arduino object of the connection to an arduino board.

pin - string name of the pin to set/get the mode of.

mode - string mode to set the pin to.

Outputs

mode - string current mode of the pin.

Valid modes can be:

- AnalogInput - Acquire analog signals from pin
- DigitalInput - Acquire digital signals from pin
- DigitalOutput - Generate digital signals from pin
- I2C - Specify a pin to use with I2C protocol
- Pullup - Specify pin to use a pullup switch
- PWM - Specify pin to use a pulse width modulator
- Servo - Specify pin to use a servo
- SPI - Specify a pin to use with SPI protocol
- Interrupt - Specify a pin to use for with interrupts
- Reserved - Specify a pin to be reserved
- Unset - Clears pin designation. The pin is no longer reserved and can be automatically set at the next operation.

See also: arduino.

9.2.4 @arduino/configurePinResource

currmode = configurePinResource (*ar*, *pin*)
configurePinResource (*ar*, *pin*, *owner*, *mode*)
configurePinResource (*ar*, *pin*, *owner*, *mode*, *force*)

Set/Get pin mode for a specified pin on arduino connection.

configurePinResource (*ar*, *pin*) will get the current mode of the specified pin.

configurePinResource (*ar*, *pin*, *owner*, *mode*) will attempt set the pin to the specified mode and owner.

If the pin is already owned by another owner, the configure will fail unless the force option is used. If the mode is already set, configure will fail unless force is used.

Inputs

ar - the arduino object of the connection to an arduino board.

pin - string name of the pin to set/get the mode of.

mode - string mode to set the pin to.

owner - string name to use as the pin owner.

force - boolean to force mode change. If not set, it will be false.

Outputs

currmode - current string mode of the pin.

Valid modes can be:

- AnalogInput - Acquire analog signals from pin
- DigitalInput - Acquire digital signals from pin
- DigitalOutput - Generate digital signals from pin
- I2C - Specify a pin to use with I2C protocol
- Pullup - Specify pin to use a pullup switch
- PWM - Specify pin to use a pulse width modulator
- Servo - Specify pin to use a servo
- SPI - Specify a pin to use with SPI protocol
- Interrupt - Specify a pin to use with interrupts
- Reserved - Pin marked reserved, but not for of any particular mode
- Unset - Clears pin designation. The pin is no longer reserved and can be automatically set at the next operation.

See also: arduino, configurePin.

9.2.5 @arduino/decrementResourceCount

count = decrementResourceCount (*ar*, *resource*)

Decrement the count of a named resource by 1 and return the new count.

Inputs

ar - connected arduino object

resource - name of resource to decrement count.

Outputs

count = count of uses registered to resource.

See also: getResourceCount. incrementResourceCount.

9.2.6 @arduino/delete

`delete (dev)`
Free resources of an arduino object.

Inputs

dev - object to free

See also: arduino.

9.2.7 @arduino/display

`display (ar)`
Display the arduino object in a verbose way, showing the board and available pins.

Inputs

ar - the arduino object.

If the arduino object has debug mode set, additional information will be displayed.

See also: arduino.

9.2.8 @arduino/getEndian

`mcu = getEndian (ar)`
Get the endian used by the connected arduino.

Inputs

ar - arduino object connected to a arduino board.

Outputs

endian - string representing the endian used by the arduino board.

'L' means little endian, 'B' means big endian

See also: arduino, getMCU.

9.2.9 @arduino/getI2CTerminals

`pinlist = getI2CTerminals (ar)`
`pinlist = getI2CTerminals (ar, bus)`
Get a cell list of pin Ids available are used for I2C mode.

Inputs

ar - the arduino object.

bus - optional bus number 0 or 1 for boards that support more than 1 bus.

Outputs

pinlist - cell list of pin numbers available for I2C use.

See also: arduino.

9.2.10 @arduino/getInterruptTerminals

`pinlist = getInterruptTerminals (ar)`
Get a cell list of pin Ids available have interrupt functionality

Inputs

ar - the arduino object.

Outputs

pinlist - cell list of pin numbers available for interrupt use.

See also: arduino.

9.2.11 @arduino/getLEDTerminals

pinlist = `getLEDTerminals (`*ar*`)`

Get a cell list of pin Ids available are connected natively to LEDs.

Inputs

ar - the arduino object.

Outputs

pinlist - cell list of pin numbers available for LED use.

See also: arduino.

9.2.12 @arduino/getMCU

mcu = `getMCU (`*ar*`)`

Get the MCU used by the connected arduino.

Inputs

ar - arduino object connected to a arduino board.

Outputs

mcu - string representing the mcu used by the arduino board.

See also: arduino.

9.2.13 @arduino/getPWMTerminals

pinlist = `getPWMTerminals (`*ar*`)`

Get a cell list of pin Ids available for PWM use.

Inputs

ar - the arduino object.

Outputs

pinlist - cell list of pin numbers available for PWM use.

See also: arduino.

9.2.14 @arduino/getPinAlias

ouy = `getPinAlias (`*ar*, *pin*`)`

Get the pin actual pin name from a pin alias.

For example, the arduino Leonardo, pin `"D4"` is also `"A6"`.

Inputs

ar - the connected arduino object.

pin - a pin name.

Outputs

out - alias pin name, or same as *pin* if the pin doesnt have any alias names.

See also: arduino, configurePinResource, getResourceOwner.

9.2.15 @arduino/getPinInfo

`pininfo = getPinInfo (ar, pin)`
`pininfoarray = getPinInfo (ar, pinarray)`

Get the pin information from the input pins values.

getPinInfo (*ar*, *pin*) will get information for a single pin.

getPinInfo (*ar*, *pinarray*) will get a cell array of pin information

Inputs

ar - the connected arduino object.

pin - a pin number or pin name.

pinarray - the array of pin numbers or names

The pininfo struct contains the following fields:

terminal	Terminal number of the pin
name	String name of the pin
owner	Current item owner of the pin
mode	Current configured mode for the pin

Outputs

pininfo - struct on pin information.

pininfolist - cell array of pin info

See also: arduino, configurePinResource, getResourceOwner.

9.2.16 @arduino/getPinsFromTerminals

`pinnames = getPinsFromTerminals (ar, terminals)`

Get the pin names from the input terminal values.

Inputs

ar - the connected arduino object.

terminals - the numeric pin number, or array of pin numbers to get pin names.

Outputs

pinnames - the string names of each input pin. If terminals was a single value, the return will be a single string, otherwise it will return a cell array of each pin name.

See also: arduino, getTerminalsFromPins.

9.2.17 @arduino/getResourceCount

`count = getResourceCount (ar, resource)`

Get the count of uses of a given resource.

Inputs

ar - connected arduino object

resource - name of resource to get count for.

Outputs

count = count of uses registered to resource.

See also: incrementResourceCount. decrementResourceCount.

9.2.18 @arduino/getResourceOwner

`owner = getResourceOwner (ar, terminal)`

Get the owner of pin allocated previously by configurePinResource.

Inputs

ar - connected arduino object

terminal - terminal number to get owner of.

Outputs

owner = owner of the terminal pin, or `""` if not owned.

See also: configurePinResource.

9.2.19 @arduino/getSPITerminals

`pinlist = getSPITerminals (ar)`

Get a cell list of pin Ids available for SPI mode.

Inputs

ar - the arduino object.

Outputs

pinlist - cell list of pin numbers available for SPI use.

See also: arduino.

9.2.20 @arduino/getServoTerminals

`pinlist = getServoTerminals (ar)`

Get a cell list of pin Ids available for servo use.

Inputs

ar - the arduino object.

Outputs

pinlist - cell list of pin numbers available for servo use.

See also: arduino, getPWMTerminals.

9.2.21 @arduino/getSharedResourceProperty

`count = getSharedResourceProperty (ar, resource, property)`

Get the value of a property from a given resource.

Inputs

ar - connected arduino object

resource - name of resource to get property for.

property - name of property from the resource.

Outputs

propvalue - value of the property

See also: getResourceCount, setSharedResourceProperty.

9.2.22 @arduino/getTerminalMode

mode = `getTerminalMode` (*ar*, *terminal*)
Get the mode of a pin allocated previously by configurePinResource.

Inputs

ar - connected arduino object

terminal - terminal number to get owner of.

Outputs

mode - mode of the terminal pin, or `"not_set"` if not owned.

See also: configurePinResource, getResourceOwner.

9.2.23 @arduino/getTerminalsFromPins

pinnums = `getTerminalsFromPins` (*ar*, *pins*)
Get the terminal number for each pin.

Inputs

ar - connected arduino object

pins - single pin name or cell or vector array of pin names.

Outputs

pinnums - pin number of each named pin. If the input was a single string, returns a number. if the input pins was a vector or cell array, return a cell array of pin numbers corresponding to each input pin name.

See also: arduino, getPinsFromTerminals.

9.2.24 @arduino/incrementResourceCount

count = `incrementResourceCount` (*ar*, *resource*)
Increment the count value of a named resource by 1 and return the new count

Inputs

ar - connected arduino object

resource - name of resource to increment count.

Outputs

count = count of uses registered to resource.

See also: getResourceCount. decrementResourceCount.

9.2.25 @arduino/isTerminalAnalog

ret = `isTerminalAnalog` (*obj*, *terminal*)
Return true if pin is capable of analog input

Inputs

ar - the connected arduino object

terminal is a terminal number to check

Outputs

ret return 1 if terminal is a analog pin, 0 otherwise

9.2.26 @arduino/isTerminalDigital

`ret = isTerminalDigital(obj, terminal)`

Return true if pin is capable of digital functions

Inputs

ar - the connected arduino object

terminal is a terminal number to check

Outputs

ret return 1 if terminal is a digital pin, 0 otherwise

9.2.27 @arduino/playTone

`playTone (ar, pin, freq, duration)`

Play a tone of a given frequency on a specified pin.

Inputs

ar - connected arduino object

pin - digital pin to play tone on

freq - frequency in hertz to play between 0 and 32767Hz.

duration duration in seconds to play tone between 0 and 30 seconds

If duration is 0 or not specified, tone will continue to play until next tone is commanded. If frequency is 0, tone will stop playing

NOTE: use of playTone can interfere with PWM output.

9.2.28 @arduino/readAnalogPin

`value = readAnalogPin (ar, pin)`

Read analog voltage of *pin*.

Inputs

ar - connected arduino object.

pin - string name of the pin to read.

Outputs

value - analog value of the pin

Example

```
ar = arduino ();
readAnalogPin(ar, "A4");
ans =
```

```
    87
```

See also: arduino, readVoltage.

9.2.29 @arduino/readDigitalPin

value = readDigitalPin (*obj*, *pin*)
Read digital value from a digital I/O pin.

Inputs

ar - connected arduino object.

pin - string name of the pin to read.

Outputs

value - the logical value (0, 1, true false) of the current pin state.

Example

```
a = arduino ();
pinvalue = readDigitalPin (a, 'D5');
```

See also: arduino, writeDigitalPin.

9.2.30 @arduino/readVoltage

voltage = readVoltage (*ar*, *pin*)
Read analog voltage of a pin.

Inputs

ar - connected arduino.

pin - pin name or number to query for voltage

Outputs

voltage - scaled pin value as a voltage

Example

```
ar = arduino ();
readVoltage(ar, "A4");
ans =
    1.401
```

See also: arduino, readAnalogPin.

9.2.31 @arduino/reset

reset (*ar*)
Send reset command to arduino hardware to force a hardware reset.

Inputs

ar - connected arduino object.

See also: arduino.

9.2.32 @arduino/sendCommand

outdata, outsize = sendCommand (*ar*, *libname*, *commandid*)
outdata, outsize = sendCommand (*ar*, *libname*, *commandid*, *data*)
outdata, outsize = sendCommand (*ar*, *libname*, *commandid*, *data*, *timeout*)

Send a command with option data to the connected arduino, waiting up to a specified number of seconds for a response.

Inputs

ar - connected arduino object.

libname - library sending the command. The name should match a programmed library of the arduino, or an error will be displayed.

commandid - integer value for the command being sent to the arduino.

data - optional data sent with the command.

timeout - optional timeout to wait for data

Outputs

outdata - data returned back from the arduino in response to command

outsize - size of data received

If the arduino fails to respond with a valid reply, sendCommand will error.

See also: arduino.

9.2.33 @arduino/setSharedResourceProperty

setSharedResourceProperty (*ar*, *resource*, *propname*, *propvalue*)
setSharedResourceProperty (*ar*, *resource*, *propname*, *propvalue*, ...)

Set property values for a given resource.

Inputs

ar - connected arduino object

resource - name of resource to get property for.

propname - name of property from the resource.

propvalue - value of property from the resource.

Multiple *propname*, *propvalue* pairs can be given.

Outputs

None

Example

```
ar = arduino();
setSharedResourceProperty(ar, "myresource", "myproperty", [1 2 3])
```

See also: getSharedResourceProperty.

9.2.34 @arduino/uptime

sec = uptime (*ar*)
Get the number of seconds the arduino board has been running concurrently.

Inputs

ar - the arduino object of the connection to an arduino board.

Outputs

sec - the number seconds the board has been running. Note that the count will wrap around after approximately 50 days.

See also: arduino.

9.2.35 @arduino/validatePin

validatePin (*ar*, *pin*, *type*)
Validate that the mode is allowed for specified pin

If the mode is not valid, and error will be thrown.

Inputs

ar - connected arduino object

pin - name of pin to query mode validity of

mode - mode to query

Known modes are:

- 'I2C'
- 'SPI'
- 'PWM'
- 'Servo'
- 'analog'
- 'digital'

See also: arduino, configurePin.

9.2.36 @arduino/version

ver = version (*ar*)
Get version of library code installed on arduino board

Inputs

ar - the arduino object of the connection to an arduino board.

Outputs

ver - version string in format of X.Y.Z.

See also: arduino.

9.2.37 @arduino/writeDigitalPin

writeDigitalPin (*ar*, *pin*, *value*)
Write digital value to a digital I/O pin.

Inputs

ar - connected arduino object.

pin - string name of the pin to write to.

value - the logical value (0, 1, true false) to write to the pin.

If pin was unconfigured before using, pin is set into digital mode.

Example

```
a = arduino();
writeDigitalPin(a,'D5',1);
```

See also: arduino, readDigitalPin.

9.2.38 @arduino/writePWMDutyCycle

`writePWMDutyCyle` (*ar*, *pin*, *value*)

Set pin to output a square wave with a specified duty cycle.

Inputs

ar - connected arduino object

pin - pin to write to.

value - duty cycle value where 0 = off, 0.5 = 50% on, 1 = always on.

Example

```
a = arduino();
writePWMDutyCycle(a,'D5',0.5);
```

See also: arduino, writePWMVoltage.

9.2.39 @arduino/writePWMVoltage

`writePWMVoltage` (*ar*, *pin*, *voltage*)

Emulate an approximate voltage out of a pin using PWM.

Inputs

ar - connected arduino object

pin - pin to write to.

voltage - voltage to emulate with PWM, between 0 - 5.0

Example

```
a = arduino();
writePWMVoltage(a,'D5',1.0);
```

See also: arduino, writePWMDutyCycle.

9.3 Arduino I2C Functions

9.3.1 @device/delete

`delete (`*`dev`*`)`
Free resources of a device object.

Inputs

dev - object to free

See also: device.

9.3.2 @device/display

`display (`*`dev`*`)`
Display device object.

Inputs

dev - device object to display

See also: device.

9.3.3 @device/read

`data` `= read (`*`dev`*`, `*`numbytes`*`)`
`data` `= read (`*`dev`*`, `*`numbytes`*`, `*`precision`*`)`
Read a specified number of bytes from a i2c or serial device object using optional precision for bytesize.

Inputs

dev - connected i2c or serial device opened using device

numbytes - number of bytes to read.

precision - Optional precision for the output data read data. Currently known precision values are uint8 (default), int8, uint16, int16

Outputs

data - data read from the device

See also: arduino, device.

9.3.4 @device/readRegister

`data` `= readRegister (`*`dev`*`, `*`reg`*`, `*`numbytes`*`)`
`data` `= readRegister (`*`dev`*`, `*`reg`*`, `*`numbytes`*`, `*`precision`*`)`
Read a specified number of bytes from a register of an i2cdev object using optional precision for bytesize.

Inputs

dev - connected i2c device opened using device

reg - registry value number

numbytes - number of bytes to read.

precision - Optional precision for the output data read data. Currently known precision values are uint8 (default), int8, uint16, int16

Output

data - data read from device.

See also: arduino, device.

9.3.5 @device/subsref

`val = subsref (dev, sub)`
subref for device

See also: device.

9.3.6 @device/write

`write (dev, datain)`
`write (dev, datain, precision)`
Write data to a I2C or serial device object using optional precision for the data byte used for the data.

Inputs

dev - connected i2c or serial device opened using device

datain - data to write to device. Datasize should not exceed the constraints of the data type specified for the precision.

precision - Optional precision for the input write data. Currently known precision values are uint8 (default), int8, uint16, int16

See also: arduino, device, read.

9.3.7 @device/writeRegister

`writeRegister (dev, reg, datain)`
`writeRegister (dev, dev, datain, precision)`
Write data to i2c device object at a given registry position using optional precision for the data byte used for the data.

Inputs

dev - connected i2c device opened using device

reg - registry position to write to.

datain - data to write to device. Datasize should not exceed the constraints of the data type specified for the precision.

precision - Optional precision for the input write data. Currently known precision values are uint8 (default), int8, uint16, int16

See also: arduino, device, read.

9.3.8 @i2cdev/delete

`delete (dev)`
Free resources of a i2cdev object.

Inputs

dev - object to free

See also: i2cdev.

9.3.9 @i2cdev/display

`display (dev)`
Display i2cdev object.

Inputs

dev - i2cdev object

See also: i2cdev.

9.3.10 @i2cdev/i2cdev

`dev = i2cdev (ar, address)`
`dev = i2cdev (ar, address, propname, propvalue)`
`i2cdev` is depreciated and will be removed in a future version. Use `device` instead.

Create an i2cdev object to communicate to the i2c port on a connected arduino.

Inputs

ar - connected arduino object

address - address to use for device on I2C bus.

propname, *propvalue* - property name/value pair for values to pass to devices.

Currently known properties:

bus — bus number (when arduino board supports multiple I2C buses) with value of 0 or 1.

Outputs

dev - new created i2cdev object.

Properties

The i2cdev object has the following public properties:

parent — The parent (arduino) for this device

pins — pins used by this object

bus — bus used for created object

address — I2C address set for object

See also: arduino.

9.3.11 @i2cdev/read

`data = read (dev, numbytes)`
`data = read (dev, numbytes, precision)`
Read a specified number of bytes from a i2cdev object using optional precision for bytesize.

Inputs

dev - connected i2c device opened using i2cdev

numbytes - number of bytes to read.

precision - Optional precision for the output data read data. Currently known precision values are uint8 (default), int8, uint16, int16

Outputs

data - data read from i2cdevice

See also: arduino, i2cdev.

9.3.12 @i2cdev/readRegister

`data = readRegister (dev, reg, numbytes)`
`data = readRegister (dev, reg, numbytes, precision)`
Read a specified number of bytes from a register of an i2cdev object using optional precision for bytesize.

Inputs

dev - connected i2c device opened using i2cdev

reg - registry value number

numbytes - number of bytes to read.

precision - Optional precision for the output data read data. Currently known precision values are uint8 (default), int8, uint16, int16

Output

data - data read from device.

See also: arduino, i2cdev.

9.3.13 @i2cdev/subsref

`val =  subsref (dev, sub)`
subref for i2cdev

See also: i2cdev.

9.3.14 @i2cdev/write

`write (dev, datain)`
`write (dev, datain, precision)`
Write data to a i2cdev object using optional precision for the data byte used for the data.

Inputs

dev - connected i2c device opened using i2cdev

datain - data to write to device. Datasize should not exceed the constraints of the data type specified for the precision.

precision - Optional precision for the input write data. Currently known precision values are uint8 (default), int8, uint16, int16

See also: arduino, i2cdev, read.

9.3.15 @i2cdev/writeRegister

`writeRegister (dev, reg, datain)`
`writeRegister (dev, dev, datain, precision)`
Write data to i2cdev object at a given registry position using optional precision for the data byte used for the data.

Inputs

dev - connected i2c device opened using i2cdev

reg - registry position to write to.

datain - data to write to device. Datasize should not exceed the constraints of the data type specified for the precision.

precision - Optional precision for the input write data. Currently known precision values are uint8 (default), int8, uint16, int16

See also: arduino, i2cdev, read.

9.3.16 scanI2Cbus

`retval = scanI2Cbus (ar)`
`retval = scanI2Cbus (ar, bus)`
Scan arduino for devices on the I2C bus.

Inputs

ar - arduino object connected to a arduino board.

bus - bus number to scan I2C devices, when multiple buses are available. If the bus is not specified, it will default to 0.

Outputs

retval - cell array of addresses as strings in format of "0xXX".

Example

```
# create arduino connection.
ar = arduino();
# scan for devices on the I2C bus
scanI2Cbus (ar)
# output is each detected i2c address as a string
ans =
{
 [1,1] = 0x50
}
```

See also: arduino, i2cdev, checkI2CAddress.

9.4 Arduino Rotary Encoder Functions

9.4.1 @rotaryEncoder/delete

`delete (dev)`
Free resources of a encoder object.

Inputs

dev - object to free

See also: rotartEncoder.

9.4.2 @rotaryEncoder/display

retval = display (*obj*)
Display the rotary encoder object in a verbose way,

Inputs

obj - the arduino rotary encoder object created with rotaryEncoder

See also: rotaryEncoder.

9.4.3 @rotaryEncoder/readCount

[*count*, *time*] = readCount (*obj*)
[*count*, *time*] = readCount (*obj*, *name*, *value*)
read count value from the rotary encoder.
subsubheading Inputs *obj* - rotary encoder object created with rotaryEncoder call.
name, *value* - optional name,value pairs
Valid option name pairs currently are:

reset	Reset the count after reading (if true)

Outputs

count - returned count read from the encoder.
time - seconds since arduino started

See also: rotaryEncoder, resetCount.

9.4.4 @rotaryEncoder/readSpeed

speed = readSpeed (*obj*)
read rotational speed from the rotary encoder.

Inputs

obj - rotary encoder object created with rotaryEncoder call.

Outputs

speed - returned speed in revolutions per minute read from the encoder.

See also: rotaryEncoder, resetCount.

9.4.5 @rotaryEncoder/resetCount

reset (*obj*)
reset (*obj*, *cnt*)
reset the rotary encoder count values

Inputs

obj - the rotaryEncoder object
cnt - optional count value to reset to

See also: rotaryEncoder, readCount.

9.4.6 @rotaryEncoder/rotaryEncoder

obj = rotaryEncoder (*ar*, *chanApin*, *chanBpin*)
obj = rotaryEncoder (*ar*, *chanApin*, *chanBpin*, *ppr*)
Create a rotaryEncoder object controlled by the input pins.

Inputs

ar - connected arduino object.

chanApin - pin used for channel A

chanBpin - pin used for channel B

ppr - count of encoder pulsed required for a full revolution of the encoder.

Outputs

obj - created rotary encoder object

Example

```
a = arduino ();
enc = rotaryEncoder(a, "d2", "d3", 180);
```

Properties

The rotaryEncoder object has the following public properties:

parent	The parent (arduino) for this device
pins	pins used by this object
ppr	Number of pulses used per rotation

See also: arduino.

9.4.7 @rotaryEncoder/subsref

`val = subsref (dev, sub)`
subref for rotaryEncoder

See also: rotaryEncoder.

9.5 Arduino Servo Functions

9.5.1 @servo/delete

`delete (dev)`
Free resources of a servo object.

Inputs

dev - object to free

See also: servo.

9.5.2 @servo/display

`display (dev)`
Display servo object.

Inputs

dev - device to display

See also: servo.

9.5.3 @servo/readPosition

position = readPosition (*servo*)

Read the position of a servo

Inputs

servo - servo object created from arduino.servo.

Outputs

position - value between 0 .. 1 for the current servo position, where 0 is the servo min position, 1 is the servo maximum position.

See also: servo, writePosition.

9.5.4 @servo/servo

obj = servo (*arduinoobj*, *pin*)
obj = servo (*arduinoobj*, *pin*, *propertyname*, *propertyvalue*)

Create a servo object using a specified pin on a arduino board.

Inputs

obj - servo object

arduinoobj - connected arduino object

propertyname, *propertyvalue* - name value pairs for properties to pass to the created servo object.

Current properties are:

minpulseduration
: min PWM pulse value in seconds.

maxpulseduration
: max PWM pulse value in seconds.

Outputs

obj - created servo object.

Example

```
# create arduino connection
ar = arduino();
# create hobby servo (1 - 2 ms pulse range)
servo = servo(ar, "d9", "minpulseduration", 1.0e-3, "maxpulseduration", 2e-
3);
# center the servo
writePosition(servo, 0.5);
```

Properties

The servo object has the following public properties:

parent
: The parent (arduino) for this device

pins
: pins used by this object

minpulseduration
: minpulseduration set for object

maxpulseduration
maxpulseduration set for object

See also: arduino, readPosition, writePosition.

9.5.5 @servo/subsref

val = `subsref` (*dev*, *sub*)
subref for servo

See also: servo.

9.5.6 @servo/writePosition

`writePosition` (*servo*, *position*)
Write the position to a servo.

Inputs

servo - servo object created from arduino.servo.

position - value between 0 .. 1 for the current servo position, where 0 is the servo min position, 1 is the servo maximum position.

See also: servo, readPosition.

9.6 Arduino Shiftregister Functions

9.6.1 @shiftRegister/delete

`delete` (*dev*)
Free resources of a shiftRegister object.

Inputs

dev - object to free

See also: shiftRegister.

9.6.2 @shiftRegister/display

retval = `display` (*register*)
Display the register object in a verbose way,

Inputs

register - the arduino register object created with shiftRegister.

See also: shiftRegister.

9.6.3 @shiftRegister/read

retval = `read` (*register*)
retval = `read` (*register*, *precision*)
read a value from the shift register.

Inputs

register - shift register created from shiftRegister call.

precision - optional precision of the data, where precision can be a number in a multiple of 8 (ie: 8,16,32) or can be a named integer type: 8 of 'uint8', 'uint16', 'uint32'. The default precision is 8.

Outputs

retval - returned data read from the register.

See also: shiftRegister, write.

9.6.4 @shiftRegister/reset

`reset (`*`register`*`)`
clear the shift register value.

Inputs

register - shift register created from shiftRegister call.

See also: shiftRegister, read, write.

9.6.5 @shiftRegister/shiftRegister

`register` `= shiftRegister (`*`ar`*`, `*`shifttype`*`, `*`dataPin`*`, `*`clockPin`*` ...)`
`register` `= shiftRegister (`*`ar`*`,'74hc164', `*`dataPin`*`, `*`clockPin`*`, `*`resetPin`*`)`
`register` `= shiftRegister (`*`ar`*`,'74hc165', `*`dataPin`*`, `*`clockPin`*`, `*`loadPin`*`, `*`clockEnablePin`*`)`
`register` `= shiftRegister(`*`ar`*`,'74hc595', `*`dataPin`*`, `*`clockPin`*`, `*`latchPin`*` , `*`resetPin`*`)`
Create shift register of a given type, controlled by the input pins.

Inputs

Common function parameter definition:

ar - connected arduino object.

shifttype - string name of the shift register type.

dataPin - pin used for data in/out of the device.

clockPin - pin used for clocking data on the shiftRegister.

Other variables are dependent on the shift register type:

'74hc164'	Additional inputs: *resetPin* - optional pin for resetting the shift register.
'74hc165'	Additional inputs: *loadPin* - load pin to the shift register. *clockEnablePin* - clock enable pin.
'74hc595'	Additional inputs: *latchPin* - latching data to the shift register. *resetPin* - optional pin for resetting the shift register.

Outputs

register - register object

Properties

The shiftRegister object has the following public properties:

parent	The parent (arduino) for this device
pins	pins used by this object
model	model set for object

See also: arduino.

9.6.6 @shiftRegister/subsref

val = subsref (*dev*, *sub*)
subref for shiftRegister

See also: shiftRegister.

9.6.7 @shiftRegister/write

write (*register*, *dataIn*)
write (*register*, *dataIn*, *precision*)
Write a value to the shift register.

Inputs

register - shift register created from shiftRegister call.

dataIn - data to clock into the shiftRegister.

precision - optional precision of the data, where precision can be a number in a multiple of 8 (ie: 8,16,32) or can be a named integer type of 'uint8', 'uint16', 'uint32'. The default precision is 8.

See also: shiftRegister, read.

9.7 Arduino SPI Functions

9.7.1 @device/delete

delete (*dev*)
Free resources of a device object.

Inputs

dev - object to free

See also: device.

9.7.2 @device/display

display (*dev*)
Display device object.

Inputs

dev - device object to display

See also: device.

9.7.3 @device/subsref

val = subsref (*dev*, *sub*)
subref for device

See also: device.

9.7.4 @device/writeRead

dataOut = readWrite (*spi*, *dataIn*)
Write uint8 data to spi device and return back clocked out response data of same size.

Inputs

spi - connected spi device on arduino

dataIn - uint8 sized data to send to spi device framed between SS frame.

Outputs

dataOut - uint8 data clocked out during send to dataIn.

See also: arduino, device.

9.7.5 @spidev/delete

`delete (dev)`
Free resources of a spidev object.

Inputs

dev - spidev object to free

See also: spidev.

9.7.6 @spidev/display

`display (dev)`
Display spidev object.

Inputs

dev - spidev object to display

See also: spidev.

9.7.7 @spidev/spidev

`dev = spidev (ar, cspin)`
`dev = spidev (ar, cspin, propname, propvalue)`
`spidev` is depreciated and will be removed in a future version. Use `device` instead.

Create an spidev object to communicate to the SPI port on a connected arduino.

Inputs

ar - connected arduino object

cspin - chip select pin for attached spi device.

propname, *propvalue* - property name/value pair for values to pass to devices.

Currently known properties:

bitrate	bit rate speed in Mbs
bitorder	'msbfirst' or 'lsbfirst'
mode	SPI mode 0 - 3.

Outputs

dev - created spidev object

Properties

The spidev object has the following public properties:

parent — The parent (arduino) for this device

pins — pins used by this object

mode — mode used for created object

bitrate — Bitrate set for object

bitorder — Bitorder set for object

chipselectpin — Pin used for chipselect

See also: arduino, readWrite.

9.7.8 @spidev/subsref

`val = subsref (dev, sub)`
subref for spidev

See also: spidev.

9.7.9 @spidev/writeRead

`dataOut = readWrite (spi, dataIn)`
Write uint8 data to spi device and return back clocked out response data of same size.

Inputs

spi - connected spi device on arduino

dataIn - uint8 sized data to send to spi device framed between SS frame.

Outputs

dataOut - uint8 data clocked out during send to dataIn.

See also: arduino, spidev.

9.8 Arduino Serial Functions

9.8.1 @device/delete

`delete (dev)`
Free resources of a device object.

Inputs

dev - object to free

See also: device.

9.8.2 @device/device

`dev = device (ar, 'I2CAddress', address)`
`dev = device (ar, 'SPIChipSelectPin', pin)`
`dev = device (ar, 'Serial', serialid)`
`dev = device (..., propname, propvalue)`
Create an i2c, spi or serial object to communicate on a connected arduino.

Inputs

ar - connected arduino object

propname, *propvalue* - property name/value pair for values to pass to devices.

A property of 'i2caddress', 'spichipselectpin' or 'serial' must be specified to denote the device type to create.

i2caddress - address to use for device on I2C bus.

pin - pin to use for device SPI chip select.

serialid - Serial port id to use

Additional properties can also be specified for the device object

Currently known input I2C properties values:

bus	bus number (when arduino board supports multiple I2C buses) with value of 0 or 1.
noprobe	Do not probe the existence of device on creation if set to 1 (default 0)
bitrate	bit rate speed in Mbs - default 100000

Currently known input SPI properties values:

bitrate	bit rate speed in Mbs
bitorder	'msbfirst' or 'lsbfirst'
spimode	SPI mode 0 - 3.

Currently known input Serial properties values:

baudrate	baudrate value (default 9600)
databits	number of databits (5,6,7,8) (default 8)
stopbits	number of stopbits (1,2) (default 1)
parity	parity of device ('odd','even','none') (default 'none')

Outputs

dev - new created device object.

Properties

The object has the following public properties:

parent	The parent (arduino) for this device
interface	The interface type for this device ("SPI" or "I2C" or "Serial")

In addition, depending on type, the object will have these properties:

I2C Properties

The object has the following public properties:

bus	bus used for created object
i2caddress	I2C address set for object
sclpin	the SCL pin of the device
sdapin	the SDA pin of the device
bitrate	bit rate for the i2c clock

SPI Properties

The object has the following public properties:

spimode mode used for created object

bitrate Bitrate set for object

bitorder Bitorder set for object

spichipselectpin
Pin used for chipselect

mosipin Pin used for mosi

misopin Pin used for miso

sckpin Pin used for sckpin

Serial Properties

The object has the following public properties:

id serial port id

baudrate baudrate

databits number of databits (5,6,7,8)

stopbits number of stopbits (1,2)

parity parity of device ('odd','even','none')

See also: arduino, i2cdev, spidev.

9.8.3 @device/display

`display (`*`dev`*`)`
Display device object.

Inputs

dev - device object to display

See also: device.

9.8.4 @device/flush

`data` `= flush (`*`dev`*`)`
`data` `= flush (`*`dev`*`, "`*`input`*`")`
`data` `= flush (`*`dev`*`, "`*`output`*`")`
Flush the serial port buffers

Inputs

dev - connected serial device opened using device

If an additional parameter is provided of "input" or "output", then only the input or output buffer will be flushed

Outputs

None

See also: arduino, device, read.

9.8.5 @device/read

data = read (*dev*, *numbytes*)
data = read (*dev*, *numbytes*, *precision*)
Read a specified number of bytes from a i2c or serial device object using optional precision for bytesize.

Inputs

dev - connected i2c or serial device opened using device

numbytes - number of bytes to read.

precision - Optional precision for the output data read data. Currently known precision values are uint8 (default), int8, uint16, int16

Outputs

data - data read from the device

See also: arduino, device.

9.8.6 @device/subsref

val = subsref (*dev*, *sub*)
subref for device

See also: device.

9.8.7 @device/write

write (*dev*, *datain*)
write (*dev*, *datain*, *precision*)
Write data to a I2C or serial device object using optional precision for the data byte used for the data.

Inputs

dev - connected i2c or serial device opened using device

datain - data to write to device. Datasize should not exceed the constraints of the data type specified for the precision.

precision - Optional precision for the input write data. Currently known precision values are uint8 (default), int8, uint16, int16

See also: arduino, device, read.

9.9 Arduino Device Functions

9.9.1 @device/delete

delete (*dev*)
Free resources of a device object.

Inputs

dev - object to free

See also: device.

9.9.2 @device/device

dev = device (*ar*, *'I2CAddress'*, *address*)
dev = device (*ar*, *'SPIChipSelectPin'*, *pin*)
dev = device (*ar*, *'Serial'*, *serialid*)
dev = device (*...*, *propname*, *propvalue*)

Create an i2c, spi or serial object to communicate on a connected arduino.

Inputs

ar - connected arduino object

propname, *propvalue* - property name/value pair for values to pass to devices.

A property of 'i2caddress', 'spichipselectpin' or 'serial' must be specified to denote the device type to create.

i2caddress - address to use for device on I2C bus.

pin - pin to use for device SPI chip select.

serialid - Serial port id to use

Additional properties can also be specified for the device object

Currently known input I2C properties values:

bus	bus number (when arduino board supports multiple I2C buses) with value of 0 or 1.
noprobe	Do not probe the existence of device on creation if set to 1 (default 0)
bitrate	bit rate speed in Mbs - default 100000

Currently known input SPI properties values:

bitrate	bit rate speed in Mbs
bitorder	'msbfirst' or 'lsbfirst'
spimode	SPI mode 0 - 3.

Currently known input Serial properties values:

baudrate	baudrate value (default 9600)
databits	number of databits (5,6,7,8) (default 8)
stopbits	number of stopbits (1,2) (default 1)
parity	parity of device ('odd','even','none') (default 'none')

Outputs

dev - new created device object.

Properties

The object has the following public properties:

parent	The parent (arduino) for this device
interface	The interface type for this device ("SPI" or "I2C" or "Serial")

In addition, depending on type, the object will have these properties:

I2C Properties

The object has the following public properties:

bus	bus used for created object
i2caddress	I2C address set for object
sclpin	the SCL pin of the device
sdapin	the SDA pin of the device
bitrate	bit rate for the i2c clock

SPI Properties

The object has the following public properties:

spimode	mode used for created object
bitrate	Bitrate set for object
bitorder	Bitorder set for object
spichipselectpin	Pin used for chipselect
mosipin	Pin used for mosi
misopin	Pin used for miso
sckpin	Pin used for sckpin

Serial Properties

The object has the following public properties:

id	serial port id
baudrate	baudrate
databits	number of databits (5,6,7,8)
stopbits	number of stopbits (1,2)
parity	parity of device ('odd','even','none')

See also: arduino, i2cdev, spidev.

9.9.3 @device/display

`display (`*`dev`*`)`
Display device object.

Inputs

dev - device object to display

See also: device.

9.9.4 @device/flush

`data` `= flush (`*`dev`*`)`
`data` `= flush (`*`dev`*`, "`*`input`*`")`
`data` `= flush (`*`dev`*`, "`*`output`*`")`
Flush the serial port buffers

Inputs

dev - connected serial device opened using device

If an additional parameter is provided of "input" or "output", then only the input or output buffer will be flushed

Outputs

None

See also: arduino, device, read.

9.9.5 @device/read

`data = read (dev, numbytes)`
`data = read (dev, numbytes, precision)`

Read a specified number of bytes from a i2c or serial device object using optional precision for bytesize.

Inputs

dev - connected i2c or serial device opened using device

numbytes - number of bytes to read.

precision - Optional precision for the output data read data. Currently known precision values are uint8 (default), int8, uint16, int16

Outputs

data - data read from the device

See also: arduino, device.

9.9.6 @device/readRegister

`data = readRegister (dev, reg, numbytes)`
`data = readRegister (dev, reg, numbytes, precision)`

Read a specified number of bytes from a register of an i2cdev object using optional precision for bytesize.

Inputs

dev - connected i2c device opened using device

reg - registry value number

numbytes - number of bytes to read.

precision - Optional precision for the output data read data. Currently known precision values are uint8 (default), int8, uint16, int16

Output

data - data read from device.

See also: arduino, device.

9.9.7 @device/subsref

`val =  subsref (dev, sub)`

subref for device

See also: device.

9.9.8 @device/write

`write` (*dev*, *datain*)
`write` (*dev*, *datain*, *precision*)
Write data to a I2C or serial device object using optional precision for the data byte used for the data.

Inputs

dev - connected i2c or serial device opened using device

datain - data to write to device. Datasize should not exceed the constraints of the data type specified for the precision.

precision - Optional precision for the input write data. Currently known precision values are uint8 (default), int8, uint16, int16

See also: arduino, device, read.

9.9.9 @device/writeRead

dataOut = `readWrite` (*spi*, *dataIn*)
Write uint8 data to spi device and return back clocked out response data of same size.

Inputs

spi - connected spi device on arduino

dataIn - uint8 sized data to send to spi device framed between SS frame.

Outputs

dataOut - uint8 data clocked out during send to dataIn.

See also: arduino, device.

9.9.10 @device/writeRegister

`writeRegister` (*dev*, *reg*, *datain*)
`writeRegister` (*dev*, *dev*, *datain*, *precision*)
Write data to i2c device object at a given registry position using optional precision for the data byte used for the data.

Inputs

dev - connected i2c device opened using device

reg - registry position to write to.

datain - data to write to device. Datasize should not exceed the constraints of the data type specified for the precision.

precision - Optional precision for the input write data. Currently known precision values are uint8 (default), int8, uint16, int16

See also: arduino, device, read.

9.10 Arduino Ultrasonic Functions

9.10.1 @ultrasonic/delete

`delete` (*dev*)
Free resources of a ultrasonic object.

Inputs

dev - ultrasonic object to free

See also: ultrasonic.

9.10.2 @ultrasonic/display

`display (dev)`
Display ultrasonic object.

Inputs

dev - ultrasonic object to display

See also: ultrasonic.

9.10.3 @ultrasonic/readDistance

`distance = readDistance (dev)`
Read the distance from a ultrasonic device

Inputs

dev - connected ultrasonic device opened using ultrasonic

Outputs

distance - distance value in meters from the ultrasonic device, or Inf if out of sensor range

See also: arduino, ultrasonic.

9.10.4 @ultrasonic/readEchoTime

`time = readEchoTime (dev)`
Measure the time for waves to reflect back to the ultrasonic device

Inputs

dev - connected ultrasonic device opened using ultrasonic()

Outputs

time - time in seconds, or Inf if out of sensor range

See also: arduino, ultrasonic.

9.10.5 @ultrasonic/subsref

`val =  subsref (dev, sub)`
subref for ultrasonic

See also: ultrasonic.

9.10.6 @ultrasonic/ultrasonic

`dev = ultrasonic (ar, triggerpin)`
`dev = ultrasonic (ar, triggerpin, echopin)`
`dev = ultrasonic (ar, triggerpin, echopin, propname, propvalue)`
Create an ultrasonic object to communicate to a connected ultrasonic device

Inputs

ar - connected arduino object

triggerpin - trigger pin for attached device.

echopin - trigger pin for attached device.

propname, *propvalue* - property name/value pair for values to pass to devices.

Currently known properties:

outputformat
: string designating number format for output ('double')

Outputs

dev - created ultrasonic object

Properties

The ultrasonic object has the following public properties:

parent
: The parent (arduino) for this device

pins
: pins used by this object

triggerpin
: trigger used for created object

echopin
: Echo pin set for object

outputformat
: Output format for the created object

See also: arduino, readDistance, readEchoTime.

9.11 Arduino Addons

9.11.1 addon

`retval = addon (ar, addonname)`
`retval = addon (ar, addonname, varargs)`

Create an addon object using the addon named class.

Inputs

ar - connected arduino object

addonname - the name of the addon to create. The addon name can be a user addon or an inbuilt addon, however must appear in the listArduinoLibraries output and have been programmed onto the arduino.

varargs - optional values that will be provided verbatim to the the addon class constructor.

Outputs

retval - cell array of string library names.

See also: arduino, arduinosetup, listArduinoLibraries.

9.11.2 arduinoioaddons.EEPRomAddon.EEPRom

`arduinoioaddons.EEPRomAddon.EEPRom`

EEPROM addon for arduino

Allows read and write of uint8 data to the onboard arduino EEPROM.

Example

Assuming eeprom addon has been programmed into the Arduino:

```
a = arduino ();
e = addon (a, "eepromaddon/eeprom");
write (e, 0, uint8("hello world"));
str = uint8( read(e, 0, 11) )
```

See also: addon.

Properties

length - Size of the EEPROM.

Methods

eeprom = EEPRom ()
Constructor to create EEPROM device.

Outputs

eeprom - created EEPROM device.

erase ()
Erase all values in EEPROM (Effectively setting the 0xFF)

write (*address*, *uintdata*)
Write data to EEPROM at the provided address.

Inputs

address - start address to write data to, should be an integer between 0 and the size of the EEPROM.

uintdata a value or array of uint8 data to write to EEPROM.

data = read (*address*)
data = read (*address*, *count*)
Read data from starting address of EEPROM.

Inputs

address - start address to read data from, should be an integer between 0 and the size of the EEPROM.

count - Number of uint8 values to read from the EEPROM (default is 1)

Outputs

data a value or array of uint8 data read from the EEPROM.

9.11.3 arduinoioaddons.ExampleAddon.Echo

arduinoioaddons.ExampleAddon.Echo
Basic Example matlab/octave code to illustrate creating a user addon.

See also: addon.

Properties

Parent - the parent arduino object.
Pins - the pins allocated the addon.

Methods

obj = Echo(arObj)
Constructor to create Echo addon

Inputs

arObj - the arduino parent object

Outputs

obj - created Echo object

response = shout(*text*)
Send text to arduino and receive back the echoed reply

Inputs

text - text to send to arduino

Outputs

response - response from the arduino, which should be the same as the input text.

9.11.4 arduinoioaddons.ExampleLCD.LCD

arduinoioaddons.LCDAddon.LCD
Basic Example octave addon for LCD

Allows basic manipulation of an LCD as a illustration of using the addon functionality.

Example

Assuming the arduino has been programmed with the lcd addon:

```
a = arduino();
lcd = addon(a, "examplelcd/lcd", "d8", "d9", "d4", "d5", "d6", "d7")
clearLCD(lcd);
printLCD(lcd, "Hello");
# go to next line
gotoLCD(lcd, 0, 1);
printLCD(lcd, "World");
```

See also: addon.

Properties

Pins - the pins allocated the LCD display.

Methods

lcd = LCD(arObj, *rs, enable, d0, d1, d2, d3*)
Constructor to create LCD device

Inputs

arObj - the arduino parent object

rs - the pin to use for the rs line.

enable - the pin to use for the enable line.

d0 - the pin to use for the d0 line.

d1 - the pin to use for the d1 line.

d2 - the pin to use for the d2 line.

d3 - the pin to use for the d3 line.

Outputs

lcd - created LCD object

`freeLCD()`
Free the LCD

Should be called before discarding the LCD

Inputs

None.

Outputs

None.

`clearLCD()`
Clear the LCD display and set the cursor position to the home position.

Inputs

None.

Outputs

None.

`printLCD(text)`
Display text on LCD starting at the current cursor position.

Inputs

text - text to display on LCD

Outputs

None.

`gotoLCD(col, row)`
Set the cursor position to row, col

Inputs

col - 0 indexed LCD column to position to.

row - 0 indexed LCD row to position to.

Outputs

None.

9.11.5 arduinoioaddons.RTCAddon.DS1307

`arduinoioaddons.RTCAddon.DS1307`
DS1307 addon

See also: addon.

Properties

Parent - the parent arduino object.

Pins - the pins allocated the addon.

Methods

`obj = DS1307(arObj)`
`obj = DS1307(arObj, propertyname, propertyvalue ....)`

Constructor to create DS1307 addon

Inputs

arObj - the arduino parent object

propertyname, propertyvalue - optional property name, value pairs. Current known properties are:

address I2C address of the DS1307 (default 0x68)

Outputs

obj - created DS1307 object

Example

```
a = arduino()
rtc = addon(a, "rtcaddon/ds1307")
```

`date = clock(dsObj)`
`clock(dsObj, date)`

Get/set the DS1307 clock

Inputs

dsObj - the ds1307 object

date - a date vector in same format as datevec and clock

Outputs

date - a date vector in same format as datevec and clock

Example

```
a = arduino()
rtc = addon(a, "rtcaddon/ds1307")
# get and display rtc time as a date string
datestr(rtc.clock)
```

See also: datevec.

`ctrl = control(dsObj)`
`control(dsObj, ctrl)`

Get/set the DS1307 clock

Inputs

dsObj - the ds1307 object

ctrl - a structure containing the control bit fields.

Outputs

ctrl - a structure containing the control bit fields.

Control structure fields are: Current properties are:

out	Out bit in the control register
sqwe	Square wave enable bit in control register
rs	The combined RS0, RS1 value

`YN = isstarted(dsObj)`
Get whether the RTC clock is currently counting time

Inputs

dsObj - the ds1307 object

Outputs

YN - returns true if the RTC is counting

See also: start, stop.

`start(dsObj)`
Start the RTC counting

Inputs

dsObj - the ds1307 object

Outputs

None

See also: datevec.

`stop(dsObj)`
Stop the RTC counting

Inputs

dsObj - the ds1307 object

Outputs

None

See also: datevec.

9.11.6 arduinoioaddons.SimpleStepper.SimpleStepper

`arduinoioaddons.SimpleStepper`
Stepper class for stepper control using ULN2003 and compatible drivers

Properties

Id	Id of the stepper (Read only)
Speed	Number of steps to do per second.
Status	Status of stepper (Read only). 0 = not moving, 1 = moving, 2 = rotating
Parent	the Arduino parent (read only)
Pins	the pins used for the stepper (read only)

Methods

`obj = SimpleStepper(aObj, pin1, pin2, pin3, pin4)`
`obj = SimpleStepper(aObj, pin1, pin2, pin3, pin4, pin5)`

Constructor to create a stepper object

Inputs

aObj - The arduino

pin1 - The first pin of the controller

pin2 - The second pin of the controller

pin3 - The third pin of the controller

pin4 - The fourth pin of the controller

pin5 - The fifth pin of the controller

Outputs

s - a simplestepper object

Example

```
a = arduino()
# create stepper object
s = addon(a, "simplestepper/simplestepper", "d2", "d3", "d4", "d5")
# start rotating left
s.rotate(-1);
```

See also: addon.

`move(sObj, steps)`

Move the motor the specified number of steps using the configured Speed.

Inputs

sObj - the stepper object

steps - the number of steps to move. steps less than 0 will be moving left.

Outputs

None

See also: rotate.

`rotate(sObj, dir)`

Start steppermotor moving in the specified direction using the configured Speed.

Inputs

sObj - the stepper object

dir - Direction to move. -1 = left, 0 = stop, 1 = right.

Outputs

None

See also: move.

`release(sObj)`
Release this stepper motor

Inputs

sObj - the stepper object

Outputs

None

9.11.7 arduinoioaddons.adafruit.dcmotorv2

`arduinoioaddons.adafruit.dcmotorv2`
DC Motor class for dc motor control on the adafruit motor shield

See also: arduinoioaddons.adafruit.motorshieldv2.

Properties

Speed - The speed value set for the motor

Parent - The parent shield for object (read only)

MotorNumber - The motor number (read only) values 1-4

IsRunning - boolean for if the motor is started (read only)

Methods

`obj = dcmotorv2(mObj, mnum)`
`obj = dcmotorv2(mObj, mnum, propertyname, propertyvalue ....)`
Constructor to create dcmotor object

Inputs

mObj - the motor shield object

mnum - The motor number (1 - 4)

propertyname, propertyvalue - Optional property name/value pairs to pass to motor object.

Current known properties are:

Speed Initial speed (default 0). Should be a value between -1 and 1.

Outputs

s - a dcmotorv2 object

Example

```
a = arduino()
ms = addon(a, "adafruit/motorshieldv2")
mtr = dcmotor(ms, 1)
```

`start(`*dcObj*`)`
Start the motor moving in previously set speed/direction

Inputs

dcObj - the dcmotor object

Outputs

None

See also: adafruit.motorshieldv2.

`stop(`*dcObj*`)`
Stop the motor moving

Inputs

dcObj - the dcmotor object

Outputs

None

See also: adafruit.motorshieldv2.

9.11.8 arduinoioaddons.adafruit.motorshieldv2

`arduinoioaddons.adafruit.motorshieldv2`
Adafruit motor shield addon

See also: addon.

Properties

Parent - the parent arduino object.
Pins - the pins allocated the addon.
I2CAddress - the i2c address used for accessing this shield.
PWMFrequency - the set PWM frequency for this shield.

Methods

obj `= motorshieldv2(`*arObj*`)`
obj `= motorshieldv2(`*arObj*`, `*propertyname, propertyvalue ....*`)`
Constructor to create motorshieldv2 addon object

Inputs

arObj - the arduino parent object

propertyname, propertyvalue - optional property name, value pairs. Current known properties are:

address I2C address of the motor shield (default 0x60)

pwmfrequency
PWM Frequency to set on shield (default 1600)

Outputs

obj - created motorshieldv2 object

Example

```
a = arduino()
mtr = addon(a, "adafruit/motorshieldv2")
```

`s = servo(mObj, mtrnum)`
`s = servo(mObj, mtrnum, propertyname, propertyvalue ...)`
Create a servo object

Inputs

mObj - the motor shield object

mtrnum - The servo motor number, where 1 is servo on pin "d10" and 2 is a servo on pin "d9"

propertyname, *propertyvalue* - Optional property name/value pairs to pass to servo object.

Properties are the same as the base servo object.

Outputs

s - a servo object

Example

```
a = arduino()
ms = addon(a, "adafruit/motorshieldv2")
# get servo 1 (servo on pin D10)
s = ms.servo(1)
```

The function if the equivalent of calling the arduino.servo with the D9 or D10 pin has the input pin.

See also: servo.

`s = stepper(mObj, mtrnum, stepsperrev)`
`s = stepper(mObj, mtrnum, stepsperrev, propertyname, propertyvalue ...)`
Create a stepper motor object

Inputs

mObj - the motor shield object

mtrnum - The stepper motor number (1 or 2)

stepsperrev - Number of steps per revolution.

propertyname, *propertyvalue* - Optional property name/value pairs to pass to stepper object.

Outputs

s - a stepper object

`s = dcmotor(mObj, mtrnum)`
`s = dcmotor(mObj, mtrnum, propertyname, propertyvalue ...)`
Create a dcmotor motor object

Inputs

mObj - the motor shield object

mtrnum - The motor number (1 - 4)

propertyname, *propertyvalue* - Optional property name/value pairs to pass to motor object.

Outputs

s - a dcmotorv2 object

9.11.9 arduinoioaddons.adafruit.stepper

`arduinoioaddons.adafruit.stepper`
Stepper class for stepper control on the adafruit motor shield

See also: arduinoioaddons.adafruit.motorshieldv2.

Properties

RPM The rpm value set for the stepper motor

StepType the StepType for the stepper (string) which can be "single", "double", "interleave" or "microstep"

StepsPerRevolution
the StepsPerRevoluion for the stepper (read only)

MotorNumber
the motor number for the stepper (read only) value will be 1 or 2.

Parent the parent shield of this stepper (read only)

Methods

`obj = stepper(mObj, mnum, stepsperrev)`
`obj = stepper(mObj, mnum, stepsperrev, propertyname, propertyvalue ....)`
Constructor to create dcmotor object

Inputs

mObj - the motor shield object

mnum - The motor number (1 or 2)

stepsperrev - Number of steps per revolution.

propertyname, *propertyvalue* - Optional property name/value pairs to pass to motor object.

Current known properties are:

RPM the RPM for the stepper (revolutions per minute)

StepType the StepType for the stepper (string) which can be "single", "double", "interleave" or "microstep"

Outputs

s - a stepper object

Example

```
a = arduino()
ms = addon(a, "adafruit/motorshieldv2")
mtr = stepper(ms, 1, 200)
```

`move(`*`sObj, steps`*`)`
Move the motor moving in the specified steps using the configured RPM.

Inputs

sObj - the stepper object

Outputs

None

See also: adafruit.motorshieldv2.

`release(`*`sObj`*`)`
Release this motor

Inputs

sObj - the stepper object

Outputs

None

See also: adafruit.motorshieldv2.

9.12 Arduino Sensors

9.12.1 arduinosensor.DS1307

`arduinosensor.DS1307`
DS1307 realtime clock sensor

Methods

`obj` `= DS1307(`*`arObj`*`)`
`obj` `= DS1307(`*`arObj, propertyname, propertyvalue ....`*`)`
Constructor to create DS1307 sensor

Inputs

arObj - the arduino parent object

propertyname, propertyvalue - optional property name, value pairs. Current known properties are: Current properties are:

i2caddress I2C address of the DS1307 (default 0x68)

Outputs

obj - created DS1307 object

Example

```
a = arduino()
rtc = arduinosensor.DS1307(a)
```

`date = clock(dsObj)`
`clock(dsObj, date)`
Get/set the DS1307 clock

Inputs

dsObj - the ds1307 object

date - a date vector in same format as datevec and clock

Outputs

date - a date vector in same format as datevec and clock

Example

```
a = arduino()
rtc = arduinosensor.DS1307(a)
# get and display rtc time as a date string
datestr(rtc.clock)
```

See also: datevec.

`ctrl = control(dsObj)`
`control(dsObj, ctrl)`
Get/set the DS1307 clock

Inputs

dsObj - the ds1307 object

ctrl - a structure containing the control bit fields.

Outputs

ctrl - a structure containing the control bit fields.

Control structure fields are: Current properties are:

out	Out bit in the control register
sqwe	Square wave enable bit in control register
rs	The combined RS0, RS1 value

`YN = isstarted(dsObj)`
Get whether the RTC clock is currently counting time

Inputs

dsObj - the ds1307 object

Outputs

YN - returns true if the RTC is counting

See also: start, stop.

`start(dsObj)`
Start the RTC counting

Inputs

dsObj - the ds1307 object

Outputs

None

See also: datevec.

`stop(dsObj)`
Stop the RTC counting

Inputs

dsObj - the ds1307 object

Outputs

None

See also: datevec.

9.12.2 arduinosensor.GUVAS12SD

`arduinosensor.GUVAS12SD`
A thin wrapper for the GUVAS12SD analog UV-B sensor

Methods

`obj = GUVAS12SD(arObj, pin)`
Constructor to create GUVAS12SD sensor

Inputs

arObj - the arduino parent object

pin - the analog pin that the sensor is connected to

Outputs

obj - created GUVAS12SD object

Example

```
a = arduino()
# create sensor attached to pin a0.
sensor = arduinosensor.GUVAS12SD(a, "a0")
```

`V = read(dsObj)`
Read the voltage of the sensor

Inputs

dsObj - the GUVAS12SD object

Outputs

V - read voltage - effectively equivalent to readAnalogPin(arObj, pin).

Example

```
a = arduino()
s = arduinosensor.GUVAS12SD(a)
# voltage
volts = s.read
```

See also: arduinosensor.GUVAS12SD.

`Idx = readIndex(dsObj)`
Read the UV index

Inputs

dsObj - the GUVAS12SD object

Outputs

Idx - the sensor reading as a UV index reading

`uA = readuA(dsObj)`
Read the uA of the sensor

Inputs

dsObj - the GUVAS12SD object

Outputs

uA - the sensor reading as a uAmp value

9.12.3 arduinosensor.MPC3002

`arduinosensor.MPC3002`
MCP3002 ADC sensor

Methods

`obj = MPC3002(arObj, selectPin)`
`obj = MPC3002(arObj, selectPin, propertyname, propertyvalue ....)`
Constructor to create MPC3002 sensor

Inputs

arObj - the arduino parent object

selectPin - the SPI cs select pin

propertyname, propertyvalue - optional property name, value pairs.

Current properties are:

referenceVoltage
: Reference voltage for scaling the ADC inputs (default 5.0)

Outputs

obj - created MCP3002 object

Example

```
a = arduino()
sensor = arduinosensor.MPC3002(a, "d10")
```

voltage = readVoltage(*dsObj, chan*)

Read the voltage from a channel

Inputs

dsObj - the MPC3002 object

chan - the channel to read (0 or 1)

Outputs

voltage - read voltage.

Example

```
a = arduino()
s = arduinosensor.MPC3002(a, "d10")
volts = readVoltage(s, 0)
```

See also: arduinosensor.MPC3002.

9.12.4 arduinosensor.SI7021

arduinosensor.SI7021

SI7021 temperature and humidity sensor

Methods

obj = SI7021(*arObj*)
obj = SI7021(*arObj, propertyname, propertyvalue* ....)

Constructor to create SI7021 sensor

Inputs

arObj - the arduino parent object

propertyname, propertyvalue - optional property name, value pairs. Current known properties are: Current properties are:

i2caddress I2C address of the SI7021 (default 0x40)

Outputs

obj - created SI7020 object

Example

```
a = arduino()
sensor = arduinosensor.SI7021(a)
```

`C = temperature(dsObj)`
Read the temperature

Inputs

dsObj - the si7021 object

Outputs

C - read temperature in deg C.

Example

```
a = arduino()
s = arduinosensor.SI7021(a)
# get temp
temp = s.temperature
```

See also: arduinosensor.SI7021.

`relH = humidity(dsObj)`
Read the relative humidity

Inputs

dsObj - the si7021 object

Outputs

relH - relative humidity as a percentage (0 - 100.0)

`relH = info(dsObj)`
Read the sensor info

Inputs

dsObj - the si7021 object

Outputs

inf - structure containing the sensor information.

Structure fields are:

version	Chip firmware version
id	sensor id1,id2 value
type	String for detected chip type

9.13 Arduino I/O package

9.13.1 arduinoio.AddonBase

`arduinoio.AddonBase`
Base class used for arduino library sensors

See also: arduinoio.LibraryBase.

Properties

Base properties are expected to be inherited and overwritten in inherited classes. and are constant in order to query through the metaobject mechanism.

Parent - parent librarybase object

Methods

ab = `AddonBase` ()
Constructor of base class

Outputs

The return value *ab* is an object of the arduinio.AddonBase class.

See also: arduino, addon.

`display` ()
Display the addon in a verbose way.

9.13.2 arduinoio.FilePath

retval = `arduinoio.FilePath` (*fullpathname*)
Get the directory component of a pathname.

Inputs

fullpathname filepath to get directory component of.

Outputs

retval the directory part of the filename.

9.13.3 arduinoio.LibFiles

filelist = `arduinoio.LibFiles` ()
Get the list of files used for the building arduino library

Outputs

filelist - string cell array of files for the arduino project

9.13.4 arduinoio.LibraryBase

`arduinoio.LibraryBase`
Base class used for arduino library plugins

See also: arduino, listArduinoLibraries, addon.

Properties

Base properties are expected to be inherited and overwritten in inherited classes and are constant in order to query through the metaobject mechanism.

LibraryName - name of the addon library

DependentLibraries - array of dependent library names that must be included when installing this plugin.

CppHeaderFile - name (if any) of header file that will be included into the arduino project when adding this library.

CppSourceFile - name (if any) of source file that will be included into the arduino project when adding this library.

CppClassName - name of the cpp class for the addon library. project when adding this library.

Pins - pins allocated to the addon

Parent - parent arduino object.

Methods

`lb = LibraryBase ()`
Constructor of base class

The constructor is usually not called but called indirectly from the addon function.

Outputs

The return value *lb* is an object of the arduinio.LibraryBase class.

See also: arduino, listArduinoLibraries, addon.

`display ()`
Display the addon in a verbose way.

9.13.5 arduinoio.getBoardConfig

`retval = arduinoio.getBoardConfig (boardname)`
Return the configuration for a known arduino board type

Function is used to get the expected pin/board configuration for a named board type which is used to verify and identify the functionality of the board.

Inputs

boardname - name of board to get configuration of ie: `"uno"`

Outputs

retval configuration struct.

9.14 Matlab Compatability Classes

9.14.1 matlabshared.addon.LibraryBase

`matlabshared.addon.LibraryBase`
Compatability class used for arduino library plugins using matlabshared.addons.LibraryBase

See also: arduinoio.LibraryBase, arduino, listArduinoLibraries, addon.

Properties

Base properties are expected to be inherited and overwritten in inherited classes and are constant in order to query through the metaobject mechanism.

LibraryName - name of the addon library

DependentLibraries - array of dependent library names that must be included when installing this plugin.

CppHeaderFile - name (if any) of header file that will be included into the arduino project when adding this library.

CppSourceFile - name (if any) of source file that will be included into the arduino project when adding this library.

CppClassName - name of the cpp class for the addon library. project when adding this library.

Pins - pins allocated to the addon

Parent - parent arduino object.

Methods

`lb = LibraryBase ()`
Constructor of base class

The constructor is usually not called but called indirectly from the addon function.

Outputs

The return value *lb* is an object of the matlabshare.addons.LibraryBase class.

See also: arduino, listArduinoLibraries, addon.

`display ()`
Display the addon in a verbose way.

9.15 Sensors

9.15.1 bme280

`bme280`
BME280 pressure, temperature and humidity sensor

Methods

`obj = bme280(arObj)`
`obj = bme280(arObj, propertyname, propertyvalue ....)`
Constructor to create BME280 sensor

Inputs

arObj - the arduino parent object

propertyname, propertyvalue - optional property name, value pairs. Current known properties are: Current properties are:

I2CAddress
: I2C address of the sensor (default 0x40)

Bus
: I2C bus - 0 or 1 (default 0)

Outputs

obj - created object

Example

```
a = arduino()
sensor = bme280(a)
```

`[C, timestamp] = readTemperature(obj)`
Read the temperature

Inputs

obj - the sensor object

Outputs

C - read temperature in deg C.

timestamp - timestamp when read

Example

```
a = arduino()
s = bme280(a)
# get temp
temp = s.readTemperature
```

See also: bme280.

`[relH, timestamp] = readHumidity(obj)`
Read the relative humidity

Inputs

obj - the sensor object

Outputs

relH - relative humidity as a percentage (0 - 100.0)

timestamp - timestamp when read

`[P, timestamp] = readPressure(obj)`
Read the pressure

Inputs

obj - the sensor object

Outputs

P - pressure reading from sensor.

timestamp - timestamp when read

`[readings, overrun] = read(obj)`
`[P, H, C, timestamp, overrun] = read(obj)`
Read the sensor data

Inputs

obj - the sensor object

Outputs

P - pressure reading from sensor.

H - humidity reading from sensor.

C - temperature reading from sensor.

timestamp - timestamp when read

overrun - overrun flag.

readings - table structure with fields for Timestamp, Pressure, Temperature and Humidity.

`inf = info(obj)`
Read the sensor info

Inputs

obj - the sensor object

Outputs

inf - structure containing the sensor information.

Structure fields are:

Version	Chip firmware version
SensorId	sensor id value
Type	sensor type 'bme280'
Status	Status value read from sensor

`flush(obj)`
Flush sensor data

Inputs

obj - the sensor object

Outputs

None

`release(obj)`
Release the resources of the sensor

Inputs

obj - the sensor object

Outputs

None

9.15.2 bno055

`bno055`
BNO055 9 axis orientation sensor

Methods

`obj = bno055(arObj)`
`obj = bno055(arObj, propertyname, propertyvalue ....)`
Constructor to create BME280 sensor

Inputs

arObj - the arduino parent object

propertyname, *propertyvalue* - optional property name, value pairs. Current known properties are: Current properties are:

I2CAddress
: I2C address of the sensor (default 0x40)

Bus
: I2C bus - 0 or 1 (default 0)

OperatingMode
: Operating mode 'ndof' or 'amg'

Outputs

obj - created object

Example

```
a = arduino()
sensor = bno055(a)
```

`[C, timestamp] = readTemperature(obj)`
Read the temperature

Inputs

obj - the sensor object

Outputs

C - read temperature in deg C.

timestamp - timestamp when read

Example

```
a = arduino()
s = bno055(a)
# get temp
temp = s.readTemperature
```

See also: bno055.

`[readVal, timestamp] = readAcceleration(obj)`
Read the acceleration rate

Inputs

obj - the sensor object

Outputs

readVal - the 3 acceleration values

timestamp - timestamp when read

`[readVal, timestamp] = readAngularVelocity(obj)`
Read the angular velocity

Inputs

obj - the sensor object

Outputs

readVal - the 3 angular velocity values

timestamp - timestamp when read

`[readVal, timestamp] = readMagneticField(obj)`
Read the magnetic field components

Inputs

obj - the sensor object

Outputs

readVal - the 3 magnetic field values

timestamp - timestamp when read

`[readVal, timestamp] = readOrientation(obj)`
Read the oriientation components

Inputs

obj - the sensor object

Outputs

readVal - the 3 orientation values

timestamp - timestamp when read

`[readings, overrun] = read(obj)`
`[accel, gyro, mag, timestamp, overrun] = read(obj)`
`[accel, gyro, mag, orientation, timestamp, overrun] = read(obj)`
Read the sensor data

Inputs

obj - the sensor object

Outputs

accel - acceleration reading from sensor.

gyro - angular acceleration reading from sensor.

mag - magnetic field reading from sensor.

orientation - orientation reading from sensor.

timestamp - timestamp when read

overrun - overrun flag.

readings - table structure with fields for Timestamp, Acceleration, AngularVelocity, MagneticField, Orientation.

`inf = readCalibrationStatus(obj)`
Read the sensor calibration status

Inputs

obj - the sensor object

Outputs

status - structure containing the calibration information.

Structure fields are:

System — System calibrarion

Accelerometer — Accelerometer calibration status

Gyroscope — Gyroscope calibration status

Magnetometer — Magnetometer calibration status

Values for each will be either 'uncalibrated', 'partial' or 'full'.

`inf = info(obj)`
Read the sensor info

Inputs

obj - the sensor object

Outputs

inf - structure containing the sensor information.

Structure fields are:

Version — Software firmware version

SensorId — sensor id value

Type — sensor type 'bno055'

`flush(obj)`
Flush sensor data

Inputs

obj - the sensor object

Outputs

None

`release(obj)`
Release the resources of the sensor

Inputs

obj - the sensor object

Outputs

None

9.15.3 lis3dh

`lis3dh`
LIS3DH 3 degrees sensor

Methods

obj = lis3dh(*arObj*)
obj = lis3dh(*arObj*, *propertyname*, *propertyvalue* ....)
Constructor to create LIS3DH sensor

Inputs

arObj - the arduino parent object

propertyname, propertyvalue - optional property name, value pairs. Current known properties are: Current properties are:

I2CAddress
: I2C address of the sensor (default 0x40)

Bus
: I2C bus - 0 or 1 (default 0)

Outputs

obj - created object

Example

```
a = arduino()
sensor = lis3dh(a)
```

[*readVal*, *timestamp*] = readAcceleration(*obj*)
Read the acceleration rate

Inputs

obj - the sensor object

Outputs

readVal - the 3 acceleration values

timestamp - timestamp when read

[*readings*, *overrun*] = read(*obj*)
[*accel*, *timestamp*, *overrun*] = read(*obj*)
Read the sensor data

Inputs

obj - the sensor object

Outputs

accel - acceleration reading from sensor.

timestamp - timestamp when read

overrun - overrun flag.

readings - table structure with fields for Timestamp, Acceleration.

`inf = info(obj)`
Read the sensor info

Inputs

obj - the sensor object

Outputs

inf - structure containing the sensor information.

Structure fields are:

SensorId	sensor id value
Type	sensor type 'lis3dh'
Status	sensor status value

`flush(obj)`
Flush sensor data

Inputs

obj - the sensor object

Outputs

None

`release(obj)`
Release the resources of the sensor

Inputs

obj - the sensor object

Outputs

None

9.15.4 lps22hb

`lps22hb`
LPS22HB absolute pressure and temperature sensor

Methods

`obj = lps22hb(arObj)`
`obj = lps22hb(arObj, propertyname, propertyvalue ....)`
Constructor to create LPS22HB sensor

Inputs

arObj - the arduino parent object

propertyname, propertyvalue - optional property name, value pairs. Current known properties are: Current properties are:

I2CAddress
: I2C address of the sensor (default 0x5C)

Bus
: I2C bus - 0 or 1 (default 0)

Outputs

obj - created object

Example

```
a = arduino()
sensor = lps22hb(a)
```

`[C, timestamp] = readTemperature(obj)`
Read the temperature

Inputs

obj - the sensor object

Outputs

C - read temperature in deg C.

timestamp - timestamp when read

Example

```
a = arduino()
s = lps22hb(a)
# get temp
temp = s.readTemperature
```

See also: lps22hb.

`[P, timestamp] = readPressure(obj)`
Read the pressure

Inputs

obj - the sensor object

Outputs

P - pressure reading from sensor.

timestamp - timestamp when read

`[readings, overrun] = read(obj)`
`[P, C, timestamp, overrun] = read(obj)`
Read the sensor data

Inputs

obj - the sensor object

Outputs

P - pressure reading from sensor.

C - temperature reading from sensor.

timestamp - timestamp when read

overrun - overrun flag.

readings - table structure with fields for Timestamp, Pressure, Temperature and Humidity.

`inf = info(obj)`
Read the sensor info

Inputs

obj - the sensor object

Outputs

inf - structure containing the sensor information.

Structure fields are:

Version	Chip firmware version
SensorId	sensor id value
Type	sensor type 'lps22hb'
Status	Status value read from sensor

`flush(obj)`
Flush sensor data

Inputs

obj - the sensor object

Outputs

None

`release(obj)`
Release the resources of the sensor

Inputs

obj - the sensor object

Outputs

None

9.15.5 lsm6dso

`lsm6dso`
LSM6DSO 6 degrees sensor

Methods

`obj = lsm6dso(arObj)`
`obj = lsm6dso(arObj, propertyname, propertyvalue ....)`
Constructor to create LSM6DSO sensor

Inputs

arObj - the arduino parent object

propertyname, *propertyvalue* - optional property name, value pairs. Current known properties are: Current properties are:

I2CAddress
: I2C address of the sensor (default 0x40)

Bus
: I2C bus - 0 or 1 (default 0)

Outputs

obj - created object

Example

```
a = arduino()
sensor = lsm6dso(a)
```

`[C, timestamp] = readTemperature(obj)`
Read the temperature

Inputs

obj - the sensor object

Outputs

C - read temperature in deg C.

timestamp - timestamp when read

Example

```
a = arduino()
s = lsm6dso(a)
# get temp
temp = s.readTemperature
```

See also: lsm6dso.

`[readVal, timestamp] = readAcceleration(obj)`
Read the acceleration rate

Inputs

obj - the sensor object

Outputs

readVal - the 3 acceleration values

timestamp - timestamp when read

`[readVal, timestamp] = readAngularVelocity(obj)`

Read the angular velocity

Inputs

obj - the sensor object

Outputs

readVal - the 3 angular velocity values

timestamp - timestamp when read

`[readings, overrun] = read(obj)`
`[accel, gyro, mag, timestamp, overrun] = read(obj)`

Read the sensor data

Inputs

obj - the sensor object

Outputs

accel - acceleration reading from sensor.

gyro - angular acceleration reading from sensor.

timestamp - timestamp when read

overrun - overrun flag.

readings - table structure with fields for Timestamp, Acceleration, AngularVelocity.

`inf = info(obj)`

Read the sensor info

Inputs

obj - the sensor object

Outputs

inf - structure containing the sensor information.

Structure fields are:

SensorId	sensor id value
Type	sensor type 'lsm6dso'

`flush(obj)`

Flush sensor data

Inputs

obj - the sensor object

Outputs

None

`release(obj)`

Release the resources of the sensor

Inputs

obj - the sensor object

Outputs

None

9.15.6 mpu6050

`mpu6050`
MPU-6050 6 degrees sensor

Methods

obj = mpu6050(*arObj*)
obj = mpu6050(*arObj*, *propertyname*, *propertyvalue* ....)
Constructor to create MPU-6050 sensor

Inputs

arObj - the arduino parent object

propertyname, *propertyvalue* - optional property name, value pairs. Current known properties are: Current properties are:

I2CAddress
: I2C address of the sensor (default 0x40)

Bus
: I2C bus - 0 or 1 (default 0)

Outputs

obj - created object

Example

```
a = arduino()
sensor = mpu6050(a)
```

[*C*, *timestamp*] = readTemperature(*obj*)
Read the temperature

Inputs

obj - the sensor object

Outputs

C - read temperature in deg C.

timestamp - timestamp when read

Example

```
a = arduino()
s = mpu6050(a)
# get temp
```

```
temp = s.readTemperature
```

See also: mpu6050.

`[readVal, timestamp] = readAcceleration(obj)`
Read the acceleration rate

Inputs

obj - the sensor object

Outputs

readVal - the 3 acceleration values

timestamp - timestamp when read

`[readVal, timestamp] = readAngularVelocity(obj)`
Read the angular velocity

Inputs

obj - the sensor object

Outputs

readVal - the 3 angular velocity values

timestamp - timestamp when read

`[readings, overrun] = read(obj)`
`[accel, gyro, mag, timestamp, overrun] = read(obj)`
Read the sensor data

Inputs

obj - the sensor object

Outputs

accel - acceleration reading from sensor.

gyro - angular acceleration reading from sensor.

timestamp - timestamp when read

overrun - overrun flag.

readings - table structure with fields for Timestamp, Acceleration, AngularVelocity.

`inf = info(obj)`
Read the sensor info

Inputs

obj - the sensor object

Outputs

inf - structure containing the sensor information.

Structure fields are:

SensorId	sensor id value
Type	sensor type 'mpu6050'

`flush(obj)`
Flush sensor data

Inputs

obj - the sensor object

Outputs

None

`release(obj)`
Release the resources of the sensor

Inputs

obj - the sensor object

Outputs

None

9.15.7 si7021

`si7021`
SI7021 temperature and humidity sensor

Methods

`obj = si7021(arObj)`
`obj = si7021(arObj, propertyname, propertyvalue ....)`
Constructor to create si7021 sensor

Inputs

arObj - the arduino parent object

propertyname, propertyvalue - optional property name, value pairs. Current known properties are: Current properties are:

I2Caddress
: I2C address of the si7021 (default 0x40)

Bus
: I2C bus (default 0)

Outputs

obj - created SI7020 object

Example

```
a = arduino()
sensor = si7021(a)
```

`[C, timestamp] = readTemperature(obj)`
Read the temperature

Inputs

obj - the si7021 object

Outputs

C - read temperature in deg C.

timestamp - timestamp when read

Example

```
a = arduino()
s = si7021(a)
# get temp
temp = s.readTemperature()
```

See also: si7021.

`[relH, timestamp] = readHumidity(obj)`

Read the relative humidity

Inputs

obj - the si7021 object

Outputs

relH - relative humidity as a percentage (0 - 100.0)

timestamp - timestamp when read

`[readings, overrun] = read(obj)`
`[H, C, timestamp, overrun] = read(obj)`

Read the sensor data

Inputs

obj - the si2071 sensor object

Outputs

H - humidity reading from sensor.

C - temperature reading from sensor.

timestamp - timestamp when read

overrun - overrun flag.

readings - table structure with fields for Timestamp, Temperature and Humidity.

`relH = info(dsObj)`

Read the sensor info

Inputs

dsObj - the si7021 object

Outputs

inf - structure containing the sensor information.

Structure fields are:

Version	Chip firmware version
SensorDd	sensor id value
Type	String for detected chip type

`flush(obj)`
Flush sensor data

Inputs

obj - the sensor object

Outputs

None

`release(obj)`
Release the resources of the sensor

Inputs

obj - the sensor object

Outputs

None

9.16 Test Functions

9.16.1 arduino_bistsetup

`retval = arduino_bistsetup ()`
`retval = arduino_bistsetup (propertyname, propertyvalue)`
Install on an arduino the required core libraries to run the BIST tests

As part of the setup, the arduino IDE will be opened to allow programming the arduino board.

Inputs

propertyname, *propertyvalue* - A sequence of property name/value pairs can be given to set defaults while programming.

Currently the following properties can be set:

arduinobinary
: The value should be the name/path of the arduino IDE binary for programming. If not specified, the function will attempt to find the binary itself.

debug
: Set the debug flag when checking the arduino

Outputs

retval - return 1 if everything installed ok

See also: arduino, arduinosetup.

Appendix A GNU General Public License

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.
   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable

section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything

designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

    If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

    If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

    Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRIT-

ING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
```

```
This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘`show w`’ and ‘`show c`’ should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read http://www.gnu.org/philosophy/why-not-lgpl.html.

Index

M

O

P

Q

R

S

T

U

V

W