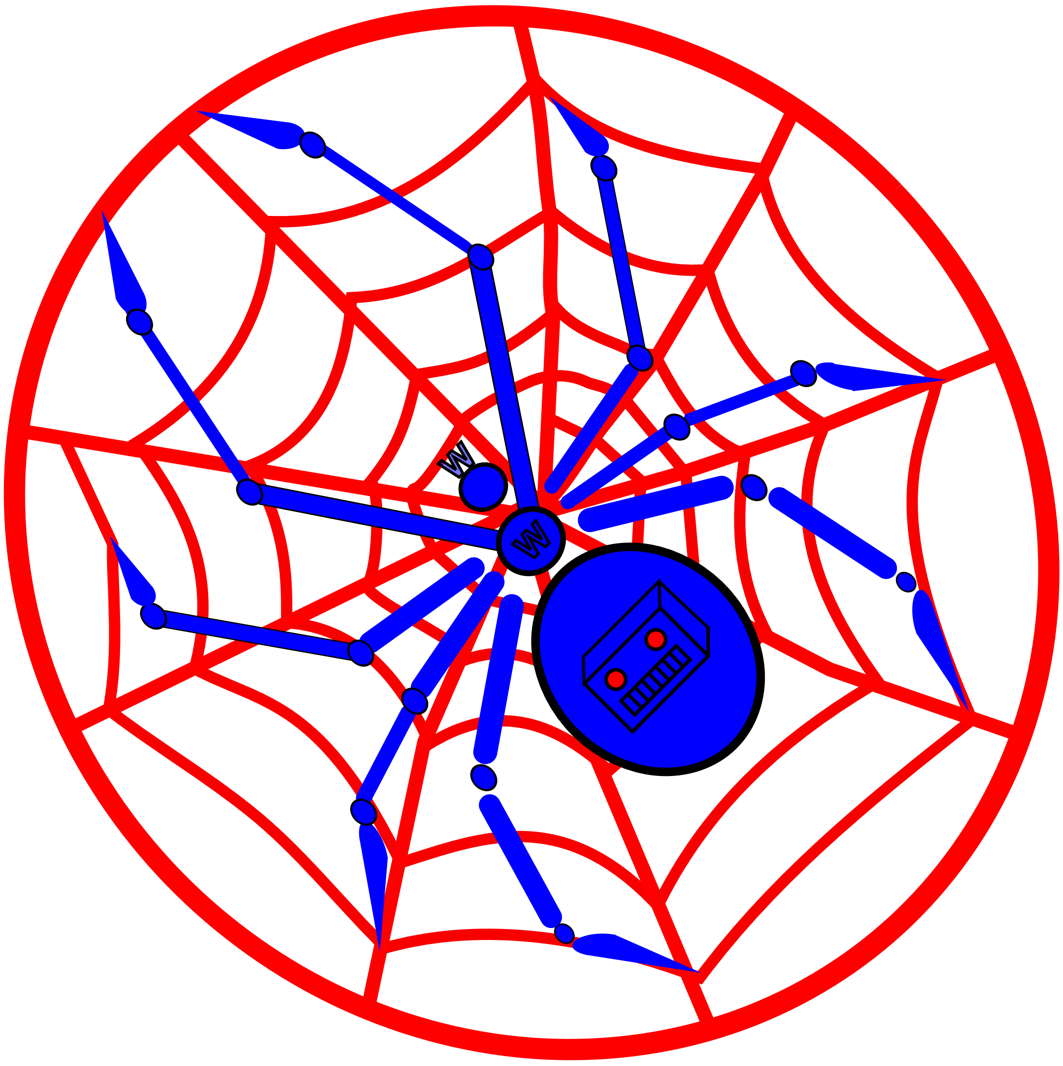


# WebbotLib for C++

21 April 2012

Contents

## Overview

WebbotLib is a C/C++ library for commercial boards like the Axon, Axon II and Roboduino, as well as custom boards based on various ATMega chips from ATmel.

Offering support for a large collection of sensors and motor controllers this is, perhaps, the most complete library available for robot builders.

The major change compared to Version 1 is that you must use Project Designer to create your project.

# The Webbot tools

To understand how WebbotLib works then you need to understand the difference between Boards and Projects.

A **Board** is the definition of a hardware board design - ie it will use a particular processor, at a given clock speed, and may be augmented with various devices such as UARTs, push buttons, motor controllers, etc which are built onto the board. Most processor pins, but not always all, are made available via header pins or sockets. Other header pins are generally available for power and ground supplies. WebbotLib includes some pre-built board designs for some of the boards that are commercially available including, but not restricted to, the Axon series from www.societyofrobots.com. Of course you may be using either a 'home made' board or a commercial board that WebbotLib doesn't know about. In this case you will need to run the Board Designer to define the board and save it as a Board design. Note: if it is a commercial board then let me know and I can add it to the list of 'built-in' boards that WebbotLib knows about. Of course you only need to create the Board once - or not at all if its already available in the list of choices.

A **Project** is based on an existing **Board** design (either one that I ship or one you've created yourself in Board Designer) and is created using Project Designer. Having selected the appropriate board you then specify what additional devices (ie sensors, motor controllers etc) you are going to plug into your board to create a given project. Hence you can create many projects from the one board design. Project Designer then allows you to generate the code for your project. This code includes all the setup and initialisation for the devices you are using as well as some example code showing how to use the device.

# Installing the library

This library can be used with Windows, Mac and Unix variants. It relies on several of my other utilities which are written using Java in order to work on these different operating systems.

## Downloading a Java JRE

So the first pre-requisite is a Java JRE (Java Runtime Environment) Version 6, also known as 1.6, or above.

To find out if you've already got a copy of Java then, on Windows, you can start a command prompt and type in: **java -version**

This will either show you an installed version number or may say that it is an invalid command if you don't have Java installed at all.

You can always download the latest Java JRE for free from: **http://www.oracle.com/technetwork/java/javase/downloads** and follow the installation details.

## Downloading WinAVR

WinAVR contains the avr-gcc C compiler and other utilities required to turn your program into a file to load onto your robot.

If you've already been using ATmel chips then you've probably already got it. Otherwise download it from: **http://winavr.sourceforge.net/download.html**

## Download Webbot Downloader

This is a handy utility that I have produced to help make sure that you are using the latest versions of my code. Each time you run it - it will automatically check, if you are online, that you've got the latest versions and download any new versions if required.

It makes installing the rest of WebbotLib a breeze!

To download it read my page at **http://webbot.org.uk/iPoint/48.page** and follow the instructions.

Once downloaded then run the downloader as described and it will automatically download everything else you need. If you've never used WebbotLib before then this may take a while and you need to wait until each entry has a green tick against it.

Once it has finished downloading then you have got copies of all of my software which are described on my website **http://webbot.org.uk**

## Finished

Ok - that's it - you've got everything you need to start programming robots.

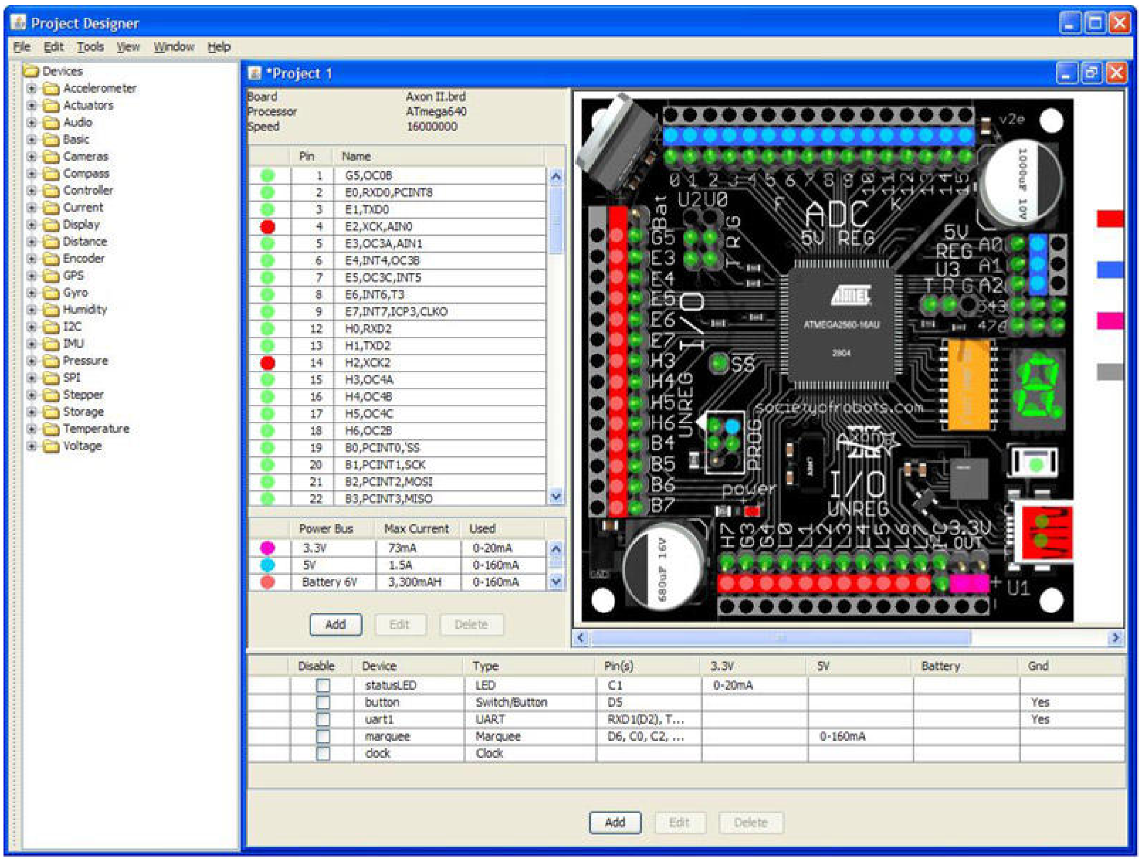
If you are using Windows then you may also want to download AVRStudio but WebbotLib does not rely upon it.

# My first program

Now that our environment has been set up correctly we can concentrate on writing some code.

We will start by writing a simple program that sends the text 'Hello World' out of the serial port every second. Ok - I know that sounds very dull - but finding out how to get your Robot to send messages back to your computer is essential when you want to start debugging your own programs.

Start up Project Designer and select File New as we want to create a new project. Select the board you are using from the list available. You will then be asked the details about the sort of battery you are using to power the board. You will now see something like the following (for the AxonII):-



| | |
|---|---|
| Board | Axon II.brd |
| Processor | ATmega640 |
| Speed | 16000000 |

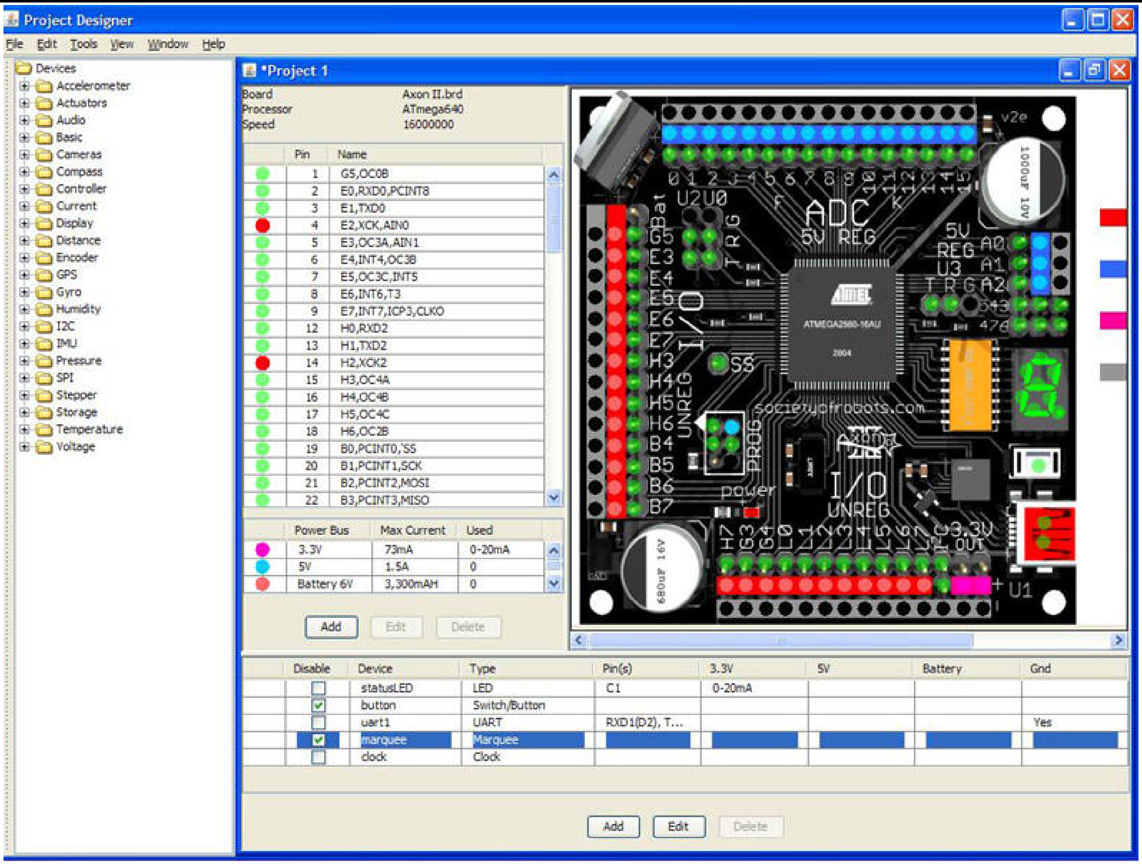
| | Pin | Name |
|---|---|---|
| 🟢 | 1 | G5,OC0B |
| 🟢 | 2 | E0,RXD0,PCINT8 |
| 🟢 | 3 | E1,TXD0 |
| 🔴 | 4 | E2,XCK,AIN0 |
| 🟢 | 5 | E3,OC3A,AIN1 |
| 🟢 | 6 | E4,INT4,OC3B |
| 🟢 | 7 | E5,OC3C,INT5 |
| 🟢 | 8 | E6,INT6,T3 |
| 🟢 | 9 | E7,INT7,ICP3,CLKO |
| 🟢 | 12 | H0,RXD2 |
| 🟢 | 13 | H1,TXD2 |
| 🔴 | 14 | H2,XCK2 |
| 🟢 | 15 | H3,OC4A |
| 🟢 | 16 | H4,OC4B |
| 🟢 | 17 | H5,OC4C |
| 🟢 | 18 | H6,OC2B |
| 🟢 | 19 | B0,PCINT0,'SS |
| 🟢 | 20 | B1,PCINT1,SCK |
| 🟢 | 21 | B2,PCINT2,MOSI |
| 🟢 | 22 | B3,PCINT3,MISO |

| | Power Bus | Max Current | Used |
|---|---|---|---|
| ● | 3.3V | 73mA | 0-20mA |
| ● | 5V | 1.5A | 0-160mA |
| ● | Battery 6V | 3,300mAH | 0-160mA |

| Disable | Device | Type | Pin(s) | 3.3V | 5V | Battery | Gnd |
|---|---|---|---|---|---|---|---|
| ☐ | statusLED | LED | C1 | 0-20mA | | | |
| ☐ | button | Switch/Button | D5 | | | | Yes |
| ☐ | uart1 | UART | RXD1(D2), T... | | | | Yes |
| ☐ | marquee | Marquee | D6, C0, C2, ... | | 0-160mA | | |
| ☐ | clock | Clock | | | | | |

Note that its good practise to disable the devices you aren't using as the generated code will be smaller. No point making the code more complex than we need - right?

You can do this by ticking the box against the device in the list at the bottom of the screen. All devices can be disabled, except for the clock, and this is much easier than deleting things as we may want to put them back later. Note that you can only delete devices that you've added - you can't delete devices that are built onto the board but you can disable them. We won't be using the push button or the marquee (the segmented LED) in this project so we will disable them:-



| | |
|---|---|
| Board | Axon II.brd |
| Processor | ATmega640 |
| Speed | 16000000 |

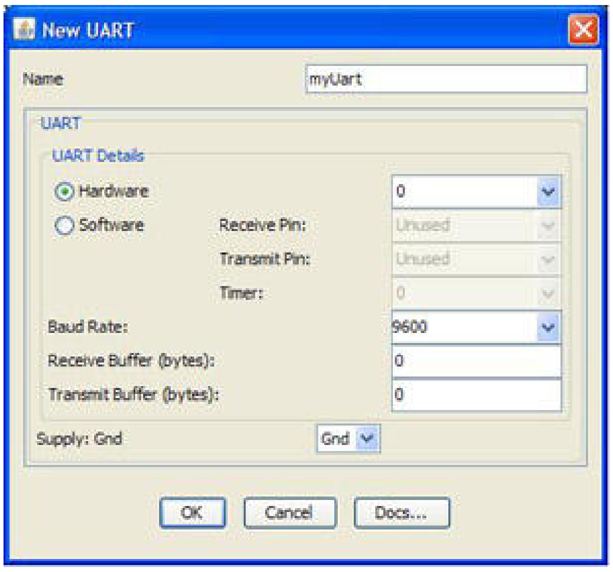
| | Pin | Name |
|---|---|---|
| 🟢 | 1 | G5,OC0B |
| 🟢 | 2 | E0,RXD0,PCINT8 |
| 🟢 | 3 | E1,TXD0 |
| 🔴 | 4 | E2,XCK,AIN0 |
| 🟢 | 5 | E3,OC3A,AIN1 |
| 🟢 | 6 | E4,INT4,OC3B |
| 🟢 | 7 | E5,OC3C,INT5 |
| 🟢 | 8 | E6,INT6,T3 |
| 🟢 | 9 | E7,INT7,ICP3,CLKO |
| 🟢 | 12 | H0,RXD2 |
| 🟢 | 13 | H1,TXD2 |
| 🔴 | 14 | H2,XCK2 |
| 🟢 | 15 | H3,OC4A |
| 🟢 | 16 | H4,OC4B |
| 🟢 | 17 | H5,OC4C |
| 🟢 | 18 | H6,OC2B |
| 🟢 | 19 | B0,PCINT0,'SS |
| 🟢 | 20 | B1,PCINT1,SCK |
| 🟢 | 21 | B2,PCINT2,MOSI |
| 🟢 | 22 | B3,PCINT3,MISO |

| | Power Bus | Max Current | Used |
|---|---|---|---|
| ● | 3.3V | 73mA | 0-20mA |
| ● | 5V | 1.5A | 0 |
| ● | Battery 6V | 3,300mAH | 0 |

| | Disable | Device | Type | Pin(s) | 3.3V | 5V | Battery | Gnd |
|---|---|---|---|---|---|---|---|---|
| | ☐ | statusLED | LED | C1 | 0-20mA | | | |
| | ✓ | button | Switch/Button | | | | | |
| | ☐ | uart1 | UART | RXD1(D2), T... | | | | Yes |
| | ✓ | marquee | Marquee | | | | | |
| | ☐ | clock | Clock | | | | | |

Next we will look at the settings of the UART. In my case I will be using 'uart1' as on the AxonII this has a USB connector making it much easier to connect to my PC.

If no UART is listed for your board then you need to create one. You can do this by opening the 'Communications' folder on the left hand side and then drag the 'UART' entry into your list of devices and you will see a window like this:-

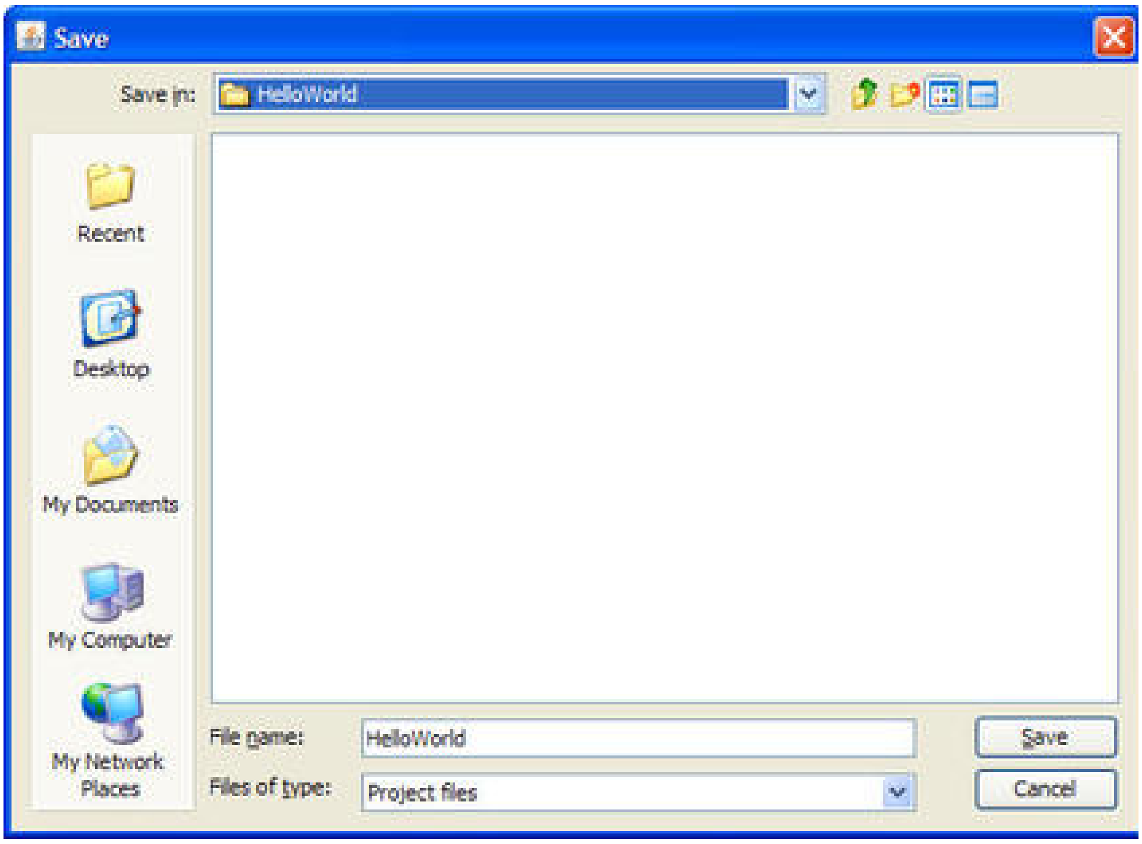


| | |
|---|---|
| Name | myUart |
| (●) Hardware | 0 |
| ( ) Software Receive Pin: | Unused |
| Transmit Pin: | Unused |
| Timer: | 0 |
| Baud Rate: | 9600 |
| Receive Buffer (bytes): | 0 |
| Transmit Buffer (bytes): | 0 |
| Supply: Gnd | Gnd |

If you are using an existing UART then just double click on it in the list of used devices to see the same window. So whether you are using an existing UART or you have created a new one then make a note of the baud rate as we will need that later when setting up the PC end of things.

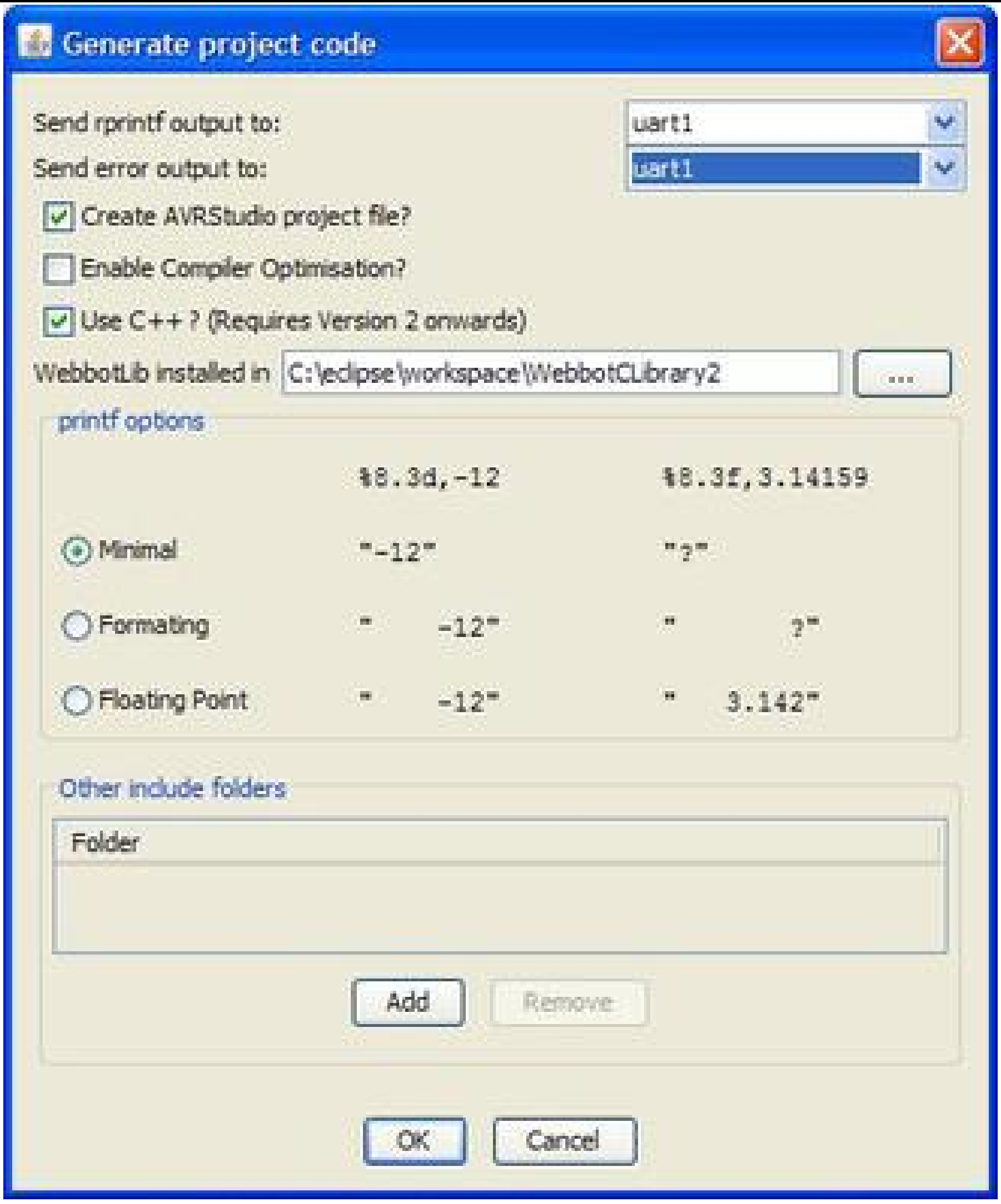
You may be wondering why all of the UARTs don't already come as built in devices. This is because on most micro-controllers each individual pin can be used for many different purposes. So if you are running out of I/O pins but aren't using all of the UARTs then you may want to use the UART pins as standard I/O pins instead. The only reason why things like UART1 on the Axon series are built in is because they've got extra hardware and a USB connector attached to them - so WebbotLib stops you from trying to use them for other things as that may risk blowing up this additional hardware

We have now got the devices we need set up so the only thing left to do in Project Designer is to save the project and generate the code. Select Tools Generate... from the menu. Project Designer will then ask you where you want to save the project. You should store each project in a new directory. In the 'Save' window create a new directory somewhere and then give the project a name in the 'File name:' field. For example: I have created a new folder called 'HelloWorld' and I have named my project as 'HelloWorld':-



File name: HelloWorld
Files of type: Project files

Once the project has been saved you will be prompted to enter some information for the code generation step:-


Generate project code

Send rprintf output to: uart1

Send error output to: uart1

[✓] Create AVRStudio project file?

[ ] Enable Compiler Optimisation?

[✓] Use C++ ? (Requires Version 2 onwards)

WebbotLib installed in C:\eclipse\workspace\WebbotCLibrary2

printf options

|  | %8.3d,-12 | %8.3f,3.14159 |
|---|---|---|
| [●] Minimal | "-12" | "?" |
| ( ) Formating | "     -12" | "        ?" |
| ( ) Floating Point | "     -12" | "   3.142" |

Other include folders

| Folder |
|---|
|  |

The first field allows us to select from the drop down where we want any text output to be sent to by default. Since we are going to be outputting the text "Hello World" then we select the UART we want to send the text out of.

The second field allows us to select where WebbotLib will send any error messages to. We can leave this field blank if we want - but we may as well send them out to the PC over the UART as well.

The first tick box indicates whether or not you want an AVRStudio project file to be created or not. This box may be greyed out if you don't have AVRStudio installed or if there is already an AVRStudio file in the project folder.

The second tick box dictates whether your code will be compiled with optimisation. Leave it un-checked for now.

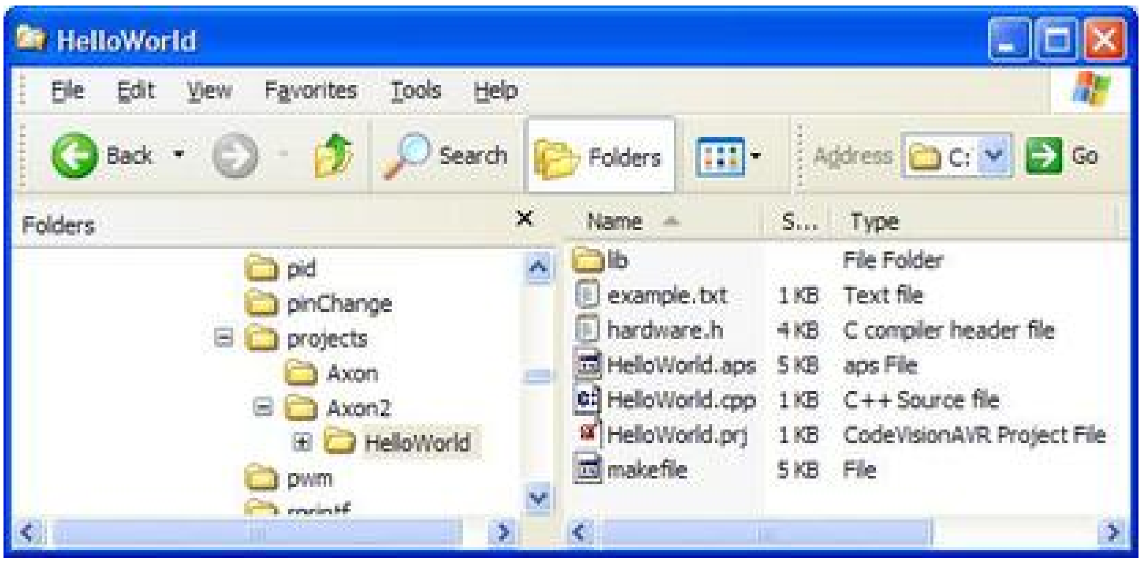
The third tick box allows us to choose between using C and C++. Tick the box to use C++.

The final field allows you to select the folder where WebbotLib has been installed. The code generation process uses this to detect whether you are using Version 1 or Version 2 of WebbotLib as well as adding the correct WebbotLib folder to the build process we will be doing next. Since this document is about WebbotLib Version 2 then I have selected a Version 2 folder.

The list box allows you to add in extra folders for locating other .H files. We wont be using it in this example so just click on the Ok button.

Project Designer remembers all of these settings so if you have to repeat the code generation process then it will show the settings you specified last time.

Use Windows Explorer to have a look at the folder where you saved the project. You will now see lots of files. This is what I can see in my HelloWorld folder:-



| Name | S... | Type |
|---|---|---|
| lib | | File Folder |
| example.txt | 1 KB | Text file |
| hardware.h | 4 KB | C compiler header file |
| HelloWorld.aps | 5 KB | aps File |
| HelloWorld.cpp | 1 KB | C++ Source file |
| HelloWorld.prj | 1 KB | CodeVisionAVR Project File |
| makefile | 5 KB | File |

HelloWorld.prj - is the Project Designer project file

example.txt - this contains some example code. Initially the contents are identical to <Project>.cpp which in this case is HelloWorld.cpp. Why? Well we are going to modify HelloWorld.cpp to make it do what we want our project to do and if we did another code generation then we don't want Project Designer to delete all of the code we have typed in! But if we were to add extra devices then example.txt is a useful file to look at to see some example code of how to use that new device.

hardware.h - contains code to initialise your devices. Never change it by hand! Otherwise all of your changes will be lost next time you do a code generation.

HelloWorld.cpp - this is where all of your code will be added. We will do this in a minute.

makefile - This is a standard makefile to allow you to build your project on Windows, Mac or Unix. Again - you should never change this.

HelloWorld.aps - this is the AVRStudio project file. It is only created if you ticked the box during the code generation process.

You will also see a sub-folder called 'lib'. This contains a number of files that augment the standard library for your project. You should never change any of these. Just forget about them!

## Add your own code

We have now created a project and we can compile it.

You can do this in a number of ways.

1. If you are using AVRStudio then you can start it up and load the HelloWorld.aps AVRStudio file. and build the project.
2. Alternatively you can open a command prompt window and navigate to your HelloWord folder and just type in: make

The program should compile with no errors.

But wait - nowhere have we told anything that it should be writing the phrase "Hello World" over the UART.

To do this we will open up the main file (HelloWorld.cpp). It looks like this:-

```
#include "hardware.h"

// Initialise the hardware
void appInitHardware(void) {
    initHardware();
}

// Initialise the software
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    return 0;
}

// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart) {
    return 0;
}
```

So what's all of that ?

When your program starts it first calls the 'appInitHardware' function. This in turn calls 'initHardware' which is in the generated 'hardware.h'. This initialises all of the devices you are using - so don't worry about it.

It then calls 'appInitSoftware' once - and this is where you can add any 'one-off' code. But we will leave it unchanged. The loopStart parameter parameter is the number of micro-seconds since the power switch was turned on.

Finally 'appControl' is called repeatedly - it is your main loop that is called over and over. So this is where we are going to output our message to the PC. You will see that two values are passed in: loopCount and loopStart. Although we won't be using them here then its worth mentioning what they are. 'loopCount' is just a number that increases by one every time and 'loopStart' is the current time-of-day ie the number of micro-seconds since the power switch was turned on.

You will also see that 'appControl' can return a value. This represents the frequency at which you want your appControl to be called - in microseconds. So returning a value of 1,000,000 will mean that your appControl will be called every 1,000,000µs (ie 1 second) whereas returning a value of 0 makes everything run with no delay.

So WebbotLib spends some of its time running your program (appControl) and another lump of time just 'hanging around' before calling your code again. Although not yet implemented by the library this means that we can calculate the amount of time each iteration through 'appControl' actually takes and so you could implement a 'CPU utilisation' graph similar to that in the Windows Task Manager.

Let's revisit what we want our program to do: "continually print 'Hello World' every second".

Well since its going to happen continually, rather than just once, then we are going to change the 'appControl' code as that's the only one that is called repeatedly.

During the code generation process we told Project Designer to set up the default output (stdout) to go via the UART - so that piece, along with the baud rate, has already been done for us. All we have to do is add the line:-

```
PRINTF(stdout,"Hello World\n");
```

into 'appControl' - the '\n' just adds a 'newline' at the end - so that each message starts on a new line on the PC.

The only other thing we have to do is make 'appControl' return a value of 1,000,000 so that it only happens every 1 second.

So let's see what our new version of appControl looks like:-

```
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart) {
    PRINTF(stdout,"Hello World\n");
    return 1000000;
}
```

Save the file and re-compile using 'make' or 'AVRStudio' as described above.

You now have a file in your HelloWorld folder called HelloWorld.hex which can be flashed to the board. (I can't cover how to do that here as it depends on your board as well as the hardware programmer you are using).

## Testing the program

First of all we need to hook the board up to your PC. If you have used a board where the UART is a USB connector (such as uart 1 on the Axon or AxonII) then you can just use a USB cable.

However: if you are using a board, like the $50 Robot, then the UART cannot just be plugged into a RS232 port on the PC as the voltage levels are different and would blow up your micro-controller. You will need to use a piece of hardware called a level shifter, also known as TTL-RS232 convertor. You've been warned !

Once connected then run your favourite terminal emulation program such as HyperTerminal on Windows and set it up to use the same baud rate as the UART you selected in Project Designer.

Now turn on the board and you see the message "Hello World" displayed every second.

## Tweak it

Let's now add the message "Start Up Complete" which prints out once when the board is first powered up.

We can't add this to appControl because that is called repeatedly. We can't add it into appInitHardware as that's where our hardware gets initialised - so the UART may not be up and running yet! The answer is to put it into appInitSoftware as that is only called once - after the hardware has been initialised.

```
// Initialise the software
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    PRINTF(stdout, "Start Up Complete\n");
    return 0;
}
```

Compile it, upload it and test it.

## FILEs and Streams

Back in the days of C the concept of a FILE was conceived. A FILE was a variable that could be used to either read data from and/or send data to a file on a disk or to a console (ie screen and keyboard). Where a 'read' would read typed characters from the keyboard and a 'write' would write characters to the screen.

For those familiar with WebbotLib then a FILE is the combination of a Reader and/or a Writer.

C defines three FILEs by default:

1. stdout - a FILE that is used to write output to the standard output device (normally some kind of screen)
2. stdin - a FILE that is used to read input from the standard input device (normally some kind of keyboard)
3. stderr - a FILE that is used to write error messages to the standard error device. This is typically a log file on disk or may be a screen. It can be the same destination as stdout.

These FILE variables can then be used by the commands in stdio.h (which comes with your compiler) such as: fgetc, fputs, fprintf etc.

WebbotLib V2 allows you to use all of these commands and also adds the macro "*PRINTF(stream,format, args...)*" (see page 270) which places the format string in program memory to save space. This method is now preferred over the 'old' rprintf routines.

When generating your code from Project Designer you specify the device where the default output goes, and the device where any error messages go. Project Designer then sets 'stdout' and 'stdin' to the first device and 'stderr' to go to the second.

When C++ came along the concept of a FILE was extended into a class called a 'Stream'. The stream versions of stdout, stdin and stderr are called cout, cin and cerr respectively. WebbotLib provides support for streams via the class "*Stream*" (see page 365) .

This means you can print something to the standard out, in C++, using for example:

```
cout.print(12);
cout.print(1234);
cout.print("Hello World");
```

You will see how the 'print' function automatically copes with different types of data and, unlike rprintf or PRINTF, you don't need to remember the correct format string for the type of variable you are printing.

You can also keep calling the print routine on the same line ie:

```
cout.print("The answer to 4*8 = ").print(4*8).print('\n');
```

You may also use the '<<' operator to write data to a stream. So you could also write the previous code as:

```
cout << "The answer to 4*8 = " << 4*8 << '\n';
```

The choice is up to you. However: if you want to minimise the amount of RAM used by your program then you should place any fixed strings into program memory by using PSTR and then use the print_P function to print from program memory:

```
cout.print_P( PSTR("The answer to 4*8 =") ).print(4*8).print('\n');
```

There is no equivalent way of doing this using the '<<' operator.

As well as the cin, cout and cerr streams there are various things you can add to your project in Project Designer which also support streams. For example: uarts, displays and disk files on an sdCard. If you have added a uart in Project Designer called 'uart3' then you can send data to it in exactly the same way i.e.:

```
uart3.print("The answer to 4*8 = ").print(4*8).print('\n');
```

## Building the library itself

WebbotLib comes to you with all of the code pre-compiled into libraries meaning that it is ready-to-go for your own projects.

Advanced users may want to add their own code into WebbotLib or fix a minor bug (and let me know what they fixed!). If you just change the C code within the library and rebuild your project you will find that nothing has changed. Why? Well your project is using my pre-built libraries and not the source code.

So if you make any changes inside the library then you will need to recompile the whole library so that your changes get into the pre-compiled libraries used by your projects.

Its easy enough to do.....

### Changing an existing file

If you have just modified an existing piece of source code then you can rebuild the libraries by opening a command window, and moving to the WebbotLib folder. Then just type:

```
make all
```

or if you want to see all of the options then just type:

```
make
```

If you have fixed a bug then let me know about it - otherwise when you get the next release your change will have disappeared.

### Adding new files to the library

If you have added your own code to the library then you need to let 'make' know about it. Open the 'makeone' file in a text editor.

You will see 3 large assignments here: C_SRCS, OBJS and C_DEPS. Just add an entry into each of the three for your new file.

You can now rebuild the library using:

```
make all
```

### But Beware...support issues

I would recommend that before you make any changes or additions that you do so by copying the whole WebbotLib folder to a new folder and then make your changes there. You then have a 'supplied' version of the library to resort back to.

Obviously if you ask support questions to do with WebbotLib but you have altered the supplied code then it makes life difficult for everyone! If this is the case then its best to make it clear what changes you have made and why.

## Extending WebbotLib

If you have added a new feature to WebbotLib and feel that it merits being included then let me know (webbot@webbot.org.uk). This could be support for a new sensor, a bug fix, or just a 'handy piece of code'. I will appraise what you have done and either include it (with acknowledgement to you) or I may just say 'its cool - but.....'.

When submitting new stuff then remember the ethos of the library - it supports a large range of ATMega chips (not just one!) - so any chip/board/project specific stuff that would only make life easier for your own project may be dismissed! Also any new sensors need to return standard values:- distance sensors return 'cm' etc - so make sure that yours does too.

# Project Designer

This section describes the different devices that can be added using Project Designer.

Project Designer provides the following device categories:-

## 1 Wire

Adds support for Dallas One Wire devices.

So called because only one digital I/O pin is used to talk to all of the devices on the bus. In reality you must also connect the Ground from your microcontroller to each device as well.

Many of these devices only need these two wires and power themselves from the same bus itself (called 'parasitic mode'). If you are using parasitic mode then you should add up the total amount of current required by all the devices on the same bus and make sure that they don't require more current than your microprocessor can provide from an output pin (normally about 30mA).

If in doubt, or if the power requirement is too big, then you will need to use a third wire to supply regulated power to the devices (typically a 3v to 5.5v supply).

The following devices are supported:

## One Wire Bus

Creates the one wire bus to which individual devices can be added.

A one wire bus provides the following functions (but you should only need them if you are trying to support a device that isn't directly supported by WebbotLib):

| OneWireBus - Function Summary | |
|---|---|
| | applyPower<br>Often referred to in the specifications as 'strong pull-up' this will drive the signal line high so that parasitic devices can obtain power from the bus. |
| boolean | readBit<br>Reads a single bit from the bus and returns TRUE for 1, or FALSE for 0. |
| | writeBit<br>Write a single bit to the bus. |
| | setStandardSpeed<br>Sets the bus to use 'standard' speed. |
| | setOverdriveSpeed<br>Sets the bus to use the faster 'overdrive' speed. |
| boolean | reset<br>Resets the one wire bus and returns the 'presence' bit. |
| | write<br>Writes an 8 bit byte to the bus. |
| uint8_t | read<br>Reads an 8 bit byte from the bus. |
| uint16_t | list<br>Dumps out the ROM IDs of every device that is connected to the bus and returns the total number of devices found. |
| | dumpROM<br>Dumps the ROM ID of the last device found by a call to 'first' or 'next'. |
| boolean | first<br>Find the first device connected to the bus. |

## OneWireBus - Function Summary

| | |
|---|---|
| boolean | next<br>Find the next device connected to the bus. |
| uint8_t * | getROM<br>Return the 8 byte ROM ID of the device matched by the last call to first or last. |

# Accelerometer

Supports accelerometers.

Accelerometers normally come in either 2 axis, or 3 axis versions. Sometimes you can buy a combo board that contains an accelerometer and, say, a compass. I don't directly support these combos so you need to declare the two individual sensors.

## What Are They?

So what do they do? Accelerometers measure acceleration along each of their axes. This could be used to monitor vehicle acceleration, vibrations, and any other kind of movement - after all a movement requires an acceleration!

The axes are normally called X, Y and Z. A two axis device normally provides X and Y, whereas a 3 axis device provides X, Y and Z. Most devices have a small circle on the package to indicate the X axis.

However the actual direction of the axes will depend on how you mount the device onto your robot. The convention is that X axis should be the forward direction of travel, the Y axis is to the right of that, and Z is down. (I think thats correct!)

These devices either use one ADC pin for each axis, or provide an I2C interface but you will need to check the individual data sheets to see what power supply they need. Be careful some of them require a 3.3v supply so connecting them to the usual 5v will fry them!

## What Do They Return?

All accelerometers return one value per axis and this value is in 'mG' ie thousands-of-a-G where G is the gravitational constant of 9.8m/s/s. Each value may be positive or negative and a two axis device will set the Z value to zero.

If you decide to swap an accelerometer for a different one then, in Project Designer, delete or disable the old one and then add the new one with the same name and then regenerate your code. Your application code should not need changing at all.

## Manual Calibration

Please be aware that the manufacturing tolerances of these devices is not very good. This means that two similar devices may output wildly different values. If you want to achieve a more accurate measurement then you must calibrate the software to match your device. For an explanation of how to do this see "*calibrate*" (see page 422) .

All accelerometers provide the following functions:

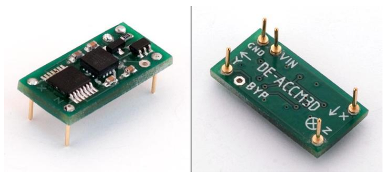
## Function Summary

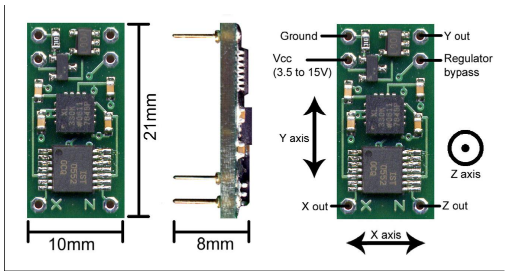
| | |
|---|---|
| | calibrateX<br>Manually calibrate the X axis. |
| | calibrateY<br>Manually calibrate the Y axis. |
| | calibrateZ<br>Manually calibrate the Z axis. |
| | calibrate<br>Manually calibrate the X, Y and Z axes all in one go. |
| ACCEL_TYPE | getX<br>Returns the X axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getY<br>Returns the Y axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getZ<br>Returns the Z axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |

The following devices are supported:

## ACCM3D2 3axis

The ACCM3D2 is a 3 axis accelerometer from Dimension Engineering. See http://www.dimensionengineering.com/DE-ACCM3D2.htm





This library assumes that the device is being powered by a 3.5V to 15V supply via the on-board 3.3V regulator.

### Example

Assuming you have called the device 'myAccel' in Project Designer then add the following code to your 'appControl' function:-

```
// Read the accelerometer and store results
myAccel.read();

// Retrieve the x axis value in mG
ACCEL_TYPE x = myAccel.getX();
// Retrieve the x axis value in mG
ACCEL_TYPE y = myAccel.getY();
// Retrieve the x axis value in mG
ACCEL_TYPE z = myAccel.getZ();

// Each value can be printed to stdout:
cout << "X=" << x << " Y=" << y << " Z=" << z;

// or all values can be dumped to stdout out using
cout << myAccel;
```

The values returned are in 'mG' ie 1000ths of the gravitational constant.

All accelerometers provide the following functions:

| Accelerometer - Function Summary | |
|---|---|
| | calibrateX<br>Manually calibrate the X axis. |
| | calibrateY<br>Manually calibrate the Y axis. |
| | calibrateZ<br>Manually calibrate the Z axis. |
| | calibrate<br>Manually calibrate the X, Y and Z axes all in one go. |
| ACCEL_TYPE | getX<br>Returns the X axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getY<br>Returns the Y axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getZ<br>Returns the Z axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## ADXL335 3 Axis Accelerometer

The ADXL335 is a 3 axis accelerometer from Analog Devices capable of measuring ±3g.

Data sheet: http://www.sparkfun.com/datasheets/Components/SMD/adxl335.pdf

Supplier: http://www.sparkfun.com/commerce/product_info.php?products_id=9268

This library assumes that the device is using the default 3 volt supply.

### Example

Assuming you have called the device 'myAccel' in Project Designer then add the following code to your 'appControl' function:-

```
// Read the accelerometer and store results
myAccel.read();

// Retrieve the x axis value in mG
ACCEL_TYPE x = myAccel.getX();
// Retrieve the x axis value in mG
ACCEL_TYPE y = myAccel.getY();
// Retrieve the x axis value in mG
ACCEL_TYPE z = myAccel.getZ();

// Each value can be printed to stdout:
cout << "X=" << x << " Y=" << y << " Z=" << z;

// or all values can be dumped to stdout out using
cout << myAccel;
```

The values returned are in 'mG' ie 1000ths of the gravitational constant.

All accelerometers provide the following functions:

| Accelerometer - Function Summary | |
|---|---|
| | calibrateX<br>Manually calibrate the X axis. |
| | calibrateY<br>Manually calibrate the Y axis. |
| | calibrateZ<br>Manually calibrate the Z axis. |
| | calibrate<br>Manually calibrate the X, Y and Z axes all in one go. |

## Accelerometer - Function Summary

| | |
|---|---|
| ACCEL_TYPE | getX<br>Returns the X axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getY<br>Returns the Y axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getZ<br>Returns the Z axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |

All sensors provide the following functions

## Sensor - Function Summary

| | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## ADXL345 3 Axis Accelerometer

The ADXL345 is a 3 axis accelerometer from Analog Devices capable of measuring ±16g.

Data sheet: http://www.sparkfun.com/datasheets/Sensors/Accelerometer/ADXL345.pdf

This library supports the device using I2C so pin 7 (CS) should be connected to pin 1 (Vdd) and pin 12 (SDO) can be used to select the I2C address. If pin 12 is connected to ground then the I2C address is 0xA6 and if pin 12 is connected to Vdd then the address is 0x3A. Project Designer will show you how to connect the pins.

### Example

Assuming you have called the device 'myAccel' in Project Designer then add the following code to your 'appControl' function:-

```
// Read the accelerometer and store results
myAccel.read();

// Retrieve the x axis value in mG
ACCEL_TYPE x = myAccel.getX();
// Retrieve the x axis value in mG
ACCEL_TYPE y = myAccel.getY();
// Retrieve the x axis value in mG
ACCEL_TYPE z = myAccel.getZ();

// Each value can be printed to stdout:
cout << "X=" << x << " Y=" << y << " Z=" << z;

// or all values can be dumped to stdout out using
cout << myAccel;
```

The values returned are in 'mG' ie 1000ths of the gravitational constant.

The ADXL345 provides the following functions:

| Function Summary | |
|---|---|
| | refresh25Hz<br>Configures the device to output new values 25 times per second. |
| | refresh50Hz<br>Configures the device to output new values 50 times per second. |
| | refresh100Hz<br>Configures the device to output new values 100 times per |

| Function Summary | |
|---|---|
| | second. |
| | refresh200Hz<br>Configures the device to output new values 200 times per second. |
| | refresh400Hz<br>Configures the device to output new values 400 times per second. |
| | refresh800Hz<br>Configures the device to output new values 800 times per second. |
| | refresh1600Hz<br>Configures the device to output new values 1,600 times per second. |
| | refresh3200Hz<br>Configures the device to output new values 3,200 times per second. |

All accelerometers provide the following functions:

| Accelerometer - Function Summary | |
|---|---|
| | calibrateX<br>Manually calibrate the X axis. |
| | calibrateY<br>Manually calibrate the Y axis. |
| | calibrateZ<br>Manually calibrate the Z axis. |
| | calibrate<br>Manually calibrate the X, Y and Z axes all in one go. |
| ACCEL_TYPE | getX<br>Returns the X axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getY<br>Returns the Y axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |

| Accelerometer - Function Summary | |
|---|---|
| ACCEL_TYPE | getZ<br>Returns the Z axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |

All sensors provide the following functions

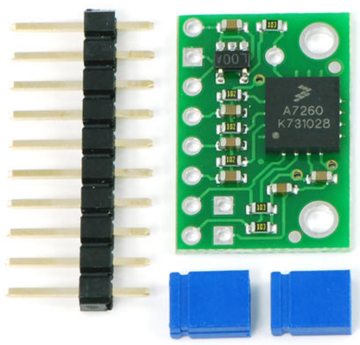
| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## MMA7260QT 3 axis (max g)

The MMA7260QT is a 3 axis accelerometer from Freescale capable of measuring up to ±6g in four different sensitivity ranges.

Data sheet: http://www.pololu.com/file/download/MMA7260QT.pdf?file_id=0J87

Supplier: http://www.pololu.com/catalog/product/766



### Note

Project Designer has several entries for this sensor depending upon the maximum acceleration you need to measure. Measuring larger 'g' values is done with less accuracy. Project Designer will show you how to set the jumpers for the 'g' range that you require.

### Example

Assuming you have called the device 'myAccel' in Project Designer then add the following code to your 'appControl' function:-

```
// Read the accelerometer and store results
myAccel.read();

// Retrieve the x axis value in mG
ACCEL_TYPE x = myAccel.getX();
// Retrieve the x axis value in mG
ACCEL_TYPE y = myAccel.getY();
// Retrieve the x axis value in mG
ACCEL_TYPE z = myAccel.getZ();

// Each value can be printed to stdout:
cout << "X=" << x << " Y=" << y << " Z=" << z;

// or all values can be dumped to stdout out using
cout << myAccel;
```

The values returned are in 'mG' ie 1000ths of the gravitational constant.

All accelerometers provide the following functions:

| Accelerometer - Function Summary | |
|---|---|
| | calibrateX<br>Manually calibrate the X axis. |
| | calibrateY<br>Manually calibrate the Y axis. |
| | calibrateZ<br>Manually calibrate the Z axis. |
| | calibrate<br>Manually calibrate the X, Y and Z axes all in one go. |
| ACCEL_TYPE | getX<br>Returns the X axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getY<br>Returns the Y axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getZ<br>Returns the Z axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Actuators

Adds support for servos and DC motors.

This library tries to standardise on how we deal with these devices so that we can swap between different devices with minimal code changes.

### Drive Speed For Motors or Continuous Rotation Servos

The drive speed is an abstract value that represents how fast, and in what direction, we want a motor to turn.

A drive speed of 0, or DRIVE_SPEED_BRAKE, means 'stop and apply the brakes'.

DRIVE_SPEED_MIN means go full speed in reverse and DRIVE_SPEED_MAX means go full speed forward.

Intermediate values will also work. ie 'DRIVE_SPEED_MAX / 2' means go at half of full speed forward.

Obviously the actual speed at which the robot moves will depend on the abilities of the actual motor/servo as well as the wheel diameter.

### Drive Speed For Servos

For a servo a drive speed of 0, or DRIVE_SPEED_CENTER, means move to the centre position.

DRIVE_SPEED_MIN means move fully in one direction and DRIVE_SPEED_MAX means move fully in the other direction.

### Connection Status

Actuators can also be 'disconnected' ie they are no longer sent any commands until they are 'reconnected'. Disconnecting will mean that the motors will free wheel to a stop. If the robot is on a slope then it may well start accelerating down the slope. NB This is different from setting the drive speed to 0 (or using the constant DRIVE_SPEED_BRAKE) which will cause the motor to brake - which will then stop the robot from rolling down the slope. Obviously if the motors are unable to hold the robot then it may skid down the slope.

Disconnecting a servo means that it will no longer be sent a signal and so all torque will be lost. Disconnecting the leg servos on your biped will mean that it will just crumple to the floor under its own weight.

### Inverted

We also try to address the common problem of differential drive robots. Since the motors are on the sides of the robot then setting them both to turn clockwise at full speed will just cause the robot to spin on the spot. You need to make one turn clockwise and the other turn counter clockwise in order to go in a straight line. This often makes the code rather less obvious. We address this issue by having an 'inverted' state for each motor. That way we can set one motor as inverted and we can then just tell the motors to go full speed ahead and the library takes care of the rest.

### Motor and Servo controller boards

Motors, due to their current requirements, are normally controlled via a separate board whereas servos can be connected directly to your controller pins or they may also be connected via a separate board.

These external boards may be connected via I2C, UARTs etc - each with their own different protocol. By using WebbotLib you don't need to know the details of how these boards work. You just call one function to set the required speed of the actuator and WebbotLib takes care of the rest by sending whatever messages it needs to send to get the job done.

### Example

Since it doesn't make any difference to your own project code quite how your actuators are being controlled then its easy to come up with a generic example that will always work!

Assuming that you have created two motors/servos in Project Designer called 'left' and 'right' respectively then the following code should make them repeatedly go forward and backward. Project Designer will generate this as part of its examples.

```
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart) {

    // Get current time in ms
    TICK_COUNT ms = loopStart / 1000;
    int16_t now = ms % (TICK_COUNT)10000; // 10 sec for a full swing
    if(now >= (int16_t)5000){ // Goes from 0ms up to 5000ms
        now = (int16_t)10000 - now; // then 5000ms down to 0ms
    }

    // Map the 0 - 5000 value into DRIVE_SPEED range
    DRIVE_SPEED speed = interpolate(now, 0, 5000, DRIVE_SPEED_MIN,
DRIVE_SPEED_MAX);

    // Set speed for all motors/servos
    left.setSpeed(speed);
    right.setSpeed(speed);
    return 0;
}
```

All actuators provide the following functions:

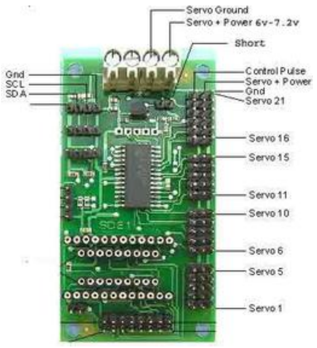
## Function Summary

| | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

The following devices are supported:

## Devantech SD21 Servo Driver

The Devantech SD21 is a servo controller for driving up to 21 servos via I2C address 0xC2.

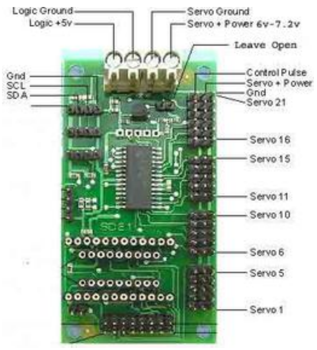


### Single Supply

If you are only using a single battery to power the servos and to power the board then connect the battery to the servo supply terminals as shown and make sure the 2 pin jumper header is shorted together.

### Dual Supply

Alternatively you can power the board logic from a regulated 5V supply and provide a separate unregulated supply to power the servos. In this case the jumper should be left open.



### Example

Just use the generic example in "*Actuators*" (see page 38)

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |

## Actuator - Function Summary

| | |
|---|---|
| `boolean` | isConnected<br>Test if an actuator is connected |
| `boolean` | isInverted<br>Test if an actuator is inverted |

## Dynamixel AX-12 Driver

Adds support for the Dynamixel AX-12 servo.

See http://www.trossenrobotics.com/dynamixel-ax-12-robot-actuator.aspx for more info and a data sheet.

These servos are controlled over a UART in half-duplex mode and allows multiple servos to be controlled via the same UART. You MUST make sure that each servo has been changed to have a unique ID number as described in the manual. Try to make sure that you label each servo with its current ID number as this will save heartaches in the future!

By default these servos communicate at 1 million baud. Yep 1 million. This works fine and you shouldn't need to change it. Note that if you feel you need to change it then you will need to make sure that all of the AX12 servos are using the same baud rate (coz they are on the same bus) and that your micro controller is using the same baud rate. Note: this is not for the faint hearted as its easy to mess up a servo - ie if you don't know what baud rate it has been set to then how can you send it messages to re-program it !

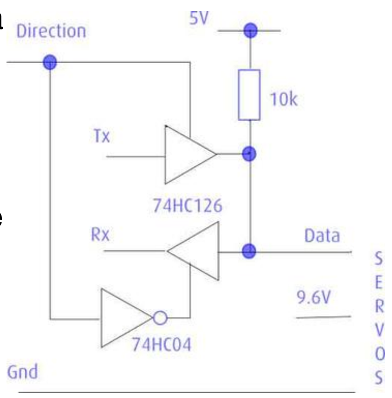
These devices are very impressive but they aren't cheap!! You can use them as a servo (you can even decide what the min/max swing is) or you can put them into continuous rotation mode so that they work like motors. You can even get information, per servo, regarding its current temperature, loading. voltage etc. No wonder they are the servo of choice to the biped market. All very cool!

### Half Duplex

The half duplex mode means that the transmit wire from your UART is connected to all of the AX-12 servos. Hence: when you send a command over the wire then all of the servos will hear it - but because the message contains the destination servo ID then only one servo, matching that ID number, will process it. If the message requires a response then it is sent back down the same wire from the servo back to your UART.

If there is more than one servo with the same ID then they will both try to send their response down the same wire and that is bad - very bad. At best the reply will be garbaged - at worst something may go bang.

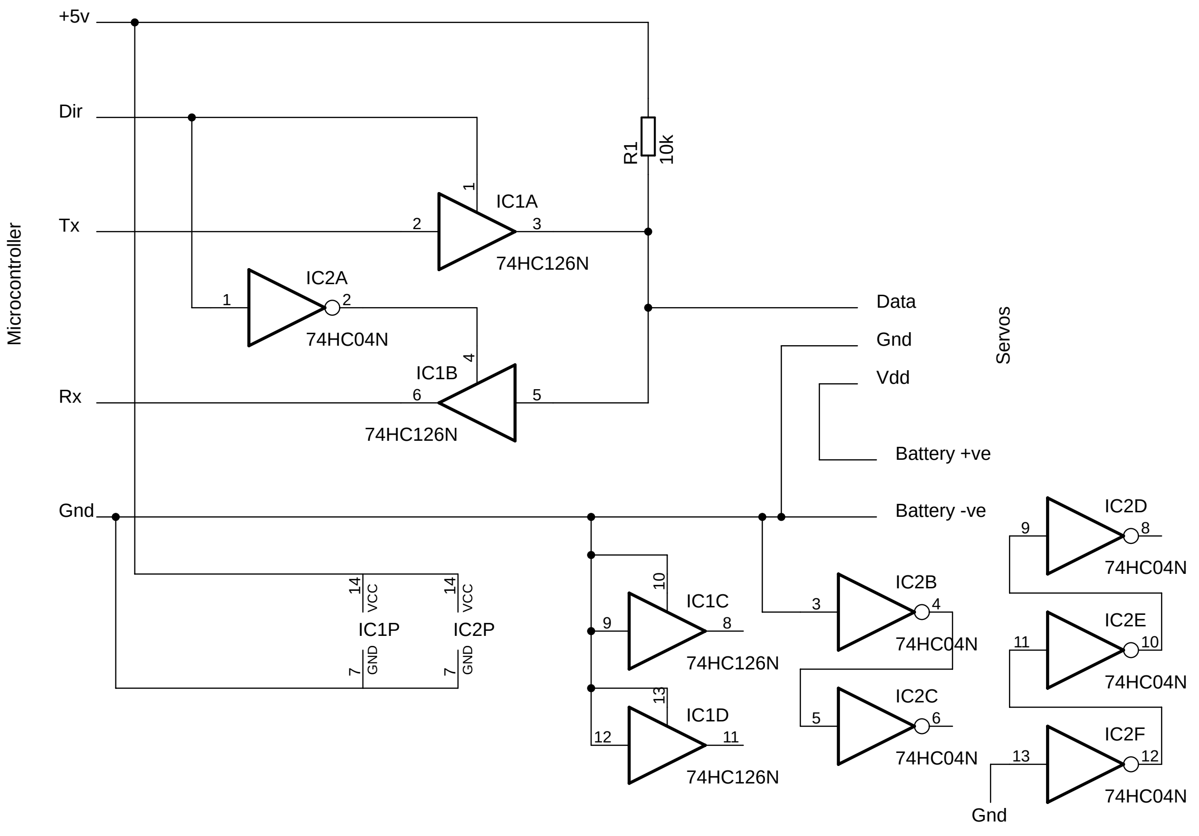
The half-duplex mode means that a single wire is used to transmit data to the servo and is also used to listen for any response. In order to connect this to the UART on your board you will need some additional circuitry like this:



Note that you only need to make one of these circuits regardless of the number of servos you are controlling. The effect of the circuit is to use the 'Direction' I/O line to connect the 'Data' line to the 'Tx' pin of your UART, or to the the 'Rx' pin of your UART - but never to both.

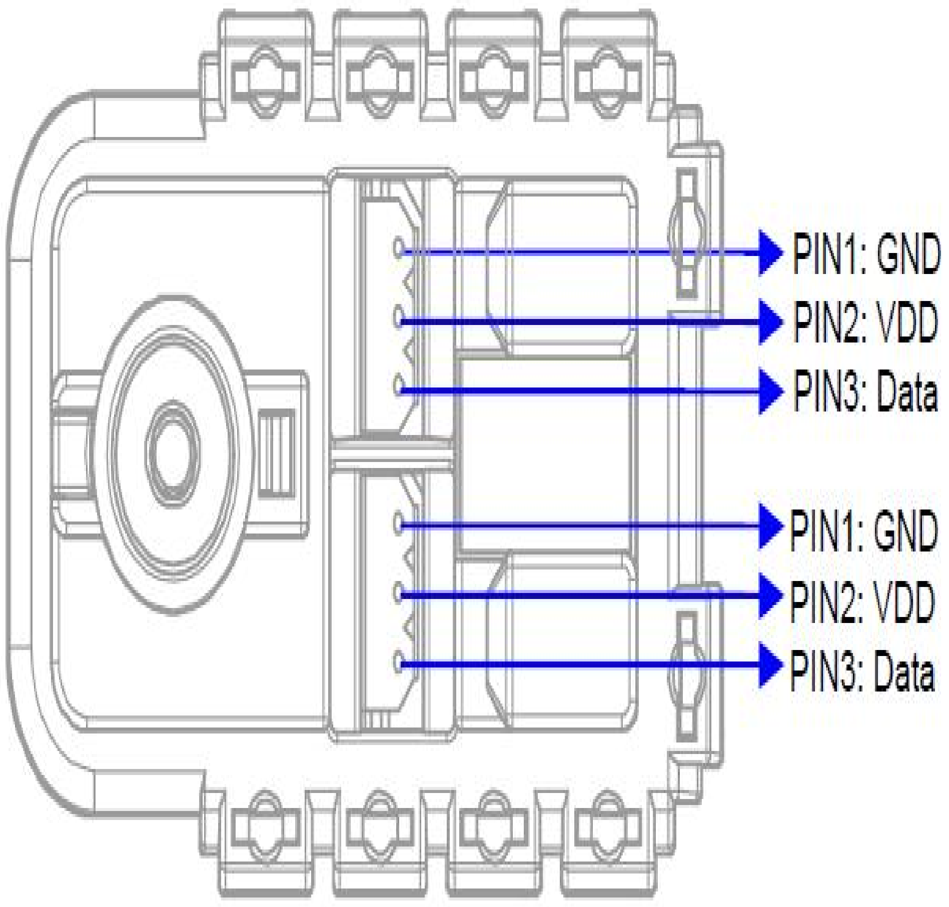
The WebbotLib driver takes care of this for you by setting the direction pin correctly at all times.

Here is a bigger schematic of the circuit. Note that the unused gates in the chips are shown in the bottom right and their inputs are connected to something to stop them noisily flapping around.



I think its possible to replace the hex inverter (of which we only use one gate) with one of the unused gates from the other chip + one resistor. Need to play with that !

Here is a schematic of the back of each servo so you can see how they are attached to the bus,



One group of 3 pins is used to feed the signal from the UART into the servo and the other group of 3 pins is used to output the signals to the next servo. The groups seem to be interchangeable - ie both groups are inputs and outputs.

All Dynamixel AX12 servos support the following functions:

| Function Summary | |
|---|---|
| uint8_t | getID<br>Returns the servo ID number. |
| uint16_t | read<br>Read the current settings from a servo. |
| DRIVE_SPEED | getActualSpeed<br>Following a successful read() this will return the current position or speed of the servo. |
| int8_t | getActualRPM<br>Following a successful read() this will return the current rotational speed in RPM (revolutions per minute). |

| Function Summary | |
|---|---|
| int16_t | getActualLoad<br>Following a successful read() this will return the current load in the range -1023 to +1023. |
| uint8_t | getActualVolts<br>Following a successful read() this will return the voltage being supplied to the servo in 10ths of a volt. |
| uint8_t | getActualTemperature<br>Following a successful read() this will return the temperature of the servo in celsius. |
| boolean | getActualIsMoving<br>Following a successful read() this will return TRUE if the servo is actually moving or FALSE if not. |
| | dump<br>Dump the current status of a given servo. |
| | ledOn<br>Turns on the LED on the servo. |
| | ledOff<br>Turns off the LED on the servo. |

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |

## Actuator - Function Summary

| | |
|---|---|
| `boolean` | isConnected<br>Test if an actuator is connected |
| `boolean` | isInverted<br>Test if an actuator is inverted |

## Generic PWM motor driver

Controls DC motors using pulse width modulation (PWM).

This is a 'catch all' in case your specific motor driver is not specifically listed.

Note that you cannot connect a DC motor directly as they require too much current and you will blow your processor - you must use a motor controller.

This module assumes that your motor controller has three inputs which can be used to achieve the following states:-

Driver actions

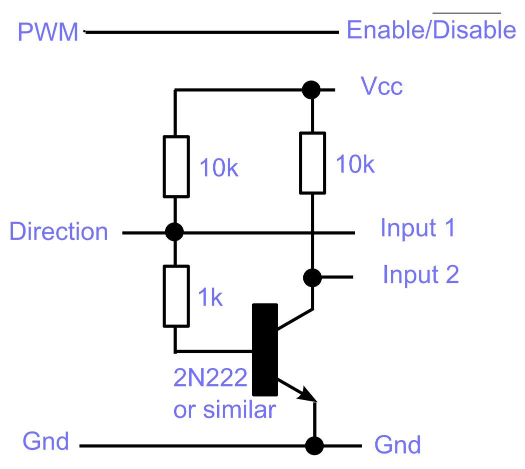
| Enable | Input 1 | Input 2 | Result |
|---|---|---|---|
| Low | Anything | Anything | Coast |
| High | Low | Low | Brake |
| High | Low | High | Forward |
| High | High | Low | Reverse |
| High | High | High | Brake |

### Motor (3 Pin)

Adding a 'Motor (3 Pin)' to the list means that the signals will be generated directly on the selected pins.

### Motor (2 Pin)

If you are short of I/O pins then you can drive each motor with only two I/O pins but you need to add some additional hardware (which I call a 'tri-state switch').



This circuit will take the output of the two pins from your processor and convert it into the three control pins required for the optimum control of your motor driver. The hardware PWM pin is forwarded on to the enable pin of the driver and is used to set the speed of rotation by alternating between 'full speed' and 'coast'. The direction pin is forwarded on to one of the inputs to the driver and is also inverted by the transistor for the second input to the driver.

By making the direction pin into a high impedance input pin then the resistors will guarantee that both of the inputs to the driver have the same value which will result in the brake being applied.

If you want to make your own motor controller board that can be used with this hardware then read http://www.societyofrobots.com/member_tutorials/node/159 for a range of DC and stepper motor boards based on the L293D, L298 or SN754410 which can be driven in this way.

Since we are using hardware PWM then the pulses sent to the motors are very exact. This library can cope with any number of motors and so the only real limitation is the number of PWM channels that your micro controller provides.

## Example

Just use the generic example in "*Actuators*" (see page 38)

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| `DRIVE_SPEED` | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| `boolean` | isConnected<br>Test if an actuator is connected |
| `boolean` | isInverted<br>Test if an actuator is inverted |

## L293D motor driver and SN754410 motor driver

Support for the L293D and SN754410.

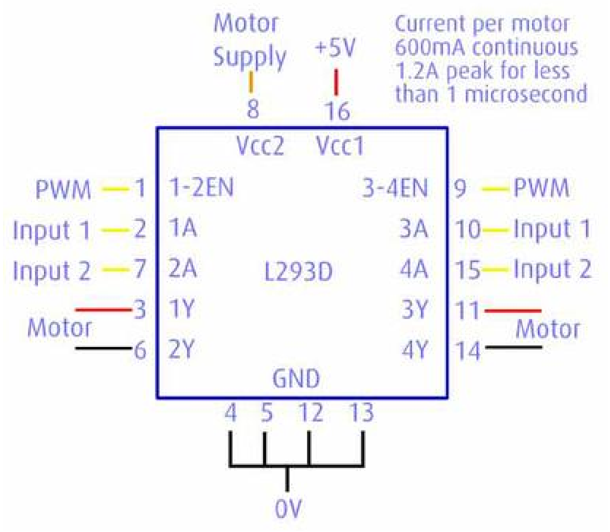
These devices have exactly the same pin-outs and so you can swap between them without any hardware/code changes. The only difference is that SN754410 can supply 1A versus 0.6A for the L293D. Both devices can supply double the current for a fraction of a second.

The SN754410 is a more modern device and so tends to be cheaper to buy. "More current and cheaper" = "No brainer"! If you decide to go with the L293D then make sure you buy one with the 'D' at the end as this has the internal diodes fitted meaning that you don't need to add any external components.

There are two different ways to use each motor: 3 pin mode or 2 pin mode. Each has advantages and disadvantages. If you have lots of available IO pins then my recommendation would be to use '3 Pin Mode'.

If you want then you can use a mixture of both 3 pin and 2 pin modes.

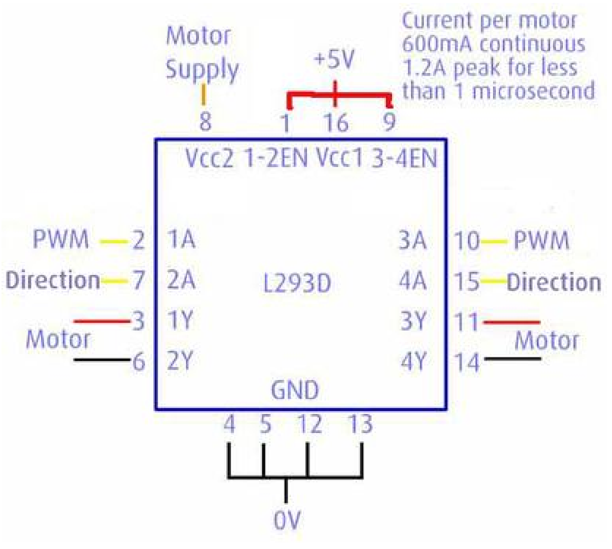
### 3 Pin Mode



This requires one hardware PWM pin plus 2 general purpose digital output pins per motor. The benefit of this mode is that it can support all the possible motor drive states required by WebbotLib ie: forward, reverse, brake and coast. Variable speed is achieved by alternating between 'full speed' and 'coast'.

## 2 Pin Mode

This requires one hardware PWM pin plus 1 general purpose digital output pin. The drawback of this mode is that it doesn't support the 'coast' motor drive state. Consequently to achieve variable speed settings the motor alternates between 'full speed' and 'brake' via the PWM pin. This places the motor under some stress and will 'wear out' the motor more quickly than the 3 Pin Mode.



## Example

Just use the generic example in "*Actuators*" (see page 38)

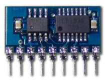
All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Pololu Dual Serial

Adds support for various Pololu motor controllers.

Namely the following:-

-  Micro Dual Serial Motor Controller (SMC02B)
- Low Voltage Dual Serial Motor Controller (SMC05A)
- 3Amp motor controller with feedback (SMC03A)
- High power motor controller with feedback (SMC04B)

These controllers require a UART port where the Gnd and Transmit pins are connected to each board. Each motor has a unique number which can be set by sending it a 'one-off' sequence of characters - see the datasheets. Avoid motor numbers 0 and 1 if you have more than one board since all boards will re-act to these numbers.

### Example

Just use the generic example in "*Actuators*" (see page 38) .

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |

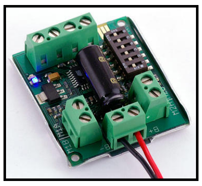
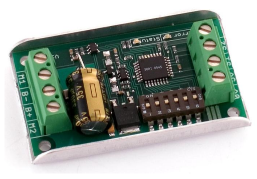
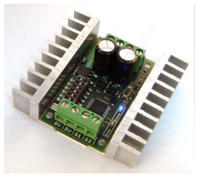
## Actuator - Function Summary

| | |
|---|---|
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Sabertooth Motor Controller

Adds support for various motor controllers from Dimension Engineering.

Namely the Sabertooth 2 x 5 Amp controller, the SyRen10 and SyRen25 boards.

These controllers require a UART port where the Gnd and Transmit pins are connected to each board on the 0V and S1 inputs respectively.

These boards can be driven in a number of different modes but we will use only two of these modes namely:

- Packetized serial (mode 4)
- Simplified serial mode (mode 3)

Each board can drive up to two motors, called motors 1 and 2.

### Simplified Serial Mode

Simplified serial mode can only be used to control one board (so a maximum of 2 motors) - but has the advantage that it only has to send one byte for each motor and the baud rate can be set via the DIP switches to select either 2400, 9600, 19200 or 38400 baud.

### Packetized Serial Mode

Packetized serial mode has the advantage that multiple boards (up to 8 i.e. 16 motors) can be placed on the same serial line - each board has a unique address set via the DIP switches. This combination of address and motor number is therefore unique for each motor. The advantage of this mode is the ability to control multiple boards but the disadvantage is that we need to send 4 bytes for each motor compare with one for simplified serial mode. However: the greatest disadvantage is that packetized mode uses an 'auto baud rate' detection algorithm with valid baud rates of 2400, 9600, 19200 and 38400 baud. This detection is based on the first byte that it is automatically sent by this library. Although this 'sounds' easy the practical issue is that your main controller board 'may' suffer from contact bounce on its power switch - the end result being that one or more spurious characters may be sent over the UART and confuse the Sabertooth board as to what the actual baud rate should be. If you are using separate power supplies for the Sabertooth and your micro-controller board then this also means that you need to 'turn on' the Sabertooth card before, or at the same time, as your micro-controller. When using this mode the library will wait for 2 seconds before sending the automatic baud detection character.

### DIP Switches

When adding the motors in Project Designer take careful note of the DIP switch settings and set up your board accordingly.

## Example

Just use the generic example in "*Actuators*" (see page 38)

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Sanyo LB1836M motor driver

The LB1836M is a rather light weight motor driver and is included on 'old' BabyOrangutan boards.

The device can support two DC motors. Each motor requires two PWM channels and is therefore quite 'hungry' on timers.

Project Designer contains a board design for the BabyOrangutan with this motor controller already set up for you.

### Example

Just use the generic example in "*Actuators*" (see page 38)

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Serial Servo Driver

A generic solution for using various different serial servo controllers.

Currently this driver supports the following serial protocols:-

- Mini SSC
- Pololu Compact

Check the data sheet of your servo controller to see if it supports any of these protocols and, if so, whether you need to configure it in any way to support the required protocol.

The Mini SSC protocol only sends 3 bytes to move a servo whereas the Pololu Compact protocol requires 4. If your robot has lots of servos, such as a humanoid, then you want to send the message for all of the servos as quickly as possible. So the protocol requiring the fewest bytes is normally the best choice and you should use the fastest possible baud rate.

The only 'disadvantage' of the Mini SSC protocol is that you cannot change the center and range positions of the servo. However: some controllers such as the Pololu Maestro series have an application to allow you to do this.

This driver only requires two wires between the micro controller and the servo controller. You must connect the Ground supply from your micro controller to the Ground supply on the servo controller as well as the Transmit pin from your micro controller UART to the receive pin of the servo controller. This allows you to connect multiple servo controller boards to the same UART - often referred to as 'daisy chaining'.

If your board supports daisy chaining then you normally have to configure each board to respond to a different starting number for the servos. For example: if you have 16 servos and 2 controller boards that support 8 servos each then configure one board to start at servo 0 (so it controls servos 0 to 7) and configure the other board to start at servo 8 (so it controls servos 8 to 15).

### Example

Just use the generic example in "*Actuators*" (see page 38)

All servos provide the following functions

| **Function Summary** | |
|---|---|
| | setConfig<br>Changes the servo centre position and range at runtime. |

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Servo Driver (Hardware PWM)

This is a servo driver for controlling servos that are plugged directly into your board and uses hardware PWM to send commands to each servo.

For hardware PWM each servo must be connected to a different 16 bit hardware PWM pin.

Hardware PWM tends to be 'more exact' than software PWM and requires less processing overhead. However: a processor only tends to have a handful of hardware PWM pins which may not be enough for your needs. If this is the case then look at one of the alternative servo drivers.

### Example

Just use the generic example in "*Actuators*" (see page 38)

All servos provide the following functions

| Function Summary | |
|---|---|
| | setConfig<br>Changes the servo centre position and range at runtime. |

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |

## Actuator - Function Summary

| | |
|---|---|
| `boolean` | isInverted<br>Test if an actuator is inverted |

## Servo Driver (Software PWM)

This is a servo driver for controlling servos that are plugged directly into your board and uses software PWM to control each servo.

For software PWM each servo in the list can be attached to any I/O pin you like. However it does require more processing overhead and you are limited by the number of 16 bit timers available. Software PWM signals are not as accurate as hardware PWM signals because they are effected by other devices (ie interrupts from UARTs, timers etc) so you may experience some jitter - although with 'modified servos' this isn't noticeable.

Note that if you have a lot of servos then you can break them down into separate lists, or 'banks' each with their own driver and timer.

When using software PWM to control a large number of servos, ie more than 10, then research suggests that performance is optimal when you split the servos into several banks with each bank containing the same number of servos.

### Example

Just use the generic example in "*Actuators*" (see page 38)

All servos provide the following functions

| Function Summary | |
|---|---|
| | setConfig<br>Changes the servo centre position and range at runtime. |

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |

## Actuator - Function Summary

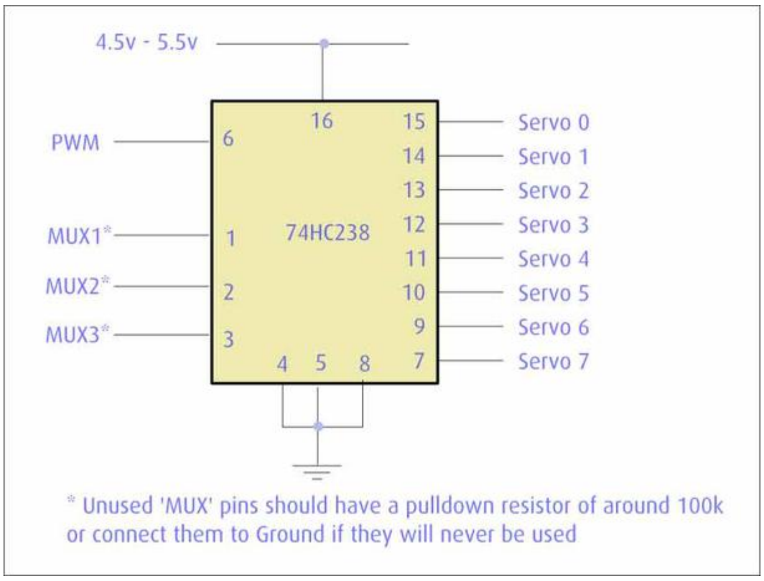
| | |
|---|---|
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Servo Driver (multiplexed)

This is a servo driver for controlling servos that are plugged directly into your board and uses hardware PWM via a multiplexer to send commands to each servo.

Although this requires additional hardware it brings the accuracy benefits of hardware PWM but requires less timers when you have a number of servos to control.

This driver only uses only one 16 bit hardware PWM pin to drive up to 8 different servo - but it does require an extra hardware chip the 74HC238:-


4.5v - 5.5v
PWM
MUX1*
MUX2*
MUX3*
74HC238
16
6
1
2
3
4
5
8
15
14
13
12
11
10
9
7
Servo 0
Servo 1
Servo 2
Servo 3
Servo 4
Servo 5
Servo 6
Servo 7
* Unused 'MUX' pins should have a pulldown resistor of around 100k or connect them to Ground if they will never be used

The hardware PWM pin from your controller is connected to pin 6 of the device. Pins 1,2 and 3 of the device are used to control which of the output servo pins receives each PWM pulse. By attaching pins 1, 2 and 3 to digital output pins on your micro-controller then WebbotLib automatically sets these pins to make sure that the PWM signal is sent to the correct servo.

The number of digital pins required depends on how many servos you want to control:-

1 servo - then MUX1, MUX2, MUX3 are not used

2 servos - connect MUX1 to a digital I/O pin on your micro-controller, MUX2 and MUX3 are not used

3 to 4 servos - connect MUX1 and MUX2 to digital I/O pins on your micro-controller, MUX3 is not used

5 to 8 servos - connect MUX1, MUX2 and MUX3 to digital I/O pins on your micro-controller

If any of the MUX pins aren't required then they should be made 'low' either by connecting them to ground or by using a pull-down resistor( say 100k but any value would do).

Note that if you have a lot of servos then you can break them down into separate lists, or 'banks', each with their own driver, timer and 74HC238 chip.

### Example

Just use the generic example in "*Actuators*" (see page 38)

All servos provide the following functions

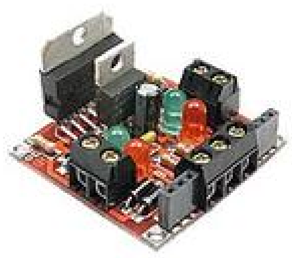
| Function Summary | |
|---|---|
| | setConfig<br>Changes the servo centre position and range at runtime. |

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Solarbotics L298 driver and Sparkfun L298 driver

Supports the Solarbotics L298 motor driver and the similar Sparkfun L298 motor driver.



See
http://www.solarbotics.com/assets/datasheets/solarbotics_l298_compact_motor_driver_kit.pdf
for the datasheet.

Each motor requires an I/O pin which provides hardware PWM and two I/O pins to control the direction.

Since we are using hardware PWM then the pulses sent to the motors are very exact. This library can cope with any number of motors and so the only real limitation is the number of PWM channels that your micro controller provides. Each motor controller board can drive two motors.

The remaining two pins are the 'enable' pins. If the motor is turning the wrong way then you can either:

- swap over the two enable pins
- swap the leads going to the motor
- toggle the 'Inverted' option for the motor in Project Designer

### Example

Just use the generic example in "*Actuators*" (see page 38)

All actuators provide the following functions:

| **Actuator - Function Summary** | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |

| Actuator - Function Summary | |
|---|---|
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Toshiba TB6612FNG (2 or 3 pin) driver

Controls two DC motors connected to the TB6612FNG motor driver using pulse width modulation (PWM).

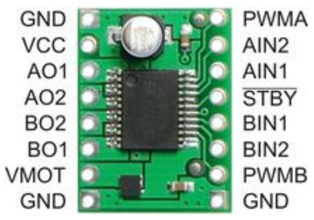
The device requires two power supplies:

- Vcc powers the 'on-board' logic and should be in the range 2.7v to 5.5v
- VM is used to drive the motors and should be in the range 4.5v to 13.5v (check what voltage your motors need) and can supply 1.2 amps per motor.

The device is a 'surface mount' device so I have used the breakout board available from Pololu:

http://www.pololu.com/catalog/product/713/resources

This device is also sold by outlets such as SparkFun and ActiveRobots.



The breakout board comes with header pins for use with a bread board and must be soldered on. Alternatively: for use in a finished robot you can just solder your wires directly to the board. The board also comes with some capacitors and reverse voltage protection for the motor supply.

I have supplied two different software drivers for this board:

### 2 Pin Mode

Requires two PWM signals per motor. On microprocessors such as the ATMega168 this will mean that the driver will use up Timer 0 and Timer 2. So it really eats into your available timers.

### 3 Pin Mode

Only requires one PWM signal per motor but requires two digital output pins per motor as well.

The 'STBY' pin can be connected to a digital output pin to put the device into 'stand by' mode. I don't support this directly in the library but if you wish to utilise this feature then connect it to any digital output pin and pull the pin low to put the device into standby or high to enable the device.

NB If you don't want to make use of the standby feature then you must connect the STBY pin to Vcc.

## 2 Pin Connections

To use the 2 pin driver then you will need to connect the device as follows:

Connect the 5v regulated supply from your micro controller to the Vcc pin

Connect the ground from your micro controller to the GND pin

Connect your motor supply (+ve) to VMOT and the ground wire to GND.

Connect the STBY pin to Vcc if you are not using an output pin to set the controller to standby mode.

DC Motor 1 should be connected to the AO1 and AO2 pins and DC Motor 2 should be connected to the BO1 and BO2 pins.

Connect PWMA to Vcc. AIN1 and AIN2 should be connected to two PWM output pins on your micro controller and these two pins will be used to control motor 1.

Connect PWMB to Vcc. BIN1 and BIN2 should be connected to two PWM output pins on your micro controller and these two pins will be used to control motor 2.

## 3 Pin Connections

To use the 3 pin driver then you will need to connect the device as follows:

Connect the 5v regulated supply from your micro controller to the Vcc pin

Connect the ground from your micro controller to the GND pin

Connect your motor supply (+ve) to VMOT and the ground wire to GND.

Connect the STBY pin to Vcc if you are not using an output pin to set the controller to standby mode.

DC Motor 1 should be connected to the AO1 and AO2 pins and DC Motor 2 should be connected to the BO1 and BO2 pins.

AIN1 and AIN2 should be connected to +5v digital output pins on your micro controller and PWMA should be connected to one of the PWM output pins on your micro controller. These three pins will be used to control motor 1.

BIN1 and BIN2 should be connected to +5v digital output pins on your micro controller and PWMB should be connected to one of the PWM output pins on your micro controller. These three pins will be used to control motor 2.

## Example

Just use the generic example in "*Actuators*" (see page 38)

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Audio

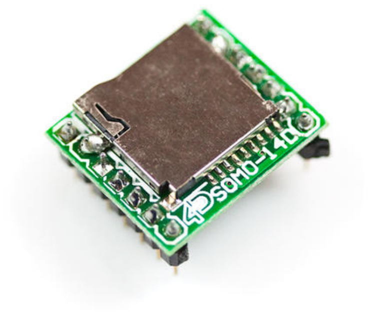
This category contains a variety of different audio output devices and routines.

The following devices are supported:

## SOMO14D

The SOMO14D is an audio file playback device.

Manufactured by 4D Systems see http://www.4dsystems.com.au/prod.php?id=73 but available from other outlets such as SparkFun.



The micro sdCard (up to 2Gb) can hold up to 512 audio tracks in 'ad4' format. Check their website for a free utility to convert from other audio file formats. Each file name must be a 4 digit number, with leading zeroes', and an extension of '.ad4' ie '0000.ad4' to '0511.ad4'. This allows the files to be accessed by number rather than by name.

The device requires a 3.3v VCC supply as well as a common GND. Their data sheet shows how it can be powered from a 5v supply by the addition of some diodes to reduce the voltage.

The device is controlled by connecting two pins from your micro controller to the CLK and DATA pins. NB if your micro controller output pins are 5V then you need to add a 100 ohm resistor in series with each of these lines to allow for the fact that the SOMO is running at 3.3V.

An optional third wire can be connected from the BUSY output to your micro controller (no extra resistor need) so that your program can sense whether a sound file is still being played.

A loudspeaker can be connected directly to the SPK+ and SPK- outputs or the AUDIO output can be used with an external amplifier - see the data sheet.

You could use this device for playing background music, sound effects, speaking sentences, or you could store individual words as individual files and then play one after another to make up sentences on the fly.

Here is a very simple example that just plays a sound on powered up (assumes you have called the device 'mySomo' in Project Designer):

```
// Initialise the software
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
// Play file '0000.ad4'
mySomo.play(0);
return 0;
}

// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart) {
// Nothing to do
return 0;
}
```

Here is another example that strings individual sounds together into a sequence so that they are played as single effect. Because we need to know when each individual sound has finished, so that we can start the next one in the sequence, then we need to make sure the 'Input pin from BUSY' parameter in Project Designer has been set up.

```
// Sequence of sound numbers ending in -1
int16_t effect1[] = { 1, 20, 15, 6, -1 };
int16_t effect2[] = { 1, 4, 5, -1 };

// Current sequence being played,
// initial value is NULL
int16_t* seqPos;
```

```
// Start playing a sequence of sounds
void play(int16_t* effect){
    // Stop any current playback
    mySomo.stop();
    seqPos = effect;

    // Start playing the first
    // sound in the sequence
    mySomo.play(*seqPos);
}
```

```
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart) {
    // If we are playing a sequence
    if(seqPos){
        // And the current sound has finished
        if(!mySomo.isBusy()){
            // Move to next sound in sequence
            seqPos ++;
            if(*seqPos==-1){
                // Sequence has finished
                seqPos = NULL;
            }else{
                // Play next sound in sequence
                mySomo.play(*seqPos);
            }
        }
    }

    // All your other code goes here ie reading sensors,
    // controlling motors, servos etc.

    // Pseudo code
    if( robot has hit something){
        play(effect1); // Say "I banged my head"
    }else if( robot has fallen over){
        play(effect2); // Say "Ouch. What happened"
    }

    return 0;
}
```

The SOMO14D provides the following functions:

| Somo14D - Function Summary | |
|---|---|
| | play<br>Start playing the specified track number. |
| | stop<br>Stop playing. |
| | setVolume<br>Set the volume level. |
| `boolean` | isBusy<br>Returns TRUE if the device is currently playing a track or FALSE if not. |

# Text to Speech

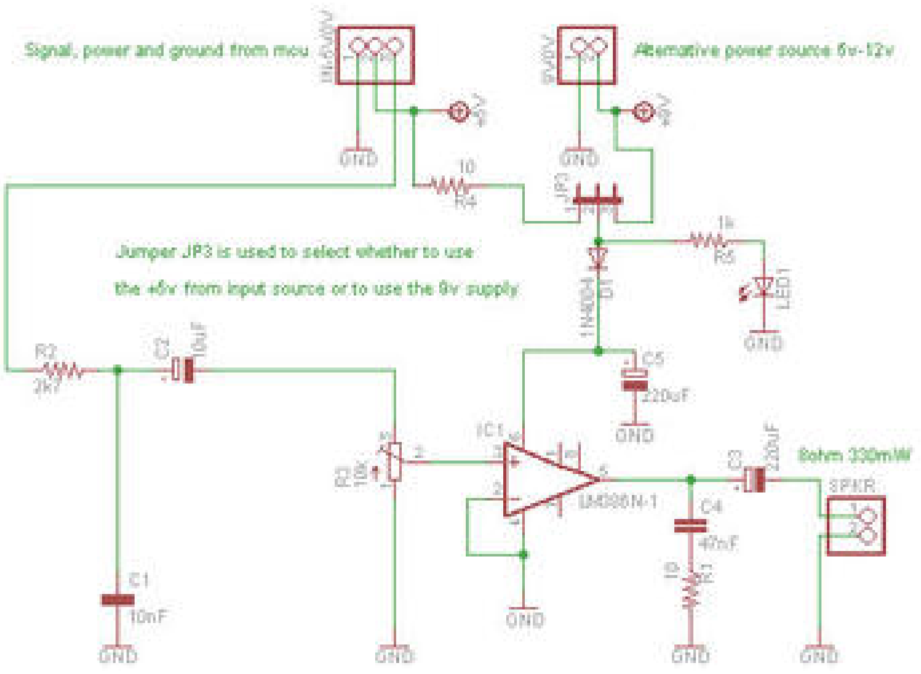
Implements a voice synthesiser using a single hardware PWM pin.

## Hardware

Note that you must add an amplifier board to drive the loud speaker. Do NOT connect a loudspeaker directly to your micro controller!!

This software works by changing the duty cycle of a 20kHz carrier frequency.

You can build a suitable amplifier yourself very cheaply without any SMD components and the board is just over an inch square:


Signal, power and ground from mcu
Alternative power source 6v-12v
GND
10
R4
JP3
1k
R5
LED1
Jumper JP3 is used to select whether to use
the +5v from input source or to use the 9v supply
1N4004
D1
C5
220uF
R2
2k7
C2
10uF
IC1
R3
10k
LM386N-1
C3
220uF
8ohm 330mW
SPKR
C4
47nF
R1
C1
10nF

I have uploaded my Eagle schematic and board files at http://www.societyofrobots.com/member_tutorials/node/216 but it would also be easy to use strip board if you don't want to make a PCB.

In brief: the amplifier input starts with R2 and C1 which filters out the 20kHz carrier frequency. C2 then passes the audio signal to the amplifier IC1. R3 is a trimmer to set the required output volume. R5 and LED1 provide a 'power on' indicator. The jumper allows you to select whether you are using the power from your robot board or from a separate battery.

Alternatively: there are some commercial boards available - just make sure the amplifier you use has a low-pass filter that filters out this 20kHz or add R2, C1 and C2 yourself.

If your robot goes outdoors then I suggest you use a Mylar speaker rather than a paper one - as they survive damp conditions without falling apart! Also make sure the speaker you purchase is easy to mount. If there are no screw holes on the speaker then you will need to hot glue it into place.

Finally some notes about speech quality:

1. In order to keep the software as small as possible then it uses 4 bit audio samples at a low sampling frequency. So don't expect the quality to be high fidelity - it's very retro!!
2. The quality is also effected by all the other interrupts that are going on depending on your application.
3. The logic to convert English text into sounds is quite complex and not always bullet proof
4. The output is better is the speaker is in some kind of box. Try cupping the loadspeaker in your hands to hear what I mean!

My website, http://webbot.org.uk has some example outputs as WAV files so that you can 'hear before you make'.

## Software

The synthesiser is an output stream and so you can access it via the various print commands provided by all streams. It can also be set to be the standard output, or standard error, destination stream.

The simplest method to output some text, assuming you have called it 'speech' in Project Designer, is:

```
speech.print("Death to all humans\n");
```

or:

```
speech << "Death to all humans\n");
```

or even better because it uses less RAM:

```
speech.print_P( PSTR("Death to all humans\n") );
```

Note that it will queue up the text until it finds a '\n' at which point it will start talking.

## Finally

Generating audio waveforms is difficult at the best of times and if your board is also talking to other sensors, motors and servos then it is even harder in a single tasking environment! I make no apologies for this.

So if the speech output is satisfactory until such time as you add all your other code then I suggest that you think about having a separate board just for speech. This board could receive the text via UART, I2C or SPI - all of which are supported using WebbotLib - and can then speak the text whilst your main board is going at full speed. Of course this board could be very simple, and cheap, such as the $50 Robot board from Society of Robots http://www.societyofrobots.com/robot_tutorial.shtml but using an ATMega328P rather than the ATMega8.

The speech synthesiser provides the following functions:

| Speech - Function Summary | |
|---|---|
| uint8_t | getPitch<br>Returns the current voice pitch. |
| | setPitch<br>Set the voice pitch. |

A Stream supports the following functions:

| Stream - Function Summary | |
|---|---|
| | print<br>Prints the value of a number, or a string, to the stream. |
| | print_P<br>Prints a string from program memory to the stream. |
| | println<br>Sends a carriage return line feed sequence to the stream to indicate the end of a line of text. |
| int | read<br>Reads the next byte from the device. |
| int | write<br>Writes a single byte to the output stream. |
| size_t | write<br>Writes out a sequence of bytes from a given position in RAM. |
| size_t | write_P<br>Writes out a sequence of bytes from a given position in program memory. |

## Tone Player

Play a tone, or ring tone tune, to any output pin for a fixed duration or until told to stop.

This will output a square wave of 50% duty cycle to any digital I/O Pin and requires the exclusive use of a timer. Multiple tone generators can be created but each one will require its own timer. In order to allow the output to be sent to any output pin then the pin toggling is done via software and hence the frequency range is limited to the audible spectrum.

Note that you must NOT connect a loudspeaker directly to the output pin as you will blow the pin. The reason being that for +5V micro processor and an 8Ω speaker then the current will be 5/8 = 625mA which is way in excess of the typical 40mA that a typical pin can provide.

The solution is to add a resistor in series with the loudspeaker. A value between 150Ω and 1kΩ would do the trick. Just don't expect a 'ghetto blaster' in terms of volume. For example: for +5v processor and using a 150Ω resistor with an 8Ω speaker then (roughly):

- total resistance = 150 + 8 = 158Ω
- total current = 5v / 158Ω = 32mA (which is within the ability of most processors)
- voltage across speaker = 5v * (8Ω / 158Ω) = 0.253V
- power for speaker = 32mA * 0.253V = 8mW

You can mix the outputs of multiple tone players into one speaker by adding a resistor from each output pin to one speaker connection and then connecting the other speaker connection to ground.

Whilst the tone player can play a range of different frequencies then there are some shortcuts defined in tone.h for different musical notes/octaves. Open up that file to see what they are called.

The tone player supports the following functions:

| **TonePlayer - Function Summary** | |
|---|---|
| | play<br>Play a single tone or a whole tune. |
| | stop<br>Stop playing the current tone or tune. |
| `boolean` | isPlaying<br>Return TRUE if a tone or tune is currently playing or FALSE if not. |

## TonePlayer - Function Summary

| | |
|---|---|
| | attach<br>Attach a callback function that is called when the current tone has stopped playing. |
| | detach<br>Remove any callback registered with this tone player. |

## Basic

This category contains a number of basic types.

The following devices are supported:

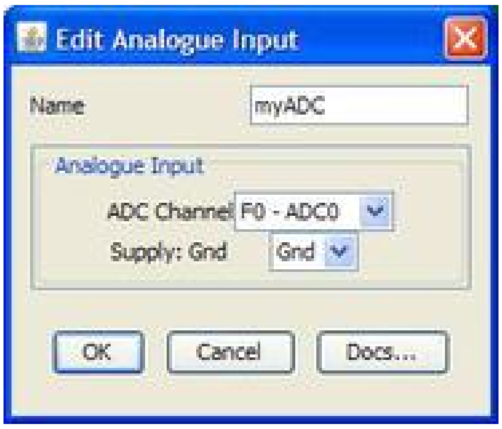
## Analogue Input

Performs analogue to digital conversion (ADC).

The ADC unit works with channels - each microprocessor will map a channel to a given pin. These pins are often the same ones as the standard I/O pins - but not always.

The number of channels available will also depend on your microprocessor.

To add an analogue input from Project Designer then drag the entry to your list of devices. This will open a window:


Edit Analogue Input

Name: myADC

Analogue Input

ADC Channel: F0 - ADC0

Supply: Gnd: Gnd

Change the name to something more meaningful to your project and select the ADC channel from the drop down list and then click Ok. Regenerate the code for your project and you will see some example code showing how to read the input. Note that you can always move the input to another channel in the drop down list and, so long as you don't change the name, then your code won't need modifying.

The ADC will, by default, be initialised with a prescaler of 64 and a reference voltage of AVCC. These settings are fine for general usage. Alternatively you can change the values at runtime with the a2dSetPrescaler and a2dSetReference commands.

There are times, e.g. a data acquisition/logging project, where you want to be able to read various inputs within a for loop and so accessing the inputs by name may not be ideal. In this case you can access the channel directly. For example: this will read all of the analogue channels on the board:

```
for(int i=0; i<NUM_ADC_CHANNELS;i++){
    a2dConvert8bit(ADC_NUMBER_TO_CHANNEL(i));
}
```

## Standard Function Summary

| | |
|---|---|
| `uint8_t` | a2dConvert8bit(ADC_CHANNEL channel)<br>Read the value from an ADC channel and return it as a single byte in the range 0-255. |
| `uint16_t` | a2dConvert10bit(ADC_CHANNEL channel)<br>Read the value from an ADC channel and return it as a value in the range 0-1023. |
| `uint8_t` | a2dReadPercent(ADC_CHANNEL channel)<br>Read the value from an ADC channel and return it as a single byte percentage in the range 0-100. |
| `uint16_t` | a2dReadMv(ADC_CHANNEL channel)<br>Returns the value of an ADC_CHANNEL in milli-volts. |

## Advanced Function Summary

| | |
|---|---|
| `uint16_t` | a2dGetAVcc(void)<br>Returns the AVcc voltage in millivolts ie 5 volts would be returned as 5000. |
| | a2dOff()<br>This will turn off the ADC unit which will save power. |
| | a2dOn()<br>This will turn on the ADC unit. |
| | a2dSetPrescaler(uint8_t prescale)<br>Change the prescaler that sets the ADC speed vs accuracy. |
| `ADC_CHANNEL` | ADC_NUMBER_TO_CHANNEL<br>This macro can be used to convert a number into a channel. |
| `uint8_t` | NUM_ADC_CHANNELS<br>Returns the number of ADC channels on this device. |

## Standard Function Detail

### a2dConvert8bit

`uint8_t a2dConvert8bit(ADC_CHANNEL channel)`

Read the value from an ADC channel and return it as a single byte in the range 0-255.

### Syntax

a2dConvert8bit(name)

a2dConvert8bit(channel)

### Parameters

name - this is the name you have specified in Project Designer for the device.

channel - the channel to be read (see example below).

### Returns

A value in the range 0 to 255 which can be stored in an 'uint8_t'.

### Example

If you have created the Analogue Input in Project Designer with a name such as 'myADC' then you can read the channel using:

```
uint8_t value = a2dConvert8bit( myADC );
```

Alternatively: if you want to read a specific channel, e.g. channel 9, then you should write:

```
uint8_t value = a2dConvert8bit( ADC_NUMBER_TO_CHANNEL(9) );
```

## a2dConvert10bit

```
uint16_t a2dConvert10bit(ADC_CHANNEL channel)
```

Read the value from an ADC channel and return it as a value in the range 0-1023.

### Syntax

a2dConvert10bit(name)

a2dConvert10bit(channel)

### Parameters

name - this is the name you have specified in Project Designer for the device.

channel - the channel to be read (see example below).

### Returns

A value in the range 0 to 1023 which can be stored in an 'uint16_t'.

### Example

If you have created the Analogue Input in Project Designer with a name such as 'myADC' then you can read the channel using:

```
uint16_t value = a2dConvert10bit( myADC );
```

Alternatively: if you want to read a specific channel, e.g. channel 9, then you should write:

```
uint16_t value = a2dConvert10bit( ADC_NUMBER_TO_CHANNEL(9) );
```

---

## a2dReadPercent

`uint8_t a2dReadPercent(ADC_CHANNEL channel)`

Read the value from an ADC channel and return it as a single byte percentage in the range 0-100.

### Syntax

a2dReadPecent(name)

a2dReadPercent(channel)

### Parameters

name - this is the name you have specified in Project Designer for the device.

channel - the channel to be read (see example below).

### Returns

A value in the range 0 to 100 which can be stored in an 'uint8_t'.

### Example

If you have created the Analogue Input in Project Designer with a name such as 'myADC' then you can read the channel using:

```
uint8_t percent = a2dReadPercent( myADC );
```

Alternatively: if you want to read a specific channel, e.g. channel 9, then you should write:

```
uint8_t percent = a2dReadPercent( ADC_NUMBER_TO_CHANNEL(9) );
```

### Notes

This function is useful, for example, if you have got a potentiometer connected to an Analogue Input pin and you want to read the 'position' rather than the actual voltage. The same piece of code would then work without modification if used on another board that uses a different reference voltage for the analogue to digital unit of the processor.

---

## a2dReadMv

`uint16_t a2dReadMv(ADC_CHANNEL channel)`

Returns the value of an ADC_CHANNEL in milli-volts. ie a value of 1.65V would return 1650.

### Syntax

a2dReadMv(name)

a2dReadMv(channel)

### Parameters

name - this is the name you have specified in Project Designer for the device.

channel - the channel to be read (see example below).

### Returns

An 'uint16_t' representing the number of millivolts.

### Example

If you have created the Analogue Input in Project Designer with a name such as 'myADC' then you can read the channel using:

```
uint16_t mV = a2dReadMv( myADC );
```

Alternatively: if you want to read a specific channel, e.g. channel 9, then you should write:

```
uint16_t mV = a2dReadMv( ADC_NUMBER_TO_CHANNEL(9) );
```

## Advanced Function Detail

### a2dGetAVcc

`uint16_t a2dGetAVcc(void)`

Returns the AVcc voltage in millivolts ie 5 volts would be returned as 5000.

#### Syntax

a2dGetAVcc()

#### Parameters

None

#### Returns

The millivolts used by the analogue to digital conversion unit as an 'uint16_t'.

#### Notes

This will always return the same fixed value for a given board design.

---

### a2dOff

`a2dOff()`

This will turn off the ADC unit which will save power.

### Syntax
a2dOff()

### Parameters
None

### Returns
Nothing

### Note
Before reading any more values you should call a2dOn() to turn the ADC unit back on again. If you are using power hungry devices like motors and servos then the power saved by turning the ADC unit off will be so minimal that it is not worth doing.

### See Also
"*a2dOn()*" (see page )

---

## a2dOn
`a2dOn()`

This will turn on the ADC unit.

This is called automatically at start up and so you should not need to call it again other than after an a2dOff().

### Syntax
a2dOn()

### Parameters
None

### Returns
None

### See Also
"*a2dOff()*" (see page )

---

## a2dSetPrescaler
`a2dSetPrescaler(uint8_t prescale)`

Change the prescaler that sets the ADC speed vs accuracy.

### Syntax
a2dSetPrescaler(prescale)

### Parameters

prescale: may be one of the following values:-

```
ADC_PRESCALE_DIV2
ADC_PRESCALE_DIV4
ADC_PRESCALE_DIV8
ADC_PRESCALE_DIV16
ADC_PRESCALE_DIV32
ADC_PRESCALE_DIV64
ADC_PRESCALE_DIV128
```

Lower values will be quicker but less accurate. The default value is ADC_PRESCALE_DIV64.

### Returns

Nothing

## ADC_NUMBER_TO_CHANNEL

`ADC_CHANNEL ADC_NUMBER_TO_CHANNEL`

This macro can be used to convert a number into a channel.

### Syntax

ADC_NUMBER_TO_CHANNEL(num)

### Parameters

num: the ADC number from 0 up to NUM_ADC_CHANNELS - 1

### Returns

The ADC channel number to be used when reading an ADC channel

### Example

To read all of the channels:

```
for(uint8_t num=0; num<NUM_ADC_CHANNELS; num++){
    uint8_t value = a2dConvert8bit( ADC_NUMBER_TO_CHANNEL(num) );
}
```

## NUM_ADC_CHANNELS

`uint8_t NUM_ADC_CHANNELS`

Returns the number of ADC channels on this device.

## Analogue Output - PWM

Provides a generic way of generating a hardware PWM signal.

The duty cycle of the PWM can be smoothed out to provide an analogue DC voltage output.

PWM has two main requirements:

1. The frequency: ie the 'off time' + the 'on time'
2. The duty cycle: the percentage of time that that the signal is 'on'

Once the 'frequency' has been set it cannot be changed - whereas the duty cycle will dictate the percentage of the time is spent 'on'. A duty cycle of 0 means that the output is always 'off', 100 means it always 'on' and there is a linear distribution in between these two values. ie a duty cycle of 50% means that it will spend half of its time 'on' and the other half 'off'.

Note that the overall frequency can only be set **once** per timer - whereas the duty cycle can be varied from 0% to 100% on each pin that is connected to the timer. So if you want multiple PWM outputs using the functions in this section then try to put all of the entries that require the same frequency on pins from the same timer.

For example: if you attempt to set up two pins with one pin requiring a frequency of 1 hertz and the second requiring 1000 hertz then you can run into problems if you use two pins from the same timer. Assuming that you set up the first pin then the timer is set to use 1 hertz and so the request for the second pin to use 1000 hertz cannot be honoured (as it would mess up the first pin).

Also note that the slowest frequency you can set will depend upon the clock speed and whether you are using an 8 bit or 16 bit timer.

For example: if you are using an 8 bit timer (maximum value of TOP = 255) on a 16 MHz board then the slowest frequency the library can create is by using a prescaler of 1024. This gives a frequency of about (16MHz / 1024)/256 = 61Hz. So if you want really low frequencies then you would be better choosing a 16 bit timer instead.

A generic PWM/analogue output provides the following functions:

| Pwm - Function Summary | |
|---|---|
| PERCENTAGE | getPercent<br>Get the current duty cycle as a percentage. |
| | setPercent<br>Set the new duty cycle as a percentage. |

## Digital Input/Output

A digital pin that is initially set up to be either an input pin or an output pin.

Digital I/O pins provide the following functions:

| Pin - Function Summary | |
|---|---|
| | high<br>Set an output pin to high. |
| | low<br>Set an output pin to low. |
| | toggle<br>Toggle an output pin. |
| | set<br>Set an output pin to the specified state. |
| boolean | get<br>Returns the current state of the input or output pin. |
| boolean | isHigh<br>Test whether the input or output pin is currently high. |
| boolean | isLow<br>Tests whether the input or output pin is currently low. |
| | makeOutput<br>Ensures that the pin is set as an output pin and sets it to the specified state. |
| | makeInput<br>Ensures that the pin is set as an input pin and enables, or disables, the internal pull-up resistor. |
| boolean | isInput<br>Returns TRUE if the pin is currently an input pin or FALSE if it is an output pin. |
| boolean | isOutput<br>Returns TRUE if the pin is currently an output pin or FALSE if it is an input pin. |

## Pin - Function Summary

| | |
|---|---|
| TICK_COUNT | pulseIn<br>Measure the duration of an incoming pulse in µS. |
| | pulseOut<br>Sends an output pulse of the specified duration. |

## LED

Use an output pin to control an LED.

Depending upon how the LED is configured then you may need to set the port high to turn it on, or you may need to set it low. Project Designer allows you to define the LED and then just use on or off commands to make it light up or not without having to worry, each time, as to how it is wired.

Users of Board Designer: call this LED 'statusLED' so that it automatically gets registered.

Users of Project Designer: if your board doesn't have a status LED and you want to add one then just make sure you call your LED 'statusLED'.

### The Status LED

Many commercial boards include one or more 'built in' LEDs and one of these is used for flashing out runtime error code numbers to give you a visual clue as to what has gone wrong in your code. On some boards this LED is, unfortunately, connected to one of the UART pins. This is handy since it flashes to show communications are happening but unfortunately means you cannot see the flashing error code. Therefore the statusLED has some special handling so that it is only turned on/off whilst it is still set as an output pin.

In your code this special LED is called 'statusLed' ie:

```
statusLed.on();
statusLed.off();
\\ etc
```

The default is that the statusLed is turned on (as a power indicator) and whilst you may want to turn it off (in appInitSoftware for example) you are discouraged from playing with it too much otherwise you will interfere with the error code flashing !

An LED supports the following functions:

| **Led - Function Summary** | |
|---|---|
| | on<br>Turns on the LED. |
| | off<br>Turns off the LED. |
| | set<br>Turn the LED on or off. |

## Switch/Button

Supports a very simple switch, or button, which can be tested to see if has been pressed or released.

All switches/buttons support the following functions:

| Switch - Function Summary | |
|---|---|
| boolean | pressed<br>Returns TRUE if the switch/button is pressed (closed) or FALSE if released (open). |
| boolean | released<br>Returns TRUE if the switch/button is released (open) or FALSE if pressed (closed). |

# Cameras

Support for various different Cameras.

## Cameras

The commercial cameras I am aware of, and whose data sheet I have read, largely work in the same basic manner but have various 'camera specific' options. The cameras I have looked at are Blackfin, AVRcam and CMUcam. Let me know if you are aware of others that are in general use.

For example: they are all driven via a UART (normally at 115,200 baud) and have a concept of a colour bin or colour map. One such entry contains a minimum and maximum colour - ie a colour range. Once these have been sent to the camera then it can then provide 'blob' detection - returning a list of screen rectangles whose contents match these ranges of colours. So if my application wants to hunt for a 'red ball', a 'green ball' and a 'blue ball' then I can initialise three colour bins: one for each coloured ball. I can then receive a list of screen rectangles where these colours can be seen.

Why bother - why not just do the calculations in WebbotLib? Well even at a small camera resolution of say 176 by 144 then it can take 4 seconds to transmit a single image from the camera even at 115,200 baud. So requesting an image frame dump from the camera should only be done either as a last resort or if you are working from a PC say and you want to see a picture of what the camera can see. Obviously transmitting the information about a rectangle (ie top/left and bottom/right corners) is very quick and is independent of the size of the rectangle.

So that covers the basics of what the cameras support. So what about all the other stuff?

This is where it tends to get specific from one camera to the next. For example: some cameras will also provide some of the following:-

- Return the colour of a pixel at X,Y
- Return the average colour for the whole screen
- Return the number of pixels in a user-specified rectangle which match a given colour bin
- Change the camera resolution
- Change between RGB and YUV colours - most cameras only work in one colour space (see "*Color*" (see page 509) in this manual - it has routines to allow you to convert from one to the other).

So as a library designer my goal is to provide you with a consistent programming interface that can be used regardless of the actual camera you are using. In the same way as my servo drivers don't change depending on the Make/Model of servo you've got plugged in - then it would also be good if you could swap one camera for another. As with all devices then you can add as many cameras as you need.

Most cameras allow you to re-program their software and so you 'could' write your own WebbotLib functions for a given camera - but this is beyond the scope of the average person and unless you are willing to publish, support and maintain your code then it doesn't benefit the community.

In the meantime: I have tried to keep the supported calls to a minimum. Why? With some of the non-basic calls then some cameras support it directly whereas others don't and so I have to spend ages downloading a screen image and do the calculations myself. So for each camera I have commented which functions will 'run slow'.

All cameras provide the following functions:

| **Function Summary** | |
|---|---|
| uint16_t | getXresolution<br>Return the camera X resolution (ie number of pixels across). |
| uint16_t | getYresolution<br>Return the camera Y resolution (ie number of pixels down). |
| uint8_t | getNumColorBins<br>Returns the number of colour bins supported by this camera. |
| boolean | setBin<br>Sets the value of a given colour bin. |
| uint8_t | getBlobs<br>Return the number of blobs which match the specified colour bin (or all colour bins). |
| | fetchBlob<br>Return the given blob from the list in the blob variable you have specified. |
| boolean | getPixel<br>Returns the colour of a given pixel. |
| char * | getVersion<br>Returns the version number of the firmware installed on the camera (if available). |
| | setMinBlobSize<br>Reduces the amount of data returned by the camera by specifying the smallest blob size (width * height) that we are interested in. |

The following devices are supported:

AVRcam Camera 95

## AVRcam Camera

Support for the AVRcam camera.

This camera seems to have little support and availability nowadays.

For the AVRcam then the following commands are slow (ie 4 seconds or so) to complete:-

- getPixel

All cameras provide the following functions:

| Camera - Function Summary | |
|---|---|
| uint16_t | getXresolution<br>Return the camera X resolution (ie number of pixels across). |
| uint16_t | getYresolution<br>Return the camera Y resolution (ie number of pixels down). |
| uint8_t | getNumColorBins<br>Returns the number of colour bins supported by this camera. |
| boolean | setBin<br>Sets the value of a given colour bin. |
| uint8_t | getBlobs<br>Return the number of blobs which match the specified colour bin (or all colour bins). |
| | fetchBlob<br>Return the given blob from the list in the blob variable you have specified. |
| boolean | getPixel<br>Returns the colour of a given pixel. |
| char * | getVersion<br>Returns the version number of the firmware installed on the camera (if available). |
| | setMinBlobSize<br>Reduces the amount of data returned by the camera by specifying the smallest blob size (width * height) that we are interested in. |

## Blackfin Camera

Support for the Blackfin camera from Surveyor Corporation http://www.surveyor.com/blackfin/

I am very grateful to Surveyor Corporation for answering all of my questions and helping me to support this device.

Note that this library ONLY supports the camera and non of the rest of the Surveyor platform. So please don't send me questions about the Matchport etc - I don't have them, haven't used them, haven't implemented them - so can't help !

The firmware on the camera contains some powerful image processing commands thereby allowing you to delegate this intensive work and free up your micro controller to concentrate on what you want it to do.

The camera uses a UART operating at 115,200 baud (although this may vary depending on the firmware version) to talk to the micro controller.

To know more about what bits to order and how to set them up then read http://webbot.org.uk/blackfin/

That page also has details on my own Blackfin Java Console which allows you to experiment with the camera connected to your PC to visualise the commands. I believe you will find it very useful.

The camera supports a number of video resolutions and these have been defined with the following constants:-

```
BLACKFIN_160_BY_120
BLACKFIN_320_BY_240
BLACKFIN_640_BY_480
BLACKFIN_1280_BY_1024
```

The default resolution is set up in Project Designer but can be changed using the setResolution function.

You must also make sure that the UART you are using to communicate with the camera has a receive buffer of around 80 bytes. If you fail to set up the receive buffer then you will get an error message. If the receive buffer is too small then you may also get an error indicating that the receive buffer has overflowed.

The next considerations is 'colours' and 'colour spaces'. The Blackfin camera uses the YUV colour space which is very different from the RGB colour space you are probably used to. This library contains routines to convert colours from one colour space to another - see Color. If you use a non-YUV colour space then this library will automatically convert it into a YUV colour as and when required.

The Surveyor Blackfin camera also provides the following functions:

| Function Summary | |
|---|---|
| | setResolution<br>Changes the working resolution of the camera. |
| BLACKFIN_RESOLUTION | getResolution<br>Returns the current camera resolution. |
| uint32_t | countPixels<br>Returns the number of pixels that match a given colour bin either for a rectangle or for the full image. |
| | getMeanColor<br>Get the mean colour across the whole camera image. |

All cameras provide the following functions:

| Camera - Function Summary | |
|---|---|
| uint16_t | getXresolution<br>Return the camera X resolution (ie number of pixels across). |
| uint16_t | getYresolution<br>Return the camera Y resolution (ie number of pixels down). |
| uint8_t | getNumColorBins<br>Returns the number of colour bins supported by this camera. |
| boolean | setBin<br>Sets the value of a given colour bin. |
| uint8_t | getBlobs<br>Return the number of blobs which match the specified colour bin (or all colour bins). |
| | fetchBlob<br>Return the given blob from the list in the blob variable you have specified. |
| boolean | getPixel<br>Returns the colour of a given pixel. |
| char * | getVersion<br>Returns the version number of the firmware installed on the camera (if available). |

## Camera - Function Summary

| | |
|---|---|
| | setMinBlobSize<br>Reduces the amount of data returned by the camera by specifying the smallest blob size (width * height) that we are interested in. |

## Communications

This category contains a number of different devices for communicating with your computer, mobile phones etc.

The following devices are supported:

## UART

Routines for handling serial communications with other devices by using UARTs.

If you run out of hardware UARTs you can use software UARTs instead to simulate a UART in code. Obviously software simulation cannot cope with very high baud rates and this will depend on the clock speed of your processor, the required baud rate, and how 'busy' your micro-controller is.

The UART is automatically initialised to the specified baud rate.

In Project Designer you can specify the size of the transmit and receive buffers.

A transmit buffer of size 0 will mean that when you send any information to the UART then your program will wait until everything except the last byte have been sent out. If you would prefer the data to be sent out in the background, ie your program doesn't wait, then set the transmit buffer to the longest message length that you send at a time.

If you are expecting to receive data via the UART then set the receive buffer size to the length of the longest message you expect to receive (as a guideline). If you fail to read the data out of the receive buffer faster than it is being received you will get an overflow error - in which case try increasing the size of the receive buffer. In some cases it can be that the receive buffer is overflowing due to the program having to 'wait' whilst transmitting in which case it could be better to increase the size of the transmit buffer.

A hardware UART provides the following functions:

| **UartHW - Function Summary** | |
|---|---|
| | setPollingMode<br>Changes a hardware UART receive mode between interrupt driven and polling modes. |

All UARTs provide the following functions:

| **Uart - Function Summary** | |
|---|---|
| | on<br>Re-initialise a UART after it has been turned off. |
| | off<br>Disables the UART once all pending transmissions have finished. |

| Uart - Function Summary | |
|---|---|
| | attach<br>Registers a callback function that is called when a character is received by the UART. |
| | detach<br>Remove any associated call back function for this UART. |
| boolean | isBusy<br>Returns FALSE if the UART has nothing left to send. |
| | flushRx<br>Flushes the receive buffer - ie discards any received characters. |
| | flushTx<br>Flushes the transmit buffer - ie it discards any outstanding bytes. |
| boolean | isRxBufferEmpty<br>Tests if the receive buffer is empty. |

A Stream supports the following functions:

| Stream - Function Summary | |
|---|---|
| | print<br>Prints the value of a number, or a string, to the stream. |
| | print_P<br>Prints a string from program memory to the stream. |
| | println<br>Sends a carriage return line feed sequence to the stream to indicate the end of a line of text. |
| int | read<br>Reads the next byte from the device. |
| int | write<br>Writes a single byte to the output stream. |
| size_t | write<br>Writes out a sequence of bytes from a given position in RAM. |

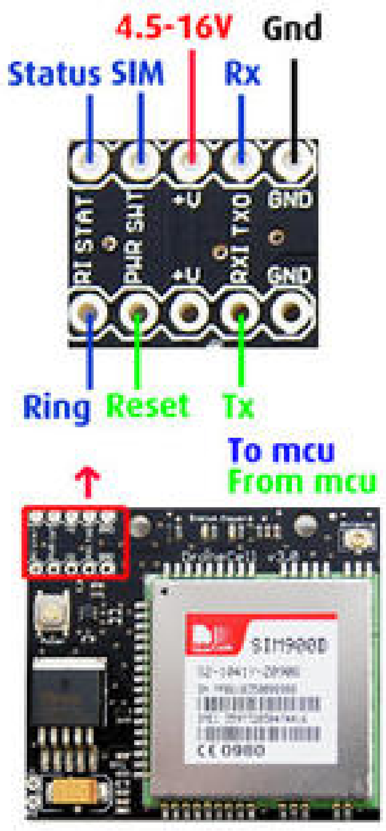
## Stream - Function Summary

| | |
|---|---|
| size_t | write_P<br>Writes out a sequence of bytes from a given position in program memory. |

## droneCell

Adds support for the DroneCell - currently limited to text messaging.

**This device is only supported in WebbotLib Version 2.03 or higher.**



When you use this device in Project Designer you will need to give it some power, as well as specifying the UART for transmitting and receiving data. The only other compulsory pin is the RESET pin which is used to initialise the device, The remaining pins are all optional input pins to the micro controller:-

- Ring indicates if there is an incoming phone call (not currently supported by WebbotLib)
- Status indicates whether the DroneCell is plugged in or not - so can easily be left blank
- SIM indicates whether or not a SIM card has been plugged in - so if you know that there is one - then this can easily be left blank.

Although the DroneCell is capable of much more - the only features currently supported by WebbotLib are to receive and send SMS messages when there is a signal. You will need to provide an appropriate SIM card for the telephone network of your choice.

One other thing you will need to define, in Project Designer, is the SMSC (SMS Message Centre) number for your network provider. Project Designer gives a list for some combinations of operator and the country you are in but is not exhaustive and I cannot guarantee that they are, or will always be, correct. If you are using the SIM card from your phone then first try going in to Phone Settings, Network Settings, Configuration etc until you can find the appropriate SMSC number. This number is in international format ie a '+' followed by the country code, then the number with the leading '0' removed.

Messages sent and received through WebbotLib always go via the SIM card ie received messages are received directly to the SIM and then WebbotLib reads them out; and sent messages are first written to the SIM card and are then sent in the background as and when you have a signal. Note that if you have problems sending lots of SMS messages one after the other then it may be that the SIM card is full - mine can only queue up a total of 10 messages 'in and out'.

The message queue, signal strength, operator name (eg 'T-Mobile' etc) are processed by WebbotLib in the background.

Obviously sending an SMS Message may have financial implications - WebbotLib makes no warranty for the open source libraries. So if you write code that sends 100s of messagess and you get a big bill - then its not my fault !

Here is some example code:-

## Showing the signal strength and network operator name

```
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart) {
cout.print_P( PSTR("Signal=") );
    cout.print(droneCell.getSignalStrength());
    cout.print_P( PSTR(" Operator=") ).print(droneCell.getOperator());
}
```

## Sending an SMS Message

Sending a message is not necessarily guaranteed to happen - the DroneCell may be doing something else - or you may have accidentally switched it off - there are lots of reasons! So there are lots of 'if' statements to keep track of where we are. Just add this inside your appControl logic:-

```
// Try to set up for sending a text message to a given number
// I've changed the destination number to contain 'x' characters so you
// dont get to see my work mobile number !
if(droneCell.initSMS("+44789xxxx638")){
    // Ok the drone cell is free so put body of message
    // You can use any of the output stream print commands
    droneCell.print_P( PSTR("Hello World") );
    // Now try to send it - may still fail eg SIM card is full or device not
connected
    boolean sent = droneCell.sendSMS();
    // We now know if the message has been sent or not so your program
    // can decide what to do !
}
```

## Auto Responder

Here is a neat way to test your code. This will listen for incoming texts from another telephone number and will then reply to the same number by prefixing the message with 'Got ya'.

ie send a message saying 'WebbotLib is cool' and get a reply of 'Got ya: WebbotLib is cool'.

```
// See if there is a received text
uint8_t msgNo = droneCell.messageWaiting();
if(msgNo){
    // Yes there is - print out which SIM card slot
    cout.print_P( PSTR("Message#") ).print(msgNo).println();
    // Try to read the message in that slot
    if(droneCell.readMessage(msgNo)){
        // we have read it - so dump it out
        cout.print_P( PSTR("Msg from '") );
        cout.print(droneCell.getMessagePhoneNumber());
        cout.print_P( PSTR("':") ).print(droneCell.getMessageBody()).println();

        // If from my work mobile
        if(strcmp_P(droneCell.getMessagePhoneNumber(),PSTR("+44789xxxx638"))==0){
            // reply
            if(droneCell.initSMS(droneCell.getMessagePhoneNumber())){
                // put body of message
                droneCell << "Got ya: " << droneCell.getMessageBody();
                // Try to send the reply
                if(droneCell.sendSMS()){
                    // Success so delete the original message
                    while(!droneCell.deleteMessage(msgNo)){
                        cout << "Delete failed\n";
                    }
                    cout << "OK Delete msg\n";
                }else{
                    // Couldn't send the reply - since we haven't deleted
                    // the original message then we will reply again later
                    cout << "Send reply failed\n";
                }
            }
        }
    }
}
```

The DroneCell supports the following functions:

| DroneCell - Function Summary | |
|---|---|
| | powerOn<br>Starts the power up sequence for the device. |
| | powerOff<br>Places the DroneCell into a low power standby mode. |
| uint8_t | getSignalStrength<br>Get the current signal strength. |
| boolean | isRegistered<br>Test if the DroneCell is currently registered with a cell phone network operator. |

| DroneCell - Function Summary | |
|---|---|
| uint8_t | messageWaiting<br>Test if a message has been received. |
| const char * | getOperator<br>Get the name of the current network operator. |
| boolean | readMessage<br>Attempt to read an incoming message from a particular SIM card slot. |
| boolean | deleteMessage<br>Delete the message in the specified SIM card slot. |
| char * | getMessagePhoneNumber<br>Having successfully read a message this will return the senders phone number. |
| char * | getMessageBody<br>Having successfully read a message this will return the body of the text message. |
| boolean | initSMS<br>Attempts to place the DroneCell into a mode where you can send an SMS text message. |
| boolean | sendSMS<br>Attempts to send the SMS text message. |

A Stream supports the following functions:

| Stream - Function Summary | |
|---|---|
| | print<br>Prints the value of a number, or a string, to the stream. |
| | print_P<br>Prints a string from program memory to the stream. |
| | println<br>Sends a carriage return line feed sequence to the stream to indicate the end of a line of text. |
| int | read<br>Reads the next byte from the device. |

## Stream - Function Summary

| | |
|---|---|
| int | write<br>Writes a single byte to the output stream. |
| size_t | write<br>Writes out a sequence of bytes from a given position in RAM. |
| size_t | write_P<br>Writes out a sequence of bytes from a given position in program memory. |

## Compass

Supports compasses.

Remember that little magnetic gadget you had as kid? Yes - a compass measures a bearing in degrees. ie hold it flat and spin around and the bearing should change all the way from 0° through to 359° and then back to 0° when you are back in your start position.

Some devices provide additional information like 'roll' and 'pitch'. Roll is like twisting your hand when you are locking or unlocking a door, and 'pitch' is like raising or lowering your 'arm' - whereas 'bearing' is like twisting your body from side to side.

All readings from this library are in degrees but devices that don't support roll and pitch will return zero for those values.

These devices either use one ADC pin for each reading or provide an I2C interface. Project Designer will guide you as to what power supply they require.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another in Project Designer and, so long as you keep the same name, then your project code wont need changing.

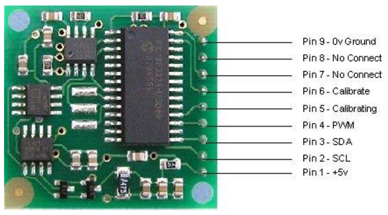
All compasses provide the following functions:

| Function Summary | |
|---|---|
| COMPASS_TYPE | getBearing<br>Returns the compass bearing, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getRoll<br>Returns the compass roll, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getPitch<br>Returns the compass pitch, in degrees, at the time of the last read(). |

The following devices are supported:

## Devantech CMPS03 Compass

The CMPS03 compass sensor.



The compass can be used either via the I2C pins 2 and 3 or by measuring the pulse length output on 'Pin 4 - PWM' of the board. Note that Project Designer lists both of these options so choose the appropriate one. The preferred method is to use I2C whenever possible. The default I2C address is 0xC0 but Project Designer lists the other possible addresses. Note that you will need to read the data sheet to find out how to modify the device to use one of the alternative I2C addresses.

The device requires a 5V regulated supply.

You may also need to calibrate the compass depending upon where you are on the earth and this is also explained in the data sheet.

All compasses provide the following functions:

| Compass - Function Summary | |
|---|---|
| COMPASS_TYPE | getBearing<br>Returns the compass bearing, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getRoll<br>Returns the compass roll, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getPitch<br>Returns the compass pitch, in degrees, at the time of the last read(). |

All sensors provide the following functions

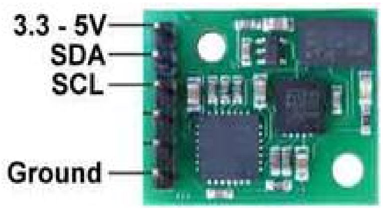
| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |

## Sensor - Function Summary

| | |
|---|---|
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Devantech CMPS09 Compass

The CMPS09 compass sensor with roll and pitch.



The compass is connected via the I2C bus and also requires a 3.3V or 5V regulated supply.

The default I2C address is 0xC0. Project Designer also list the other potential addresses that can be used but you need to read the data sheet to find out how to modify the device to use one of these alternate addresses.

You may also need to calibrate the compass depending upon where you are on the earth and this is also explained in the data sheet.

All compasses provide the following functions:

| Compass - Function Summary | |
|---|---|
| COMPASS_TYPE | getBearing<br>Returns the compass bearing, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getRoll<br>Returns the compass roll, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getPitch<br>Returns the compass pitch, in degrees, at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |

## Sensor - Function Summary

| | |
|---|---|
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## HMC5843 Compass

The HMC5843 compass sensor.

The image shows a carrier board from Sparkfun.

Datasheet: http://www.sparkfun.com/datasheets/Sensors/Magneto/HMC5843.pdf

This compass is accessed over I2C at a fixed address of 0x3C. Since the device is really a magnetometer then the floating point maths library is used to calculate the bearing, roll and pitch in degrees.

The raw magnetometer values can also be accessed via the various 'getRaw' functions.

The device allows you to specify a refresh rate (ie how many times per second the readings are updated) and this can be set up in Project Designer. Although not particularly recommended; it is possible to modify the refresh rate at run time using the various 'refresh' functions.

### Example

Assuming you have called the device 'myCompass' in Project Designer then here is some example code:

```
// read the compass values and store the results
myCompass.read();
```

```
// Print out the values in degrees
cout << " Bearing:" << myCompass.getBearing();
cout << " Roll:" << myCompass.getRoll();
cout << " Pitch:"   << myCompass.getPitch();
```

```
// Or just dump everything in one go
cout << myCompass;
```

```
// Alternatively the raw x,y,z magnetometer values can be shown
cout << Raw: << myCompass.getRawX() << ",";
cout << myCompass.getRawY() << ",";
cout << myCompass.getRawZ();
```

This compass provides the following additional functions:

| Function Summary | |
|---|---|
| int16_t | getRawX<br>Returns the raw magnetometer X value. |
| int16_t | getRawY<br>Returns the raw magnetometer Y value. |
| int16_t | getRawZ<br>Returns the raw magnetometer Z value. |
| | refresh1Hz<br>Changes the default refresh rate to once per second. |
| | refresh2Hz<br>Changes the default refresh rate to twice per second (ie every 500ms). |
| | refresh5Hz<br>Changes the default refresh rate to five times per second (ie every 200ms). |
| | refresh10Hz<br>Changes the default refresh rate to ten times per second (ie every 100ms). |
| | refresh20Hz<br>Changes the default refresh rate to twenty times per second (ie every 50ms). |
| | refresh50Hz<br>Changes the default refresh rate to fifty times per second (ie every 20ms). |
| boolean | isFunctional<br>Returns TRUE if the device is functioning correctly. |

All compasses provide the following functions:

| Compass - Function Summary | |
|---|---|
| COMPASS_TYPE | getBearing<br>Returns the compass bearing, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getRoll<br>Returns the compass roll, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getPitch<br>Returns the compass pitch, in degrees, at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## HMC5883L Compass

The HMC5883L compass sensor.

The image shows a carrier board from Sparkfun.

Datasheet: http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Sensors/Magneto/HMC5883L-FDS.pdf

This compass is accessed over I2C at a fixed address of 0x3C. Since the device is really a magnetometer then the floating point maths library is used to calculate the bearing, roll and pitch in degrees.

The raw magnetometer values can also be accessed via the various 'getRaw' functions.

The device allows you to specify a refresh rate (ie how many times per second the readings are updated) and this can be set up in Project Designer. Although not particularly recommended; it is possible to modify the refresh rate at run time using the various 'refresh' functions.

### Example

Assuming you have called the device 'myCompass' in Project Designer then here is some example code:

```
// read the compass values and store the results
myCompass.read();
```

```
// Print out the values in degrees
cout << " Bearing:" << myCompass.getBearing();
cout << " Roll:" << myCompass.getRoll();
cout << " Pitch:"   << myCompass.getPitch();
```

```
// Or just dump everything in one go
cout << myCompass;
```

```
// Alternatively the raw x,y,z magnetometer values can be shown
cout << Raw: << myCompass.getRawX() << ",";
cout << myCompass.getRawY() << ",";
cout << myCompass.getRawZ();
```

This compass provides the following additional functions:

| Function Summary | |
|---|---|
| int16_t | getRawX<br>Returns the raw magnetometer X value. |
| int16_t | getRawY<br>Returns the raw magnetometer Y value. |
| int16_t | getRawZ<br>Returns the raw magnetometer Z value. |
| | refresh1_5Hz<br>Changes the default refresh rate to one and a half times per second. |
| | refresh3Hz<br>Changes the default refresh rate to three times per second. |
| | refresh7_5Hz<br>Changes the default refresh rate to seven and a half times per second. |
| | refresh15Hz<br>Changes the default refresh rate to fifteen times per second. |
| | refresh30Hz<br>Changes the default refresh rate to thirty times per second. |
| | refresh75Hz<br>Changes the default refresh rate to seventy five times per second. |
| boolean | isFunctional<br>Returns TRUE if the device is functioning correctly. |

All compasses provide the following functions:

| Compass - Function Summary | |
|---|---|
| COMPASS_TYPE | getBearing<br>Returns the compass bearing, in degrees, at the time of the |

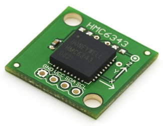
| Compass - Function Summary | |
|---|---|
| | last read(). |
| COMPASS_TYPE | getRoll<br>Returns the compass roll, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getPitch<br>Returns the compass pitch, in degrees, at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## HMC6343 Compass

The HMC6343 compass sensor.



Manufacturers Datasheet: http://www.ssec.honeywell.com/magnetic/datasheets/HMC6343.pdf

Suppliers: http://www.sparkfun.com/commerce/product_info.php?products_id=8656

This compass is accessed over I2C at address 0x32 but this address can be changed by sending it the relevant command if you know what you are doing!

The compass returns bearing, roll and pitch.

The refresh rate of the device can be defined in Project Designer and, if necessary, can be changed at runtime with the various 'refresh' functions.

### Example

Assuming you have called the device 'myCompass' in Project Designer then here is some example code:

```
// read the compass values and store the results
myCompass.read();
```

```
// Print out the values in degrees
cout << " Bearing:" << myCompass.getBearing();
cout << " Roll:" << myCompass.getRoll();
cout << " Pitch:"   << myCompass.getPitch();
```

```
// Or just dump everything in one go
cout << myCompass;
```

This compass provides the following additional functions:

| Function Summary | |
|---|---|
| | refresh1Hz<br>Changes the default refresh rate to once per second. |
| | refresh5Hz<br>Changes the default refresh rate to five times per second (ie every 200ms). |
| | refresh10Hz<br>Changes the default refresh rate to ten times per second (ie every 100ms). |

All compasses provide the following functions:

| Compass - Function Summary | |
|---|---|
| COMPASS_TYPE | getBearing<br>Returns the compass bearing, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getRoll<br>Returns the compass roll, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getPitch<br>Returns the compass pitch, in degrees, at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## HMC6352 Compass

The HMC6352 compass sensor.

Manufacturers Datasheet: http://www.ssec.honeywell.com/magnetic/datasheets/HMC6352.pdf

Suppliers: http://www.sparkfun.com/commerce/product_info.php?products_id=7915

This compass is accessed over I2C at address 0x42 but this address can be changed by sending it the relevant command if you know what you are doing!

The compass only returns the bearing and so the roll and pitch values will always be zero.

The default refresh rate can be set up in Project Designer but may, if required, be changed at runtime using the various 'refresh' commands.

### Example

Assuming you have called the device 'myCompass' in Project Designer then here is some example code:

```
// read the compass values and store the results
myCompass.read();
```

```
// Print out the bearing in degrees
cout << " Bearing:" << myCompass.getBearing();
// Or just dump everything in one go
cout << myCompass;
```

This compass provides the following additional functions:

| Function Summary | |
|---|---|
| | refresh1Hz<br>Changes the default refresh rate to once per second. |
| | refresh5Hz<br>Changes the default refresh rate to five times per second (ie |

| Function Summary | |
|---|---|
| | every 200ms). |
| | refresh10Hz<br>Changes the default refresh rate to ten times per second (ie every 100ms). |
| | refresh20Hz<br>Changes the default refresh rate to twenty times per second (ie every 50ms). |

All compasses provide the following functions:

| Compass - Function Summary | |
|---|---|
| COMPASS_TYPE | getBearing<br>Returns the compass bearing, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getRoll<br>Returns the compass roll, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getPitch<br>Returns the compass pitch, in degrees, at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Controller

Supports different hand held games controller pads.

Lots of buttons and joysticks - so great for a radio/remote controlled robot.

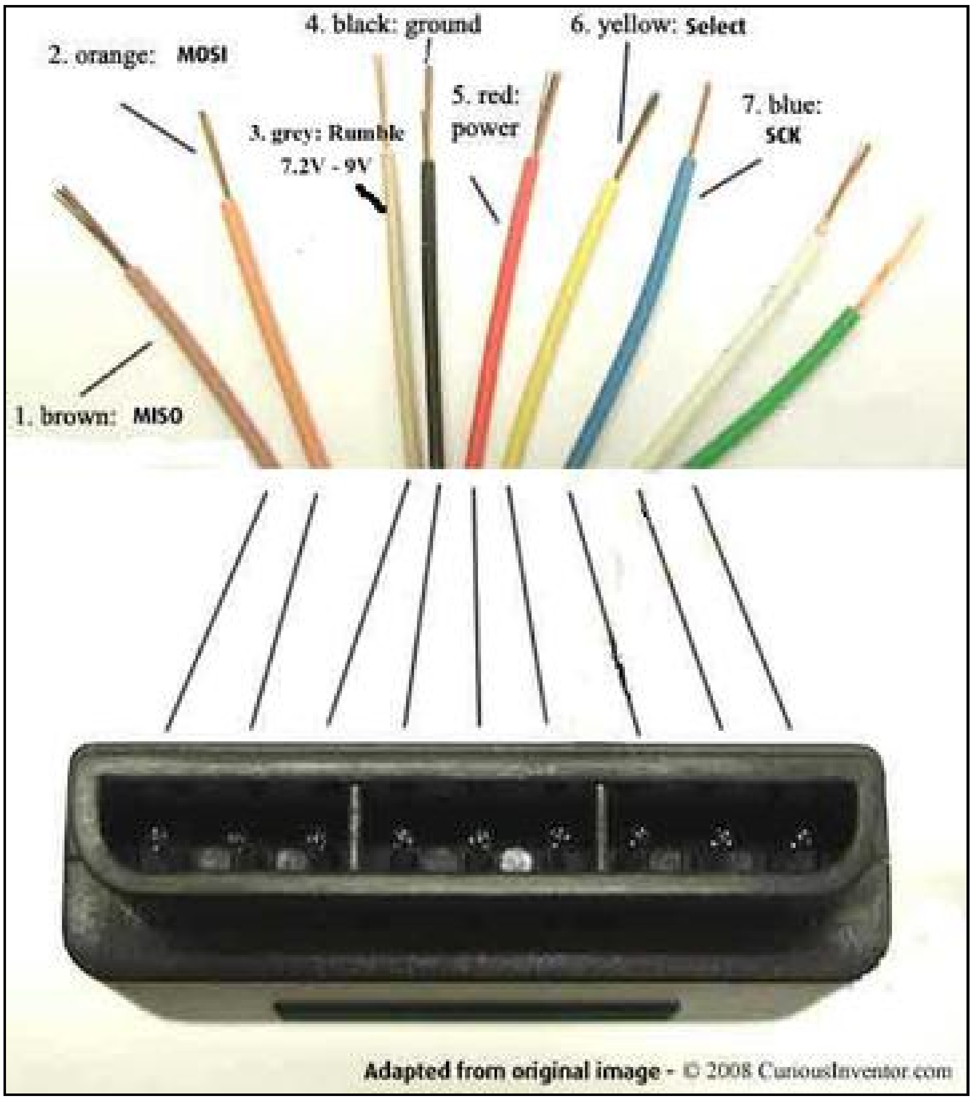
The following devices are supported:

## Sony PS2

Adds support for the Sony Playstation PS2 controllers.

Thanks to 'Dunk' for his contributions, testing and patience. Other thanks to Bill Porter.

These controllers come with a 9 pin plug - which you will need to cut off and replace with your own header pin sockets. Alternatively you can find adaptors to plug into the joystick.



You will need to provide connections for 'ground' and 'power' - to power the device. The jury is out as to whether it should be 3.3V or 5V.

The other connections are the common SPI bus lines: MISO, MOSI and SCK and the Select wire needs to be connected to a digital I/O used to select the device.

The grey wire is optional depending on whether you want to use the rumble motors. If you want to use them then connect it to an unregulated supply voltage between 7.2v and 9v.

You can of course add as many controllers as you like - each controller requires its own unique Select pin but otherwise the connections are the same for every controller.

When sharing the hardware SPI bus with an ISP programmer it is recommended that you add a pull up resistor on the Select line of the controller so that it doesn't interfere with the programmer when programming. Alternatively: only have either the controller or the programmer plugged in at any one time. If you are programming via a bootloader then this doesn't apply.

**NB The controller is slightly flakey - so, for now, only use a software SPI bus as the hardware bus seems to work too fast for the controller.**

In Project Designer you must must add the controller to an existing "*SPI Bus*" (see page 231) .

If you wish to use the joysticks then you must place the controller into analog mode. This gives access to both the left and right joysticks as well as virtual joysticks using the D-pad buttons on the left and the shape buttons on the right.

Each joystick has two values: X and Y.

Here is an example for the Axon and Axon II assuming you have called it 'controller' in Project Designer:-

```
//In appInitSoftware we will calibrate the controller
controller.calibrate(27);
```

In our main loop we read the controller:-

```
if(controller.read()){
    // We can now test the buttons
    // and the joysticks
}
```

The Sony PS2 controller supports the following functions:

## SonyPS2 - Function Summary

| | |
|---|---|
| boolean | calibrate<br>Calibrate the joysticks on the controller. |
| boolean | read<br>Read the values from the controller and store them. |
| | setRumble<br>Turn each of the rumble motors on or off. |
| boolean | buttonDown<br>Return TRUE if the button has just been pressed or FALSE if it was already pressed or is not pressed at all. |

## SonyPS2 - Function Summary

| | |
|---|---|
| boolean | buttonUp<br>Return TRUE if the button has just been released or FALSE if it was already released or is still held down. |
| boolean | buttonHeld<br>Return TRUE if the button continues to be held down. |
| boolean | buttonPressed<br>Return TRUE if the button is currently pressed. |
| uint8_t | buttonPressure<br>Return a value representing how hard a button has been pressed. |
| PS2_BUTTONS | buttonsChanged<br>Returns information on which buttons have changed state since the previous call to read the controller. |
| PS2_BUTTONS | buttonsRaw<br>Return the status of all 16 buttons. |
| boolean | isAnalogMode<br>Returns TRUE if the controller is already in analogue mode. |
| boolean | setAnalogMode<br>Activates the joysticks. |
| int8_t | joystick<br>Read a joystick value relative to its centre point including any dead zone. |
| int8_t | joystickRaw<br>Reads the raw value from a joystick. |

## Current

Supports current measurement.

All readings from this library are in 'amps' regardless of whether it is AC or DC.

Since all of the devices have been implemented in the same way then it means that you can swap one device for another in Project Designer. So long as you give the new device the same name as the old device then your project code will not need to be modified.

### Example

Assuming you have called your sensor 'myCurrent' in Project Designer then it can be accessed as follows:-

```
myCurrent.read();
cout << "Current=" << myCurrent.getAmps();
```

Or dumped to the standard out destination using:-

```
cout << myCurrent;
```

All current sensors provide the following functions:

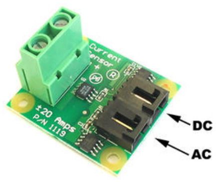
| **Function Summary** | |
|---|---|
| CURRENT_TYPE | getAmps<br>Returns the number of amps going through the sensor at the time of the last read(). |

The following devices are supported:

| | |
|---|---|
| Phidget AC Current (20 Amp) Sensor | 128 |
| Phidget DC Current (20 Amp) Sensor | 130 |
| Phidget AC Current (50 Amp) Sensor | 132 |
| Phidget DC Current (50 Amp) Sensor | 134 |

## Phidget AC Current (20 Amp) Sensor

Phidget 20A AC current sensor.



### Example

Assuming you have called your sensor 'myCurrent' in Project Designer then it can be accessed as follows:-

```
myCurrent.read();
CURRENT_TYPE amps = myCurrent.getAmps();
cout << "Current=" << amps;
```

Or dumped to the standard out destination using:-

```
cout << myCurrent;
```

All current sensors provide the following functions:

| Current - Function Summary | |
|---|---|
| CURRENT_TYPE | getAmps<br>Returns the number of amps going through the sensor at the time of the last read(). |

All sensors provide the following functions

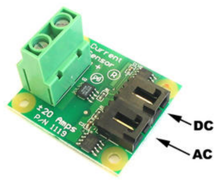
| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |

## Sensor - Function Summary

|  |  |
|---|---|
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Phidget DC Current (20 Amp) Sensor

Phidget 20A DC current sensor.



### Example

Assuming you have called your sensor 'myCurrent' in Project Designer then it can be accessed as follows:-

```
myCurrent.read();
CURRENT_TYPE amps = myCurrent.getAmps();
cout << "Current=" << amps;
```

Or dumped to the standard out destination using:-

```
cout << myCurrent;
```

All current sensors provide the following functions:

| Current - Function Summary | |
|---|---|
| CURRENT_TYPE | getAmps<br>Returns the number of amps going through the sensor at the time of the last read(). |

All sensors provide the following functions

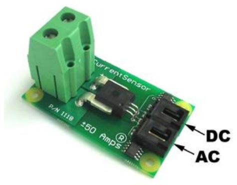
| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |

## Sensor - Function Summary

| | |
|---|---|
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Phidget AC Current (50 Amp) Sensor

Phidget 50A AC current sensor.



### Example

Assuming you have called your sensor 'myCurrent' in Project Designer then it can be accessed as follows:-

```
myCurrent.read();
CURRENT_TYPE amps = myCurrent.getAmps();
cout << "Current=" << amps;
```

Or dumped to the standard out destination using:-

```
cout << myCurrent;
```

All current sensors provide the following functions:

| Current - Function Summary | |
|---|---|
| CURRENT_TYPE | getAmps<br>Returns the number of amps going through the sensor at the time of the last read(). |

All sensors provide the following functions

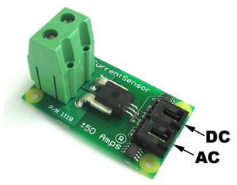
| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |

## Sensor - Function Summary

| | |
|---|---|
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Phidget DC Current (50 Amp) Sensor

Phidget 50A DC current sensor.



### Example

Assuming you have called your sensor 'myCurrent' in Project Designer then it can be accessed as follows:-

```
myCurrent.read();
CURRENT_TYPE amps = myCurrent.getAmps();
cout << "Current=" << amps;
```

Or dumped to the standard out destination using:-

```
cout << myCurrent;
```

All current sensors provide the following functions:

| Current - Function Summary | |
|---|---|
| CURRENT_TYPE | getAmps<br>Returns the number of amps going through the sensor at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |

## Sensor - Function Summary

| | |
|---|---|
| `TICK_COUNT` | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Display

Adds support for various standalone display hardware.

Most of the displays available today come in two different forms:

- Character displays
- Graphic displays

Character display have a fixed number of text columns across and a fixed number of text lines down. Think of them as being like the display in a hand held calculator. By default these devices can display standard ASCII characters like letters, numbers and punctuation. Some devices have some 'unused' characters and allow you to create your own bitmap. These are called 'custom characters'. These are useful if you want to add stuff like progress bars, or a set of icons like: play, pause, rewind, fast forward, play etc. But there are normally only a handful of available slots.

Graphic displays are more akin to your computer display - ie they are 'x' number of pixels (dots) across and 'y' pixels down. Each dot can be controlled individually - by which I mean it can be set to a particular colour. A black and white screen will either allow you turn a given pixel on or off. More advanced screens allow you to set it to one of 8, 16, 256, 65536, or even millions of colours. Displaying text is not easy as the characters are actually 'drawn'. So the size of each character is potentially variable. Hence its impossible to say how many characters across/down are allowed. You can either have lots of small characters, or less larger characters. This all sounds great - but it does come with downsides. A character display just requires you to give it a single byte representing the character you want displayed. Whereas a graphic display, assuming a character size of 5 dots by 8 dots, requires you to set all 40 dots and so may need you to send 40 bytes rather than just one depending on the abilities of the display. So these devices are more powerful - but, for the reasons above, are normally slower if you are just working with text. On the plus side: if you want your display to show maps, dials, graphs etc then you can only choose a graphic display.

The other consideration for displays is how they are connected to your robot board and the choices are either:

- Serial
- Parallel
- I2C

A serial connection just uses a UART. So only a transmit pin, ground, and a power supply are normally required.

A parallel connection tends to require either 4 or 8 I/O pins plus an overhead of around 3 extra I/O pins for control purposes. So quite I/O hungry.

An I2C connection is fast and only requires a few pins.

Which is the best? Well serial needs less pins but can be slow (since you are sending 1 bit at a time over the UART), whereas parallel displays require lots of your valuable IO pins but can be faster (as you are sending up to 8 bits at the same time). My personal favourite is I2C. Parallel devices also tend to be cheaper to buy as they are simpler designs and hence require less hardware. The serial devices often have a parallel display 'under the hood' with an extra board that accepts the serial input and then talks to the parallel display.

For example: some shop fronts, such as Sparkfun, sell a 'serial board' that plugs into a parallel display so that a parallel display can be converted into a serial display.

The actual choice of display manufacturer and model is up to you - but WebbotLib tries to make all of them appear to work in the same way as far as your code is concerned. This allows you to change from one display to another whilst minimising the effects on your code.

Here is a code snippet example of how to use a display, in this case just a plain old HD44780 compatible display, to show a bar graph of each of the analogue to digital ports:

```
// Forward reference
MAKE_WRITER(display_put_char);
// Define the display for the Axon
HD44780 display = MAKE_HD44780_4PIN(8,2,L0,L1,L2,L3,L4,L5,L6,&display_put_char);
// Create a Writer to write to display
MAKE_WRITER(display_put_char){
    return displaySendByte(&display,byte);
}
```

```
// Initialise the display and send
// rprintf output to it
void appInitHardware(void) {
    displayInit(&display);
    setErrorLog(displayGetWriter(&display));
    rprintfInit(displayGetWriter(&display));
}
```

```
// Initialise the software
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    displayAutoScroll(&display,FALSE);
    displayLineWrap(&display,FALSE);

    int cols = MIN(NUM_ADC_CHANNELS, displayColumns(&display));
    for(int n=0; n<cols;n++){
        rprintf("%d",n % 10);
    }
    return 0;
}
```

```
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart) {
    int depth = displayLines(&display)-1;
    int cols = MIN(NUM_ADC_CHANNELS, displayColumns(&display));
    for(int ch = 0; ch<cols; ch++){
        uint8_t val = a2dConvert8bit(ADC_NUMBER_TO_CHANNEL(ch));
        displayVertGraph(&display,ch,1, val, 255, depth);
    }
    return 0;
}
```

The following devices are supported:

## 4D Graphic Display

Adds support for serial graphic displays from 4D Systems.

Note that these devices require the SGC firmware to be programmed into the display so that it can be controlled via software. You may be able to order them with this already done - but if not then you will need to follow the instructions on their website.

Not all of the functionality of these devices has been made available in WebbotLib yet - notably the use of the SD card to display images and play audio and video, and the touch screen abilities.

The 4D Systems displays are either based on the GOLDELOX chip or the more powerful PICASO.

This is the only thing you need to know when using WebbotLib because it can discover information like the screen resolution at runtime by asking the display.

Model shown: µLCD-32PT © 4D Systems.

All graphic displays support Widgets (see "*Widget*" (see page 577) ) and the following functions (where possible):

| **GraphicDisplay - Function Summary** | |
|---|---|
| | shutdown<br>Shuts down the display. |
| | setPaper<br>Sets the paper colour. |
| | setInk<br>Sets the ink colour. |

## GraphicDisplay - Function Summary

| | |
|---|---|
| | moveTo<br>Moves the graphics cursor to the absolute x,y position on the display without drawing anything. |
| | moveBy<br>Moves the graphics cursor by adding the specified x,y offsets to the current graphics cursor. |
| | plotAt<br>Moves the graphics cursor to the absolute x,y position on the display and plots that pixel in the current ink colour. |
| | plotBy<br>Plots the pixel at the current graphics cursor plus the specified x,y offsets in the current ink colour. |
| boolean | readAt<br>Moves the graphics cursor to the absolute x,y position on the display and reads the colour of the pixel. |
| boolean | readBy<br>Reads the pixel at the current graphics cursor plus the specified x,y offsets. |
| | lineTo<br>Draws a line from the graphics cursor to the specified absolute x,y position in the current ink colour. |
| | lineBy<br>Draws a line from the current graphics cursor to the current graphics cursor plus the specified x,y offsets in the current ink colour. |
| PIXEL | getXRes<br>Returns the total number of pixels across the display. |
| PIXEL | getYRes<br>Returns the total number of pixels down the display. |
| PIXEL | getXPos<br>Returns the x position of the current graphics cursor. |
| PIXEL | getYPos<br>Returns the y position of the current graphics cursor. |

| GraphicDisplay - Function Summary | |
|---|---|
| | triangleAt<br>Draws, or fills, a triangle in the current ink colour between the three specified co-ordinates. |
| | triangleBy<br>Draws, or fills, a triangle in the current ink colour by adding the three specified co-ordinate offsets to the current graphics cursor. |
| | rectangleAt<br>Draws, or fills, a rectangle in the current ink colour using the two specified co-ordinates as the top left and bottom right corners. |
| | rectangleBy<br>Draws, or fills, a rectangle in the current ink colour by adding the two specified co-ordinate offsets to the current graphics cursor to give the top left and bottom right corners. |
| | circle<br>Draws, or fills, a circle in the current ink colour centred on the current graphics cursor and of the specified radius. |
| | replace<br>Replaces all pixels within the specified rectangle that are of one colour with a new colour. |
| | draw<br>Draws a text character, or string, in the current ink colour at the current graphics cursor position. |

All displays support the following functions (where possible):

| Display - Function Summary | |
|---|---|
| `DISPLAY_COLUMN` | getNumColumns<br>Returns the total number of columns available. |
| `DISPLAY_LINE` | getNumLines<br>Returns the total number of lines available. |
| | clear<br>Clear the display and set the cursor to the home position. |
| | home<br>Move the cursor to the home position ie top left corner. |

| Display - Function Summary | |
|---|---|
| | setXY<br>Move the cursor to a given location. |
| | setLineWrap<br>Turn automatic line wrapping on or off. |
| | setAutoScroll<br>Turn auto-scrolling on or off. |
| | setBacklight<br>Turn the display back light on or off (assuming that it has one and that it can be controlled in software). |
| | setBrightness<br>Set the brightness of the display. |
| | setContrast<br>Set the contrast of the display to a number between 0 (least) and 100 (most). |
| | horizGraph<br>Draws a horizontal bar graph on the display. |
| | vertGraph<br>Draws a vertical bar graph on the display. |

A Stream supports the following functions:

| Stream - Function Summary | |
|---|---|
| | print<br>Prints the value of a number, or a string, to the stream. |
| | print_P<br>Prints a string from program memory to the stream. |
| | println<br>Sends a carriage return line feed sequence to the stream to indicate the end of a line of text. |
| `int` | read<br>Reads the next byte from the device. |
| `int` | write<br>Writes a single byte to the output stream. |

## Stream - Function Summary

| | |
|---|---|
| size_t | write<br>Writes out a sequence of bytes from a given position in RAM. |
| size_t | write_P<br>Writes out a sequence of bytes from a given position in program memory. |

## 8 Segment LED

Support for an 8 segment LED display.

An 8 segment LED display provides the capability of a single digit on a typical desktop calculator. The 8 segments are made up from 7 vertical or horizontal bars and a decimal point.

These 8 elements are labelled from 'a' to 'h' as follows:-

```
--a---
|    |
f    b
|    |
--g---
|    |
e    c
|    |
--d--- h
```

Each of the above elements is turned on or off using an output pin. So we require 8 output pins to control the led segment.

WebbotLib allows us to either turn individual segments on and off or write a character.

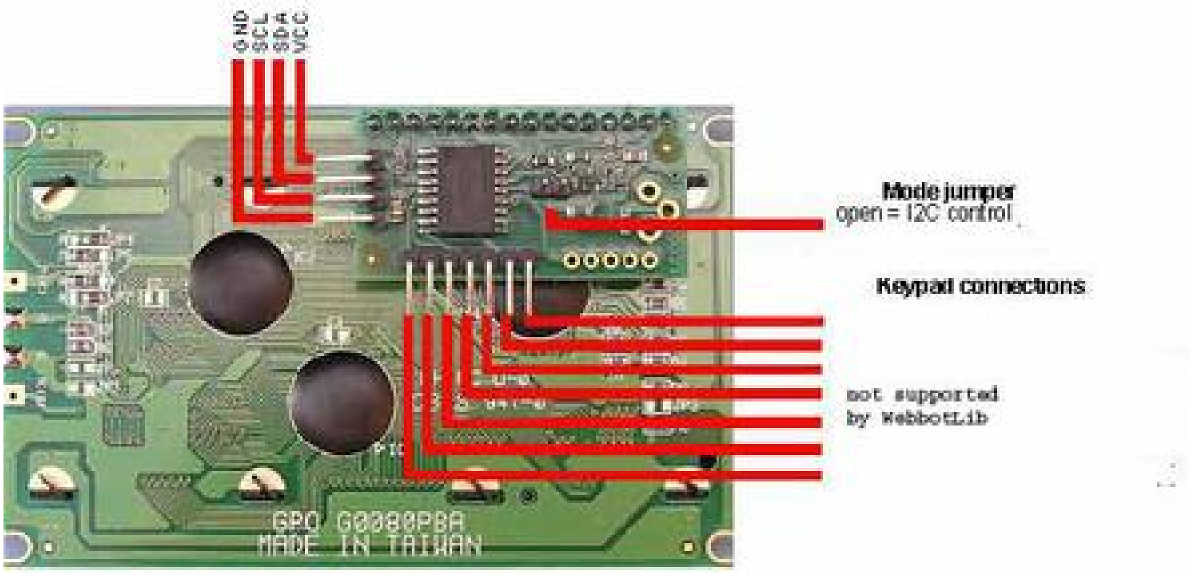
An 8 segment LED supports the following functions:

| SegLED - Function Summary | |
|---|---|
| | on<br>Turn on an individual segment of the LED display. |
| | off<br>Turn off an individual segment of the LED display. |
| | set<br>Turns an individual segment on or off. |
| uint8_t | put<br>Output a character to the LED display. |

## Devantech LCD03 (I2C) Display

Adds support for the Devantech LCD03 display using I2C.

To use the I2C interface you will need to remove the jumper that selects either serial or I2C. Do this with the power turned off.



The default I2C address for this device is 0xC6. If this clashes with another I2C device then read the LCD03 manual as you can change the address for the device. How to do this is out of the scope of this manual as its not something you should do on the fly. Alternatively: think about using a dedicated software I2C bus to avoid such conflicts - as described in the I2C section of this manual.

All displays support the following functions (where possible):

| Display - Function Summary | |
|---|---|
| DISPLAY_COLUMN | getNumColumns<br>Returns the total number of columns available. |
| DISPLAY_LINE | getNumLines<br>Returns the total number of lines available. |
| | clear<br>Clear the display and set the cursor to the home position. |
| | home<br>Move the cursor to the home position ie top left corner. |

| Display - Function Summary | |
|---|---|
| | setXY<br>Move the cursor to a given location. |
| | setLineWrap<br>Turn automatic line wrapping on or off. |
| | setAutoScroll<br>Turn auto-scrolling on or off. |
| | setBacklight<br>Turn the display back light on or off (assuming that it has one and that it can be controlled in software). |
| | setBrightness<br>Set the brightness of the display. |
| | setContrast<br>Set the contrast of the display to a number between 0 (least) and 100 (most). |
| | horizGraph<br>Draws a horizontal bar graph on the display. |
| | vertGraph<br>Draws a vertical bar graph on the display. |

A Stream supports the following functions:

| Stream - Function Summary | |
|---|---|
| | print<br>Prints the value of a number, or a string, to the stream. |
| | print_P<br>Prints a string from program memory to the stream. |
| | println<br>Sends a carriage return line feed sequence to the stream to indicate the end of a line of text. |
| int | read<br>Reads the next byte from the device. |
| int | write<br>Writes a single byte to the output stream. |

## Stream - Function Summary

| | |
|---|---|
| size_t | write<br>Writes out a sequence of bytes from a given position in RAM. |
| size_t | write_P<br>Writes out a sequence of bytes from a given position in program memory. |

## HD44780 Compatible

Adds support for LCDs based on the HD44780 controller chip - or compatibles.

This chip requires 3 control lines and either 4 or 8 data lines. All of these pins are general IO pins.

The 4 pin method requires a few less pins than the 8 pin method but is slightly slower in operation.

This device also supports a number of different sized displays. The number of lines/rows is normally 1,2 or 4 but the number of columns can typically be 8, 16, or 20.

These settings can be specified in Project Designer.

All displays support the following functions (where possible):

| **Display - Function Summary** | |
|---|---|
| DISPLAY_COLUMN | getNumColumns<br>Returns the total number of columns available. |
| DISPLAY_LINE | getNumLines<br>Returns the total number of lines available. |
| | clear<br>Clear the display and set the cursor to the home position. |
| | home<br>Move the cursor to the home position ie top left corner. |
| | setXY<br>Move the cursor to a given location. |
| | setLineWrap<br>Turn automatic line wrapping on or off. |
| | setAutoScroll<br>Turn auto-scrolling on or off. |
| | setBacklight<br>Turn the display back light on or off (assuming that it has one and that it can be controlled in software). |
| | setBrightness<br>Set the brightness of the display. |

| Display - Function Summary | |
|---|---|
| | setContrast<br>Set the contrast of the display to a number between 0 (least) and 100 (most). |
| | horizGraph<br>Draws a horizontal bar graph on the display. |
| | vertGraph<br>Draws a vertical bar graph on the display. |

A Stream supports the following functions:

| Stream - Function Summary | |
|---|---|
| | print<br>Prints the value of a number, or a string, to the stream. |
| | print_P<br>Prints a string from program memory to the stream. |
| | println<br>Sends a carriage return line feed sequence to the stream to indicate the end of a line of text. |
| int | read<br>Reads the next byte from the device. |
| int | write<br>Writes a single byte to the output stream. |
| size_t | write<br>Writes out a sequence of bytes from a given position in RAM. |
| size_t | write_P<br>Writes out a sequence of bytes from a given position in program memory. |

## Marquee

Scrolls a message across one or more LEDs.

Project Designer allows you to create a list of segmented LEDs (1 or more) and the marquee allows you to scroll a longer message across this line of LEDs. The AxonII has a single LED built in.

The library allows you to control the scroll speed as well as the 'pause' before repeating the message again.

When the message is scrolling across the marquee then there will be a brief flash if, and only if, the displayed characters for frame N are the same as for frame N-1. So if the message 'Webbot' is being scrolled across a single LED then there will be a brief flash to separate the two 'b' characters. However: there would be not flash if there are two leds as the sequence 'We', 'eb', 'bb', 'bo' and 'ot' are all unique.

### Note

The message will not appear until an end of line character '\n' has been received.

The marquee is set up as an output stream so you can use all of the 'print' and '<<' commands to send data to it.

A marquee provides the following functions:

| **Marquee - Function Summary** | |
|---|---|
| `boolean` | isActive<br>Does the marquee have an active scrolling message?If the marquee was created as a repeating message, and a message has been set, then this will always return TRUE unless marqueeStop(MARQUEE* marquee) is called to stop it. |
| | setCharDelay<br>Change the duration (in microseconds) for scrolling characters in the marquee. |
| | setEndDelay<br>Change the delay for an auto-repeating message or make it non-repeating by specifying a delay of 0. |

A Stream supports the following functions:

| Stream - Function Summary | |
|---|---|
| | print<br>Prints the value of a number, or a string, to the stream. |
| | print_P<br>Prints a string from program memory to the stream. |
| | println<br>Sends a carriage return line feed sequence to the stream to indicate the end of a line of text. |
| int | read<br>Reads the next byte from the device. |
| int | write<br>Writes a single byte to the output stream. |
| size_t | write<br>Writes out a sequence of bytes from a given position in RAM. |
| size_t | write_P<br>Writes out a sequence of bytes from a given position in program memory. |

## Matrix Orbital Serial

Adds support for various Matrix Orbital displays.

The currently supported devices are all driven via a UART and are:

```
MOSAL162A - (16 x 2 characters)
MOSAL202A - (20 x 2 characters)
```

The current code may well support other Matrix Orbital serial displays of the same format - try it but don't blame me!

These devices have a hardware header link to allow you to choose between 19200 and 9600 baud.

All displays support the following functions (where possible):

| Display - Function Summary | |
|---|---|
| DISPLAY_COLUMN | getNumColumns<br>Returns the total number of columns available. |
| DISPLAY_LINE | getNumLines<br>Returns the total number of lines available. |
| | clear<br>Clear the display and set the cursor to the home position. |
| | home<br>Move the cursor to the home position ie top left corner. |
| | setXY<br>Move the cursor to a given location. |

| Display - Function Summary | |
|---|---|
| | setLineWrap<br>Turn automatic line wrapping on or off. |
| | setAutoScroll<br>Turn auto-scrolling on or off. |
| | setBacklight<br>Turn the display back light on or off (assuming that it has one and that it can be controlled in software). |
| | setBrightness<br>Set the brightness of the display. |
| | setContrast<br>Set the contrast of the display to a number between 0 (least) and 100 (most). |
| | horizGraph<br>Draws a horizontal bar graph on the display. |
| | vertGraph<br>Draws a vertical bar graph on the display. |

A Stream supports the following functions:

| Stream - Function Summary | |
|---|---|
| | print<br>Prints the value of a number, or a string, to the stream. |
| | print_P<br>Prints a string from program memory to the stream. |
| | println<br>Sends a carriage return line feed sequence to the stream to indicate the end of a line of text. |
| `int` | read<br>Reads the next byte from the device. |
| `int` | write<br>Writes a single byte to the output stream. |
| `size_t` | write<br>Writes out a sequence of bytes from a given position in RAM. |

## Stream - Function Summary

| | |
|---|---|
| size_t | write_P<br>Writes out a sequence of bytes from a given position in program memory. |

## Sparkfun serLCD

Adds support for the Sparkfun serLCD board for various display sizes.

The serLCD product is either supplied when you buy one of their LCD devices, or can be bought as a standalone item that piggybacks onto a parallel display to convert it into a serial display.

The default baud rate is 9600 but it may changed.

At the time of writing - various www.societyofrobots.com folk have had problems with this device - something to do with its noise suppression and/or power lines - the general comment, in summary, is that 'design is not good'.

All displays support the following functions (where possible):

| Display - Function Summary | |
|---|---|
| DISPLAY_COLUMN | getNumColumns<br>Returns the total number of columns available. |
| DISPLAY_LINE | getNumLines<br>Returns the total number of lines available. |
| | clear<br>Clear the display and set the cursor to the home position. |
| | home<br>Move the cursor to the home position ie top left corner. |
| | setXY<br>Move the cursor to a given location. |
| | setLineWrap<br>Turn automatic line wrapping on or off. |
| | setAutoScroll<br>Turn auto-scrolling on or off. |
| | setBacklight<br>Turn the display back light on or off (assuming that it has one and that it can be controlled in software). |
| | setBrightness<br>Set the brightness of the display. |

| Display - Function Summary | |
|---|---|
| | setContrast<br>Set the contrast of the display to a number between 0 (least) and 100 (most). |
| | horizGraph<br>Draws a horizontal bar graph on the display. |
| | vertGraph<br>Draws a vertical bar graph on the display. |

A Stream supports the following functions:

| Stream - Function Summary | |
|---|---|
| | print<br>Prints the value of a number, or a string, to the stream. |
| | print_P<br>Prints a string from program memory to the stream. |
| | println<br>Sends a carriage return line feed sequence to the stream to indicate the end of a line of text. |
| int | read<br>Reads the next byte from the device. |
| int | write<br>Writes a single byte to the output stream. |
| size_t | write<br>Writes out a sequence of bytes from a given position in RAM. |
| size_t | write_P<br>Writes out a sequence of bytes from a given position in program memory. |

## VT100 Terminal

Allows you to use your computer, or a real VT100 terminal if you've got one or can still find one, to simulate an LCD display.

Since you will be emulating an LCD display then you wont be able to send key presses back from the PC to the micro controller. But if your project wants to simulate an LCD display, and you don't have one, then this is very useful!

If you are using Project Designer and your board already has a UART that you want to re-use as a VT100 display then click the disable check box next to the UART and then add a VT100 Display which uses the same hardware uart instead.

Since this device allows you to print out info and graphs at various points in the display then its very useful as an 'instrument panel display' as opposed to the usual scrolling text log you normally get with a terminal.

The only other thing you will need to do is make sure that your computer terminal software uses VT100 emulation.

All displays support the following functions (where possible):

| Display - Function Summary | |
|---|---|
| DISPLAY_COLUMN | getNumColumns<br>Returns the total number of columns available. |
| DISPLAY_LINE | getNumLines<br>Returns the total number of lines available. |
| | clear<br>Clear the display and set the cursor to the home position. |
| | home<br>Move the cursor to the home position ie top left corner. |
| | setXY<br>Move the cursor to a given location. |
| | setLineWrap<br>Turn automatic line wrapping on or off. |
| | setAutoScroll<br>Turn auto-scrolling on or off. |

| Display - Function Summary | |
|---|---|
| | setBacklight<br>Turn the display back light on or off (assuming that it has one and that it can be controlled in software). |
| | setBrightness<br>Set the brightness of the display. |
| | setContrast<br>Set the contrast of the display to a number between 0 (least) and 100 (most). |
| | horizGraph<br>Draws a horizontal bar graph on the display. |
| | vertGraph<br>Draws a vertical bar graph on the display. |

A Stream supports the following functions:

| Stream - Function Summary | |
|---|---|
| | print<br>Prints the value of a number, or a string, to the stream. |
| | print_P<br>Prints a string from program memory to the stream. |
| | println<br>Sends a carriage return line feed sequence to the stream to indicate the end of a line of text. |
| int | read<br>Reads the next byte from the device. |
| int | write<br>Writes a single byte to the output stream. |
| size_t | write<br>Writes out a sequence of bytes from a given position in RAM. |
| size_t | write_P<br>Writes out a sequence of bytes from a given position in program memory. |

## Distance

Supports measurement of distances.

### What Are They?

They measure the distance from the device to an obstruction in front of it.

All readings from this library are in 'cm' and so you can swap one device for another in Project Designer. So long as you give the new device the same name as the old device then your project code will not need to be modified.

But note that you may not always get the same results.

Why not?

The devices that use an infra red light beam, like the Sharp sensors, have 'tunnel vision'. When they say there is nothing ahead then what they mean is 'there is sufficient gap for a beam of light to pass through' - and your robot may be slightly wider than a beam of light. If not then let me know and perhaps we can become billionaires together. This is normally circumvented, to a degree, by mounting the sensor on a servo and flicking it from side to side like a 'laser light show'. But note that this could still miss thin objects like a chair leg.

Sonar devices emit a 'blip' of sound and listen for echoes. The width of this beam of sound dictates how 'fuzzy' the result is. Some sonars emit a very directed sound pulse whereas others use a fog-horn approach.

So think of sonars as being pessimistic and fuzzy, and light beams as optimistic and precise.

For 'exact' work you may want to use both - ie when the sonar says there is something ahead then use light beams to scan that area.

If you are using more than one of the same type then you need to be careful to avoid incorrect readings. For example: if you have two sonars sending out 'pings' of sound simultaneously then one sonar may hear the echo of the sound produced by the other sensor.This is called 'ghosting' - and also applies to light beam sensors.

The library already makes sure that a given sensor cannot receive ghost replies from the previous time it was read but if you are using more than one sensor then you may want to add extra delays between accessing each sensor.

### Example

Assuming you have called your sensor 'myDistance' in Project Designer then it can be accessed as follows:-

```
myDistance.read();
cout << "Distance=" << myDistance.getDistance();
```

Or dumped to the standard out destination using:-

```
cout << myDistance;
```

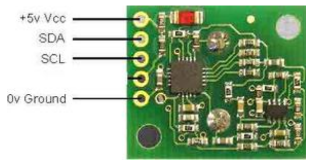
All distance sensors provide the following functions:

| Function Summary | |
|---|---|
| DISTANCE_TYPE | getDistance<br>Returns the distance in whole cm detected by the sensor during the last read(). |

The following devices are supported:

## Devantech SRF02 sonar

Devantech SRF02 sonar.



The sonar uses the I2C bus and also requires a +5v regulated supply. The sonar can measure distances from about 17cm to about 250 cm. The default I2C address is 0xE0 but can be changed to one of 16 values if you know what you are doing!

### Example

Assuming you have called your sensor 'myDistance' in Project Designer then it can be accessed as follows:-

```
myDistance.read();
DISTANCE_TYPE cm = myDistance.getDistance();
cout << "Distance=" << cm;
```

Or dumped to the standard out destination using:-

```
cout << myDistance;
```

All distance sensors provide the following functions:

| Distance - Function Summary | |
|---|---|
| DISTANCE_TYPE | getDistance<br>Returns the distance in whole cm detected by the sensor during the last read(). |

All sensors provide the following functions

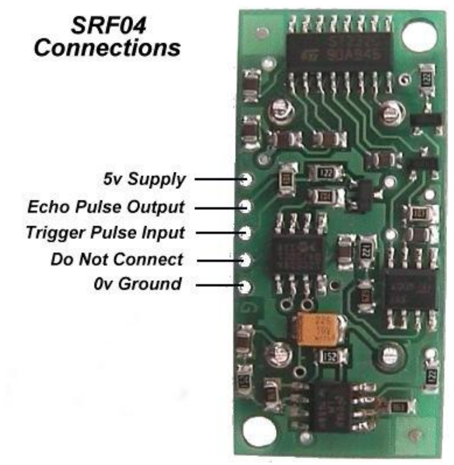
| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output |

## Sensor - Function Summary

| | |
|---|---|
| | stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Devantech SRF04 sonar

Devantech SRF04 sonar.



The sonar can be connected to any two I/O pins but requires a +5v regulated supply. The sonar can measure distances up to about 300 cm. It may be defined using:

According to the datasheet you should not read the sensor within 50mS of taking the previous reading. Otherwise you will get ghost echoes from the previous reading. This library will automatically detect if that happens and, in the event, will return the previous reading. But if you are using more than one sonar then you will need to manually add a suitable delay to stop this sensor receiving ghost echoes from other sonars.

### Example

Assuming you have called your sensor 'myDistance' in Project Designer then it can be accessed as follows:-

```
myDistance.read();
DISTANCE_TYPE cm = myDistance.getDistance();
cout << "Distance=" << cm;
```

Or dumped to the standard out destination using:-

```
cout << myDistance;
```

All distance sensors provide the following functions:

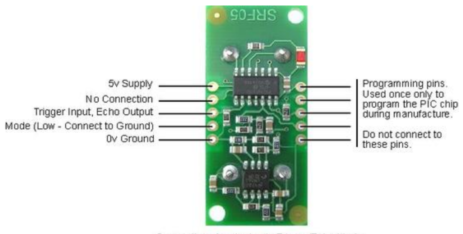
| Distance - Function Summary | |
|---|---|
| DISTANCE_TYPE | getDistance<br>Returns the distance in whole cm detected by the sensor during the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Devantech SRF05 sonar

Devantech SRF05 sonar.


5v Supply
No Connection
Trigger Input, Echo Output
Mode (Low - Connect to Ground)
0v Ground
Programming pins. Used once only to program the PIC chip during manufacture.
Do not connect to these pins.
Connections for single pin Trigger/Echo Mode

The sonar can be connected to any I/O pin but requires a +5v regulated supply. The device can be operated in two different modes. We use 'Mode 2' since that only requires a single pin to drive it - and so you must connect the 'Mode' pin on the board to the ground pin. The sonar can measure distances up to 430 cm.

### Example

Assuming you have called your sensor 'myDistance' in Project Designer then it can be accessed as follows:-

```
myDistance.read();
DISTANCE_TYPE cm = myDistance.getDistance();
cout << "Distance=" << cm;
```

Or dumped to the standard out destination using:-

```
cout << myDistance;
```

According to the datasheet you should not read the sensor within 50mS of taking the previous reading. Otherwise you will get ghost echoes from the previous reading. This library will automatically detect if that happens and, in the event, will return the previous reading. If you are using more than one sonar then you will need to manually add delays to stop this sonar from receiving ghost echoes from the other sonars.

All distance sensors provide the following functions:

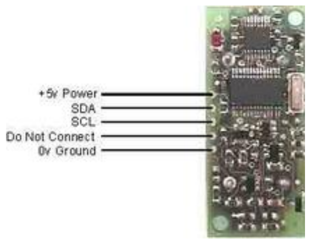
| Distance - Function Summary | |
|---|---|
| DISTANCE_TYPE | getDistance<br>Returns the distance in whole cm detected by the sensor during the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| `boolean` | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| `TICK_COUNT` | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Devantech SRF08 sonar

Devantech SRF08 sonar.



The sonar uses the I2C bus and also requires a +5v regulated supply. The sonar can measure distances from about 17cm to about 600 cm. The default I2C address is 0xE0 but can be changed to one of 16 values if you know what you are doing!

This device also has a Light Dependent Resistor (LDR) that measures the ambient light level. This value has no associated 'unit of measurement' but can be used to distinguish 'lighter' versus 'darker'.

### Example

Assuming you have called your sensor 'myDistance' in Project Designer then it can be accessed as follows:-

```
myDistance.read();
DISTANCE_TYPE cm = myDistance.getDistance();
uint8_t light = myDistance.getLightLevel();
cout << "Distance=" << cm << " Light=" << light;
```

Or dumped to the standard out destination using:-

```
cout << myDistance;
```

This sensor adds the following functions:

| Function Summary | |
|---|---|
| uint8_t | getLightLevel<br>Returns the light level from the SRF08 light sensor. |

All distance sensors provide the following functions:

| Distance - Function Summary | |
|---|---|
| DISTANCE_TYPE | getDistance<br>Returns the distance in whole cm detected by the sensor |

## Distance - Function Summary

| | |
|---|---|
| | during the last read(). |

All sensors provide the following functions

## Sensor - Function Summary

| | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Maxbotix EZ1 sonar

Maxbotix EZ1 sonar.

The sonar can output data in a variety of different formats - but we use the analogue output and so it must be connected from the 'AN' output to an ADC pin and can be driven using the +5v regulated supply. The sonar can measure distances in the range 6 to 254 inches.

### Example

Assuming you have called your sensor 'myDistance' in Project Designer then it can be accessed as follows:-

```
myDistance.read();
DISTANCE_TYPE cm = myDistance.getDistance();
cout << "Distance=" << cm;
```

Or dumped to the standard out destination using:-

```
cout << myDistance;
```

All distance sensors provide the following functions:

| **Distance - Function Summary** | |
|---|---|
| DISTANCE_TYPE | getDistance<br>Returns the distance in whole cm detected by the sensor during the last read(). |

All sensors provide the following functions

| **Sensor - Function Summary** | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |

## Sensor - Function Summary

| | |
|---|---|
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| `TICK_COUNT` | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

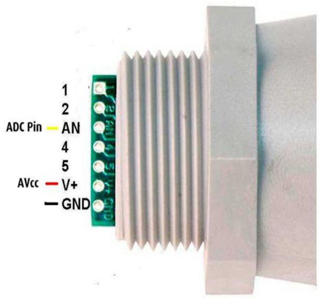
## Maxbotix MB7077 sonar

Maxbotix MB7077 sensor (suitable for use under water).

This sensor can be used in air or in water - although you may need to add extra gloop to make it water proof.

The sonar can output data in a variety of formats - but this library uses the analogue output from the 'AN' pin to an ADC pin on your micro controller.

The device can be powered using either 3.3v or 5v BUT your choice will depend on the reference voltage used by the ADC on your board. ie using 3.3v to power the device from a board that uses 5v as its ADC reference voltage will give wrong readings. But worse: powering the device with 5v from a board that expects a maximum ADC input of 3.3v may fry the ADC unit on your board.



The device can measure up to about 700cm with a 5v supply and up to about 600cm with a 3.3v supply.

The returned range from the device is based on the speed of sound and hence will vary depending upon the medium where it lives. For example: sound travels 4.3 times faster under water than it does through air. For this reason: you can specify whether the device is normally under water in Project Designer.

If your robot is amphibious then you can change the 'inWater' member variable at runtime to reflect the current environment for the device using "*setInWater*" (see page 448) .

## Example

Assuming you have called your sensor 'myDistance' in Project Designer then it can be accessed as follows:-

```
myDistance.read();
DISTANCE_TYPE cm = myDistance.getDistance();
cout << "Distance=" << cm;
```

Or dumped to the standard out destination using:-

```
cout << myDistance;
```

This sensor can be used above, or below, water and so provides the following functions:

| Function Summary | |
|---|---|
| boolean | isInWater<br>Return TRUE if you have told the sensor that it is underwater or FALSE if not. |
| | setInWater<br>Tell the sensor whether it is underwater or not. |

All distance sensors provide the following functions:

| Distance - Function Summary | |
|---|---|
| DISTANCE_TYPE | getDistance<br>Returns the distance in whole cm detected by the sensor during the last read(). |

All sensors provide the following functions

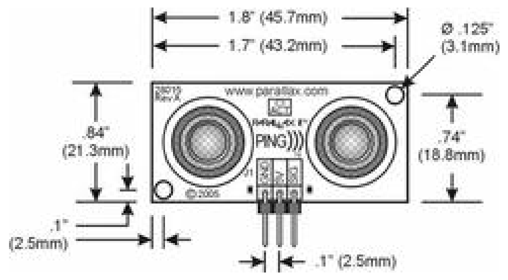
| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |

## Sensor - Function Summary

| | |
|---|---|
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## PING Sonar

Parallax PING sonar.



The sonar can be connected to any I/O pin but requires a +5v regulated supply. The sonar can measure distances in the range 2 to 300 cm.

### Example

Assuming you have called your sensor 'myDistance' in Project Designer then it can be accessed as follows:-

```
myDistance.read();
DISTANCE_TYPE cm = myDistance.getDistance();
cout << "Distance=" << cm;
```

Or dumped to the standard out destination using:-

```
cout << myDistance;
```

According to the datasheet you should not read the sensor within 200µS of taking the previous reading. Otherwise you will get ghost echoes from the previous reading. This library will automatically detect if that happens and, in the event, will return the previous reading. If you are using multiple sonars then you will need to manually add delays between reading different sonars to avoid one receiving a ghost echo from another sonar.

All distance sensors provide the following functions:

| Distance - Function Summary | |
|---|---|
| DISTANCE_TYPE | getDistance<br>Returns the distance in whole cm detected by the sensor during the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | **read**<br>Read the sensor and store its current values. |
| | **dump**<br>Dump the last read sensor values to the standard output device. |
| | **dumpTo**<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | **getTimeLastRead**<br>Returns the value of the clock at the time when the sensor was last read. |

## Sharp GP2

Sharp Infra Red distance sensors.

There are a number of different sensors but they are all used in the same way. The difference between them is the usable range over which they can detect objects. The devices work by shining a beam of light which is reflected back from an object. The further away the object is then the more it displaces the light to the sides. Therefore: you will notice that the devices which can detect further distances are physically larger because the 'eye' which sees the reflection needs to be further away from the transmitter. Equally the transmitter needs to be more powerful and is therefore larger.

### Note

Because these devices uses a beam of light then they are not very good at detecting thin objects - like chair legs. This happens because the beam of light may move from one side of the leg to the other and so therefore it misses the fact that the leg is there. For a less precise detector you should consider using a sonar.

### Noise

The devices output a voltage and so should be connected be to an ADC pin and use the +5V regulated supply to power them. A word of caution: these devices are inherently noisy and require spikes of current. The noise can be reduced by connecting the black casing to ground (yes - it looks like plastic but it is actually conductive). If your robot has a metal shell then bolt the sensor to the shell and connect the shell to ground. The current spikes can be minimised by fitting a capacitor of around 10uF close to the device - preferably by soldering an SMD capacitor into the device itself.

You will still find some spikes in the readings and so, if these spikes confuse your robot, then you should consider writing a low pass filter in software to filter out these spikes. But a word of caution: that spike may actually be correct ie something has suddenly come into view.

The range of devices we support are as follows:-

GP2D12 This measures between 10cm and 80cm

GP2D120 This measures between 4cm and 30cm

GP2D15 This measures between 10cm and 80cm

GP2Y0A02YK This measures between 20cm and 150cm

GPY0A21YK This measures between 4cm and 30cm

GP2Y0A02YK0F This measures 20cm to 150cm

GP2Y0A710K0F This measures 100cm to 550cm

GP2Y0A700K0F This measures 100cm to 550cm

Note that the larger the maximum distance then the more peak current these devices will require. It is common for them to require a peak current of 0.3 amps !

## Example

Assuming you have called your sensor 'myDistance' in Project Designer then it can be accessed as follows:-

```
myDistance.read();
DISTANCE_TYPE cm = myDistance.getDistance();
cout << "Distance=" << cm;
```

Or dumped to the standard out destination using:-

```
cout << myDistance;
```

All distance sensors provide the following functions:

| Distance - Function Summary | |
|---|---|
| DISTANCE_TYPE | getDistance<br>Returns the distance in whole cm detected by the sensor during the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |

## Sensor - Function Summary

| | |
|---|---|
| `TICK_COUNT` | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Sharp GP2 5 LED

Sharp Infra Red 'wide field' distance sensors.

These are single boxed units but contain five LEDs pointing at various angles (often 25%). Each LED can be selected individually and a single ADC pin is used to measure the result for that LED.

There are a number of different sensors but they are all used in the same way. The difference between them is the usable range over which they can detect objects.

The devices work by shining a beam of light which is reflected back from an object. The further away the object is then the more it displaces the light to the sides. Therefore: you will notice that the devices which can detect further distances are physically larger because the 'eye' which sees the reflection needs to be further away from the transmitter. Equally the transmitter needs to be more powerful and is therefore larger.

### Note

Because these devices uses a beam of light then they are not very good at detecting thin objects - like chair legs. This happens because the beam of light may move from one side of the leg to the other and so therefore it misses the fact that the leg is there. For a less precise detector you should consider using a sonar.

### Noise

The devices output a voltage and so should be connected be to an ADC pin and use the +5V regulated supply to power them. A word of caution: these devices are inherently noisy and require spikes of current. The noise can be reduced by connecting the black casing to ground (yes - it looks like plastic but it is actually conductive). If your robot has a metal shell then bolt the sensor to the shell and connect the shell to ground. The current spikes can be minimised by fitting a capacitor of around 10uF close to the device - preferably by soldering an SMD capacitor into the device itself.

You will still find some spikes in the readings and so, if these spikes confuse your robot, then you should consider writing a low pass filter in software to filter out these spikes. But a word of caution: that spike may actually be correct ie something has suddenly come into view.

Here are some examples:-

GP2Y3A002K0F This measures between 20cm and 150cm and has a total arc of 25°

GP2Y3A003K0F This measures between 40cm and 300cm and has a total arc of 25°

Note that the larger the maximum distance then the more peak current these devices will require. It is common for them to require a peak current of 0.3 amps !

Since these devices take around 25ms to obtain a single LED reading (ie 125ms for a full scan of all LEDs) then they are quite slow and so this library reads one LED at a time. Note that read() can return a TRUE or FALSE. When reading these devices it will only return TRUE once a valid reading has been taken for the current LED. If a FALSE is returned then no sensor data has changed.

## Example

Assuming you have called your sensor 'myDistance' in Project Designer then it can be accessed as follows:-

```
if(myDistance.read() == TRUE){
    // We have finished reading one LED and it has started reading the next one
    // The value in getBeamNumber() represents the next LED
    // we are now starting to read so the
    // 'just-read' LED is the previous one
    if(myDistance.getBeamNumber() == 0){
        // We have just completed reading all beams
        for(uint8_t beam = 0; beam < 5; beam++){
             cout << "Beam #" << beam << "=" << myDistance.getBeam(beam);
        }
        // The overall 'minimum' distance
        cout << "Minimum:" << myDistance.getDistance();
    }
}else{
    // The sensor is busy reading
}
```

Or dumped to the standard out destination using:-

```
cout << myDistance;
```

This sensor adds the following functions:

| Function Summary | |
|---|---|
| uint8_t | getBeamNumber<br>Returns the current beam number that is being evaluated. |

| Function Summary | |
|---|---|
| DISTANCE_TYPE | getBeam<br>Returns the distance, in cm, for the specified light beam. |

All distance sensors provide the following functions:

| Distance - Function Summary | |
|---|---|
| DISTANCE_TYPE | getDistance<br>Returns the distance in whole cm detected by the sensor during the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Encoder

Supports measurement of encoders in ticks.

All readings from this library are in 'ticks'.

### What are encoders?

They measure the clockwise and counter-clockwise rotation of an axle. For our purposes they are normally mounted onto motors or wheels.

Each encoder emits a given number of 'ticks' per revolution with the ticks increasing if the axle is turned one way and decreased if turning the other way.

If we know how many ticks the axle has moved, and the number of ticks the encoder produces per 360 degrees, then we can work out how many degrees the axle has turned. Then, by knowing the wheel circumference, we can convert this into a distance travelled. But beware of slippage: if the robot is on ice, or a steep hill, then the wheel may turn and generate the ticks but the robot may not have moved. Slippage can be minimised by changing acceleration gradually (ie avoid wheel spin unlike those drag racing cars!), by having 'gripping' wheels etc etc.

### Other Uses

There are other uses for encoders - including non-robot stuff. I bet you've had one of those old scratchy radios where every time you twiddle the volume you hear a horrible scratching noise. That is because they use a potentiometer and that noise is caused because it involves mechanical contacts which eventually wear out the carbon track on the potentiometer. More modern designs use encoders for volume control as there is no mechanical contact - so less noise and nothing to 'wear out'.

### Example

Assuming you have added an encoder within Project Designer and called in 'myEncoder' then you can access the current number of ticks as follows:-

```
// Read and store the current value
myEncoder.read();
// You can now access the returned value
ENCODER_TYPE ticks = myEncoder.getTicks();
// Or dump it the standard output stream
cout << myEncoder;
```

### Encoder interpolation

This library allows you to add some additional functionality to your encoders called interpolation.

If 'interpolation' is enabled then two additional pieces of information can be accessed following the read().

```
TICK_COUNT t1 = myEncoder.getTimeSinceLastTick();
TICK_COUNT t2 = myEncoder.getDurationOfLastTick();
```

The 'timeSinceLastTick' is the number of µS since we last a received a tick; and durationOfLastTick is the number of µS between the previous two ticks.

This means that, as a rough approximation, the 'durationOfLastTick' can be used to guess the current speed of the motor. ie if it has a value of 1000µS and the encoder is giving 200 ticks per revolution. Then a full revolution would currently require 1000 x 200 = 200,000µS = 0.2 seconds. Hence the motor is rotating at 300 rpm. Given that we know the wheel circumference then we can turn this into a ground speed.

Obviously this assumes that the motor speed is fairly constant as it is just calculated from the last 'tick'.

The 'timeSinceLastTick' is useful if your encoder only has a small number of stripes but the wheel circumference is quite high. In this case a single 'tick' can represent a significant distance. So this parameter allows you to estimate the current fraction of a tick.

For example: if 'durationOfLastTick' is 1000µS and the 'timeSinceLastTick' is 500µS then we can estimate that the wheel has turned by a further 0.5 of a 'tick'.

The 'interpolation' can be turned on and off within Project Designer and also at runtime by calling the 'setInterpolating()' function.

When interpolation is turned on then there is extra processing required for each tick and so the maximum number of ticks per second is reduced (see later for metrics). However: since interpolation is only generally useful for low resolution encoders then this should not be an issue.

All encoders provide the following functions:

| Function Summary | |
|---|---|
| ENCODER_TYPE | getTicks<br>Returns the number of ticks, as an ENCODER_TYPE, at the time when you performed the last read(). |
| | subtract<br>Subtract a given value from the encoder counter. |
| uint16_t | ticksPerRevolution<br>Returns the number of counter ticks generated for each 360° revolution. |
| | setInterpolating<br>Turn interpolation on or off. |

| Function Summary | |
|---|---|
| boolean | isInterpolating<br>Returns whether or not the encoder is using interpolation. |
| TICK_COUNT | getTimeSinceLastTick<br>Returns the number of μS between your call to read() and when the last tick was received. |
| TICK_COUNT | getDurationOfLastTick<br>Returns the duration, in μS, of the last received tick. |

The following devices are supported:

| | |
|---|---|
| Fast Quadrature | 185 |
| Quadrature | 188 |

## Fast Quadrature

Support for a generic quadrature encoder using external interrupts.

Most modern ATmel micro controllers have some external interrupts (between 2 and 8) but some of these are put on pins used by UARTs, I2C, PWM pins etc so often there are only a few of them available for use. However they are fast to process making them suitable for high resolution encoders.

A quadrature encoder provides two signals called Channel A and Channel B which are 90 degrees out of phase. When channel A changes state then by reading channel B we can tell whether the device is rotating clockwise or anti-clockwise.

So channel A provides a pulse and channel B tells us whether to increment or decrement a counter so that we know how far the device has rotated.

Assuming an encoder disk with 32 stripes is attached to a drive motor then we will get 64 pulses per revolution. Obviously the distance travelled will depend on the diameter of the wheel that is attached to the motor drive shaft.

If you have enough INT pins then you can double the output of the encoder by ticking the 'Double the ticks per revolution' setting in Project Designer. This requires two INT pins per encoder - ie for 32 stripes then this will generate 128 pulses per revolution.

Like all encoders we can read the encoder using the read() function to store the current value of the counter. We can then access this value through the getTicks() function.

Assuming that the wheel is just rotating continuously then the counter will eventually overflow back to 0 and the frequency of this will depend on how many stripes the encoder has and on how fast the motor is rotating.

Note that you cannot set the counter to a given value - instead you must use the subtract(ENCODER_TYPE count) function to reduce the value. The reason being that otherwise we may 'miss' some pulses. Assuming your encoder is called 'myEncoder' then performing:

```
// read the quadrature encoder
myEncoder.read();
// get the value from the last read
ENCODER_TYPE ticks = myEncoder.getTicks();
// reduce the value by the same amount
myEncoder.subtract(ticks);
```

will change the counter to 0 if the motor is not rotating.

## Metrics

If encoder interpolation is switched **off** then the library can cope with up to about 9,000 ticks per second per MHz of processor speed. ie for an Axon running at 16MHz then: 16*9,000 = 144,000 ticks per second.

If encoder interpolation is switched **on** then the library can cope with up to about 2,900 ticks per second per MHz of processor speed. ie for an Axon running at 16MHz then: 16*2,900 = 46,400 ticks per second.

These numbers are 'per encoder' - so if you have 4 encoders then divide the above figures by 4.

Example: assume we have a 16MHz processor and 4 encoders each generating up to 3,000 ticks per second and interpolation is turned off for all of them. Then: 4 encoders x 3,000 ticks = 12,000 ticks per second. 100% cpu = 144,000 ticks, so 12,000 ticks=8.3%. So your program will be spending 8.3% of its time just processing the encoder ticks. Obviously as the figure approaches 100% then your main code will run more slowly. Exceeding 100% means that everything will break!

All encoders provide the following functions:

| **Encoder - Function Summary** | |
|---|---|
| `ENCODER_TYPE` | getTicks<br>Returns the number of ticks, as an ENCODER_TYPE, at the time when you performed the last read(). |
| | subtract<br>Subtract a given value from the encoder counter. |
| `uint16_t` | ticksPerRevolution<br>Returns the number of counter ticks generated for each 360° revolution. |
| | setInterpolating<br>Turn interpolation on or off. |
| `boolean` | isInterpolating<br>Returns whether or not the encoder is using interpolation. |
| `TICK_COUNT` | getTimeSinceLastTick<br>Returns the number of µS between your call to read() and when the last tick was received. |
| `TICK_COUNT` | getDurationOfLastTick<br>Returns the duration, in µS, of the last received tick. |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Quadrature

Support for a generic quadrature encoder using pin change interrupts.

Most modern ATmel micro controllers have lots of pin change interrupts (up to around 24) and so they are more flexible however, due to the chip design, they are slow to process. So if you find your are missing ticks because you have a high resolution encoder then you may want to consider using the 'Fast Quadrature' device instead.

A quadrature encoder provides two signals called Channel A and Channel B which are 90 degrees out of phase. When channel A changes state then by reading channel B we can tell whether the device is rotating clockwise or anti-clockwise.

So channel A provides a pulse and channel B tells us whether to increment or decrement a counter so that we know how far the axle has rotated.

Assuming an encoder disk with 32 stripes is attached to a drive motor then we will get 64 pulses per revolution. Obviously the distance travelled will depend on the diameter of the wheel that is attached to the motor drive shaft.

Like all encoders we can read the encoder using the read() function to store the current value of the counter. We can then access this value through the getTicks() function.

Assuming that the wheel is just rotating continuously then the counter will eventually overflow back to 0 and the frequency of this will depend on how many stripes the encoder has and on how fast the motor is rotating.

Note that you cannot set the counter to a given value - instead you must use the subtract(ENCODER_TYPE count) function to reduce the value. The reason being that otherwise we may 'miss' some pulses. Assuming your encoder is called 'myEncoder' then performing:

```
// read the quadrature encoder
myEncoder.read();
// get the value from the last read
ENCODER_TYPE ticks = myEncoder.getTicks();
// reduce the value by the same amount
myEncoder.subtract(ticks);
```

will change the counter to 0 if the motor is not rotating.

### Metrics

If encoder interpolation is switched **off** then the library can cope with up to about 2,000 ticks per second per MHz of processor speed. ie for an Axon running at 16MHz then: 16*2,000 = 32,000 ticks per second.

If encoder interpolation is switched **on** then the library can cope with up to about 1,300 ticks per second per MHz of processor speed. ie for an Axon running at 16MHz then: 16*1,300 = 20,800 ticks per second.

These numbers are 'per encoder' - so if you have 4 encoders then divide the above figures by 4.

Example: assume we have a 16MHz processor and 4 encoders each generating up to 3,000 ticks per second and interpolation is turned off for all of them. Then: 4 encoders x 3,000 ticks = 12,000 ticks per second. 100% cpu = 32,000 ticks, so 12,000 ticks=37.5%. So your program will be spending 37.5% of its time just processing the encoder ticks. Obviously as the figure approaches 100% then your main code will run more slowly. Exceeding 100% means that everything will break!

All encoders provide the following functions:

| **Encoder - Function Summary** | |
|---|---|
| `ENCODER_TYPE` | getTicks<br>Returns the number of ticks, as an ENCODER_TYPE, at the time when you performed the last read(). |
| | subtract<br>Subtract a given value from the encoder counter. |
| `uint16_t` | ticksPerRevolution<br>Returns the number of counter ticks generated for each 360° revolution. |
| | setInterpolating<br>Turn interpolation on or off. |
| `boolean` | isInterpolating<br>Returns whether or not the encoder is using interpolation. |
| `TICK_COUNT` | getTimeSinceLastTick<br>Returns the number of µS between your call to read() and when the last tick was received. |
| `TICK_COUNT` | getDurationOfLastTick<br>Returns the duration, in µS, of the last received tick. |

All sensors provide the following functions

| **Sensor - Function Summary** | |
|---|---|
| `boolean` | read<br>Read the sensor and store its current values. |

| Sensor - Function Summary | |
|---|---|
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## GPS

Adds support for GPS devices.

GPS devices normally provide information on position (latitude and longitude), altitude and time of day by locking on to a number of satellites and using triangulation.

The greater the number of satellites, and the further they are away from each other, then the more accurate the calculations.

When a GPS is turned on having been powered down for a while (cold boot) then it can take several minutes for it to lock on to enough satellites to start returning valid readings. This delay tends to be much shorter after a warm-boot ie if you only turn the power off for a few seconds.

Note that the accuracy can be as rough as 10m or so. This is obviously not very useful if your are trying to track your robots location in, say, a small maze - but is useful for robots that cover a wider distance.

A further reduction in accuracy is caused by the avr-gcc compiler implementation of floating point numbers which, being only 4 bytes long, can only store a small number of significant digits (ie values get rounded).

A useful side effect of a GPS is the time-of-day, called the Fix Time, which can be useful for generating dates and times in logging applications.

There are a number of standards for how GPS devices communicate but the most widely implemented standard is called NMEA. This sends out messages, normally at 4800 baud and one message every one second, over a serial port.

### Example

Assuming you have created a gps device called myGPS in Project Designer and you want to print out any valid longitude and latitude values then you could write:-

```
// Read the GPS
myGPS.read();

// Check if valid and position changed
if( myGPS.isValid() && myGPS.isPositionUpdated()){
    // Dump the GPS values
    cout << myGPS;
}
```

All GPS devices provide the following functions:

| Function Summary | |
|---|---|
| boolean | isValid<br>Returns whether or not the GPS has got a valid fix position following a read(). |
| double | getFixTime<br>Returns the current fix time from the last read() in the format HHMMSS. |
| double | getLongitudeDegrees<br>Returns the current longitude in degrees from the last read(). |
| double | getLongitudeRadians<br>Returns the current longitude in radians from the last read(). |
| double | getLatitudeDegrees<br>Returns the current latitude in degrees from the last read(). |
| double | getLatitudeRadians<br>Returns the current latitude in radians from the last read(). |
| double | getTrackDegrees<br>Returns the current track in degrees from the last read(). |
| double | getTrackRadians<br>Returns the current track in radians from the last read(). |
| double | getSpeedKnots<br>Returns the current speed in knots from the last read(). |
| double | getSpeedCMperSecond<br>Returns the current speed in cm per second from the last read(). |
| double | getSpeedMPH<br>Returns the current speed in miles per hour from the last read(). |
| int | getNumSatellites<br>Returns the current number of satellites used to create the fix from the last read(). |
| double | getAltitudeMeter<br>Returns the current altitude above sea level in metres. |
| boolean | isUpdated<br>Indicates whether the GPS has processed any new |

## Function Summary

| | |
|---|---|
| | messages during the last read(). |
| boolean | isFixTimeUpdated<br>Indicates whether the GPS has processed any new messages containing a fix time during the last read(). |
| boolean | isLongitudeUpdated<br>Indicates whether the GPS has processed any new messages containing a longitude during the last read(). |
| boolean | isLatitudeUpdated<br>Indicates whether the GPS has processed any new messages containing a latitude during the last read(). |
| boolean | isPositionUpdated<br>Indicates whether the GPS has processed any new messages containing a latitude and/or a latitude during the last read(). |
| boolean | isSpeedUpdated<br>Indicates whether the GPS has processed any new messages containing a speed during the last read(). |
| boolean | isSatellitesUpdated<br>Indicates whether the GPS has processed any new messages containing the number of satellites during the last read(). |
| boolean | isAltitudeUpdated<br>Indicates whether the GPS has processed any new messages containing the altitude during the last read(). |
| boolean | isTrackUpdated<br>Indicates whether the GPS has processed any new messages containing the track during the last read(). |

The following devices are supported:

| | |
|---|---|
| GPS NMEA | 194 |

## GPS NMEA

A GPS sensor that interprets NMEA messages.

The sensor must be connected to a GPS device that sends NMEA messages over a serial link. You will therefore need to connect it via a UART (either a software UART or a hardware UART). The UART baud rate should be set by you - the standard value is 4800 baud but check the datasheet for your specific device.

### Example

Assuming you have called your GPS device 'myGPS' in Project Designer.

```
// This routine is called repeatedly - its your main loop
TICK_CONTROL appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    // Read the sensor
    myGPS.read();

    // Dump it to standard output
    cout << myGPS;

    return 0; // no delay - go at full speed
}
```

All GPS devices provide the following functions:

| Gps - Function Summary | |
|---|---|
| boolean | isValid<br>Returns whether or not the GPS has got a valid fix position following a read(). |
| double | getFixTime<br>Returns the current fix time from the last read() in the format HHMMSS. |
| double | getLongitudeDegrees<br>Returns the current longitude in degrees from the last read(). |
| double | getLongitudeRadians<br>Returns the current longitude in radians from the last read(). |
| double | getLatitudeDegrees<br>Returns the current latitude in degrees from the last read(). |
| double | getLatitudeRadians<br>Returns the current latitude in radians from the last read(). |

## Gps - Function Summary

| | |
|---|---|
| double | getTrackDegrees<br>Returns the current track in degrees from the last read(). |
| double | getTrackRadians<br>Returns the current track in radians from the last read(). |
| double | getSpeedKnots<br>Returns the current speed in knots from the last read(). |
| double | getSpeedCMperSecond<br>Returns the current speed in cm per second from the last read(). |
| double | getSpeedMPH<br>Returns the current speed in miles per hour from the last read(). |
| int | getNumSatellites<br>Returns the current number of satellites used to create the fix from the last read(). |
| double | getAltitudeMeter<br>Returns the current altitude above sea level in metres. |
| boolean | isUpdated<br>Indicates whether the GPS has processed any new messages during the last read(). |
| boolean | isFixTimeUpdated<br>Indicates whether the GPS has processed any new messages containing a fix time during the last read(). |
| boolean | isLongitudeUpdated<br>Indicates whether the GPS has processed any new messages containing a longitude during the last read(). |
| boolean | isLatitudeUpdated<br>Indicates whether the GPS has processed any new messages containing a latitude during the last read(). |
| boolean | isPositionUpdated<br>Indicates whether the GPS has processed any new messages containing a latitude and/or a latitude during the last read(). |

| Gps - Function Summary | |
|---|---|
| boolean | isSpeedUpdated<br>Indicates whether the GPS has processed any new messages containing a speed during the last read(). |
| boolean | isSatellitesUpdated<br>Indicates whether the GPS has processed any new messages containing the number of satellites during the last read(). |
| boolean | isAltitudeUpdated<br>Indicates whether the GPS has processed any new messages containing the altitude during the last read(). |
| boolean | isTrackUpdated<br>Indicates whether the GPS has processed any new messages containing the track during the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Gyro

Supports gyros/gyroscopes.

Gyros normally come in either 2 axis, or 3 axis versions. Sometimes you can buy a combo board that contains gyro and, say, an accelerometer. I don't directly support these combos so you need to declare the two individual sensors.

So what do they do? Gyros measure rotational velocity in degrees per second. This could be used to monitor whether a robot biped is falling over, so that you can compensate, or if measuring the rotation of an axle then, knowing the wheel circumference, you can work out the current speed of the robot.

These devices either use one ADC pin for each axis, or provide an I2C interface but you will need to check the individual data sheets to see what power supply they need.

All gyros return one value per axis and this value is in 'degrees per second'. Each value may be positive or negative and a two axis device will set the Z value to zero.

### Example

Assuming you have called the gyro 'myGyro' in Project Designer then:-

```
// Read the gyro and save the values
myGyro.read();

// Retrieve the x,y,z values
GYRO_TYPE x = myGyro.getX();
GYRO_TYPE y = myGyro.getY();
GYRO_TYPE z = myGyro.getZ();

// Print them out
cout << "X=" << x << " Y=" << y << " Z=" << z;

// Or dump it out
cout << myGyro;
```

All gyros provide the following functions:

| Function Summary | |
|---|---|
| GYRO_TYPE | getX<br>Returns the X axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getY<br>Returns the Y axis rotational velocity in degrees per second |

## Function Summary

| | |
|---|---|
| | at the time of the last read(). |
| GYRO_TYPE | getZ<br>Returns the Z axis rotational velocity in degrees per second at the time of the last read(). |

The following devices are supported:

## IDG300

The IDG300 is a 2 axis gyro that can measure rotations up to 500 degrees per second.

Data sheet: http://www.sparkfun.com/datasheets/Components/IDG-300_Datasheet.pdf

Requires 2 ADC channels.

Note that the device requires a 3v supply.

All gyros provide the following functions:

| Gyro - Function Summary | |
|---|---|
| GYRO_TYPE | getX<br>Returns the X axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getY<br>Returns the Y axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getZ<br>Returns the Z axis rotational velocity in degrees per second at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## IDG500

The IDG500 is a 2 axis gyro that can measure rotations up to 500 degrees per second.

Data sheet: http://www.sparkfun.com/datasheets/Components/SMD/Datasheet_IDG500.pdf

Requires 2 ADC channels.

Note that the device requires a 3v supply.

The device operates in two modes:-

- FAST - can measure faster rotation speeds, up to 500 degrees per second, but with less accuracy, or
- SLOW - can measure rotation speeds up to 110 degrees per second with greater accuracy.

Note that this is not a 'software setting' and so this library cannot automatically choose the correct value. It actually depends on which pin on the device you use to connect to the ADC. The 'SLOW' option is referred to in the data sheet as the '4.5' output (coz its like the 'FAST' speed divided by 4.5).

All gyros provide the following functions:

| Gyro - Function Summary | |
|---|---|
| `GYRO_TYPE` | getX<br>Returns the X axis rotational velocity in degrees per second at the time of the last read(). |
| `GYRO_TYPE` | getY<br>Returns the Y axis rotational velocity in degrees per second at the time of the last read(). |
| `GYRO_TYPE` | getZ<br>Returns the Z axis rotational velocity in degrees per second at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| `boolean` | read<br>Read the sensor and store its current values. |

## Sensor - Function Summary

| | |
|---|---|
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## ITG3200 3 axis Gyro

The ITG3200 is a 3 axis gyro that can measure rotations up to 2,000 degrees per second via an I2C interface. It also contains a temperature sensor.

A breakout board is available from SparkFun: http://www.sparkfun.com/products/9801

That page also has a link to the complete datasheet.

Note that the device requires a supply in the range 2.1v to 3.6v and draws approximately 6.5mA. This voltage should be applied to both the VDD pin and VLOGIC pin (also called VIO).

The I2C address of the device can be configured to be either 0xD2 or 0xD0 depending on whether the AD0 pin is pulled high or low.

The CLK pin is not used and so should be connected to ground.

The INT output pin can be used to signal to your microprocessor when there is new data available. This is not supported by WebbotLib by default, ie WebbotLib always reads the current values, but you could always connect it to a digital input pin and use it to signal whether or not it is worth reading new values or not.

The remaining pins: SCL, SDA and GND form the I2C bus.

The ITG3200 provides the following functions:

| Function Summary | |
|---|---|
| TEMPERATURE_TYPE | getCelsius<br>Returns the ambient temperature in Celsius at the time of the last read(). |
| TEMPERATURE_TYPE | getFahrenheit<br>Returns the ambient temperature in Fahrenheit at the time of the last read(). |

All gyros provide the following functions:

| Gyro - Function Summary | |
|---|---|
| GYRO_TYPE | getX<br>Returns the X axis rotational velocity in degrees per second at the time of the last read(). |

| Gyro - Function Summary | |
|---|---|
| GYRO_TYPE | getY<br>Returns the Y axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getZ<br>Returns the Z axis rotational velocity in degrees per second at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## LPR530AL 2 axis

The LPR530AL is a 2 axis gyro that can measure rotations up to 1200 degrees per second in the x and y axes.

Data sheet:http://www.sparkfun.com/datasheets/Sensors/IMU/lpr530al.pdf

Requires 2 ADC channels.

Note that the device requires a 3.3v supply.

The device can measure fast rotations up to 1200 degrees per second; or slower rotation up to 300 degrees per second with greater accuracy. The device provides different output pins for the fast or slow options.

All gyros provide the following functions:

| Gyro - Function Summary | |
|---|---|
| GYRO_TYPE | getX<br>Returns the X axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getY<br>Returns the Y axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getZ<br>Returns the Z axis rotational velocity in degrees per second at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |

## Sensor - Function Summary

| | |
|---|---|
| `TICK_COUNT` | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## LY530AL 1 axis

The LY530AL is a single axis gyro that can measure yaw rotations up to 1200 degrees per second.

Data sheet: http://www.sparkfun.com/datasheets/Sensors/IMU/LY530ALH.pdf

Requires 1 ADC channels.

Note that the device requires a 3.3v supply.

The device operates in two modes:-

- FAST - can measure faster rotation speeds, up to 1200 degrees per second, but with less accuracy, or
- SLOW - can measure rotation speeds up to 300 degrees per second with greater accuracy.

Note that this is not a 'software setting' and so this library cannot automatically choose the correct value. It actually depends on which pin on the device you use to connect to the ADC. The 'SLOW' option is referred to in the data sheet as the '4.5' output (coz its like the 'FAST' speed divided by 4.5).

All gyros provide the following functions:

| **Gyro - Function Summary** | |
|---|---|
| `GYRO_TYPE` | getX<br>Returns the X axis rotational velocity in degrees per second at the time of the last read(). |
| `GYRO_TYPE` | getY<br>Returns the Y axis rotational velocity in degrees per second at the time of the last read(). |
| `GYRO_TYPE` | getZ<br>Returns the Z axis rotational velocity in degrees per second at the time of the last read(). |

All sensors provide the following functions

| **Sensor - Function Summary** | |
|---|---|
| `boolean` | read<br>Read the sensor and store its current values. |

## Sensor - Function Summary

| | |
|---|---|
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| `TICK_COUNT` | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## LPR530AL 2 axis plus LY530ALH 1 axis

This is a combination (in software as there is no single device) of the LPR530AL 2 axis device and LY530ALH 1 axis device into a logical 3 axis gyro.

Requires 3 ADC channels.

Note that the devices require a 3.3v supply.

The devices can measure fast rotations up to 1200 degrees per second; or slower rotation up to 300 degrees per second with greater accuracy. The devices provide different output pins for the fast or slow options.

All gyros provide the following functions:

| Gyro - Function Summary | |
|---|---|
| GYRO_TYPE | getX<br>Returns the X axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getY<br>Returns the Y axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getZ<br>Returns the Z axis rotational velocity in degrees per second at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |

## Sensor - Function Summary

| | |
|---|---|
| `TICK_COUNT` | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Humidity

Supports humidity measurement.

All readings from this library are as a 'percentage'

### Example

Assuming your sensor is called 'myHumidity' in Project Designer then:-

```
// Read sensor and store values
myHumidity.read();
// Get the humidity
HUMIDITY_TYPE h = myHumidity.getHumidity();
// Print it
cout << "Humidity:" << h;
```

All humidity sensors provide the following functions:

| Function Summary | |
|---|---|
| HUMIDITY_TYPE | getHumidity<br>Returns the humidity as a percentage at the time of the last read(). |

The following devices are supported:

| | |
|---|---|
| Phidget Humidity Sensor | 211 |

## Phidget Humidity Sensor

Phidget humidity sensor.

This must be connected to an ADC pin.

All humidity sensors provide the following functions:

| Humidity - Function Summary | |
|---|---|
| HUMIDITY_TYPE | getHumidity<br>Returns the humidity as a percentage at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

# I2C

Provides support for communicating with other I2C slave devices/sensors.

Here is a somewhat simplistic view of I2C to save you reading the data sheet.

I2C allows you to connect one-or-many I2C devices onto the bus. Each I2C device has its own unique address (like a telephone number). The processor can only talk to one device at a time - in the same way as you have a list of phone numbers but you can only talk to one person at a time (before anyone asks - there is an I2C version of a 'conference call' but let's keep it simple!

## I2C Master

This option allows you to access I2C sensors, displays, servo controllers etc.

## I2C Slave

Allows your board to become a slave which can then be queried by a master robot controller board. This allows you to break a large project into lots of smaller boards that are responsible for a specific job - freeing your main board to do the overall control.

The following devices are supported:

## Generic I2C Device

If the I2C device you want to add is not directly supported by this library then you can use this type in Project Designer instead.

All that is required is the I2C address of the device.

You will then be able to use the I2C functions of the library to communicate with the device.

All I2C devices support the following functions:

**I2cDevice - Function Summary**

| | |
|---|---|
| `boolean` | readRegisters<br>A complete communication session to read one or more sequential registers from the device. |
| `boolean` | receive<br>Performs a complete communication with a slave device to receive a number of bytes. |
| `boolean` | writeRegisters<br>Performs a complete communication session with a device to write to a sequential series of registers. |
| `boolean` | writeRegister<br>Performs a complete communication session with a device to write to a value to a single register. |
| `boolean` | send<br>Performs a complete communication session with a device to write a number of bytes. |
| `boolean` | transfer<br>Performs a complete communication with a slave device to write a number of bytes and then read a number of bytes. |

## I2C Master

To query other sensors and devices you will need to add an 'I2C Master' to your Project Designer project. From within this you can build up the list of all the devices you have got connected to the bus. Think of this as a 'phone directory'.

At this point you may get an error if the devices don't have unique addresses. The easiest solution is to add one, or more, software I2C busses so that you can move conflicting devices onto different busses to resolve the conflict and end up with several busses each with its own set of now unique addresses.

The I2C sub-system is set up to use 100kbps (kilo bits per second) as the default communication speed. If you wish to change this then you can call "*setBitRate*" (see page 550) in appInitSoftware - ie after the bus has been initialised.

The I2C interface requires pull-up resistors on the SCL and SDA lines. For a hardware bus the library achieves this by using the internal pull-up resistors on the processor pins however for a software bus you **must** add the extra resistors (4.7k should work).

If the device you want to add is directly supported by Project Designer then WebbotLib will do all the I2C calls for you. If your device is not in the list then add a 'Generic I2C Device'. You can then use functions it provides to read/write the registers normally found in a slave device (check its datasheet). This should allow you to get up and running quickly. If your device needs more complex communications then you can use the low-level start, get, put, stop functions to code the communications yourself from scratch.

An I2C bus provides the following low-level functions:

| **I2cBus - Function Summary** | |
|---|---|
| | setBitRate<br>Sets the communications speed of the I2C bus. |
| `boolean` | start<br>A low level function to start communicating with a given slave device if you are writing your own I2C communications from scratch. |
| | stop<br>This is a low-level call to indicate that the communication session has finished (ie 'hang up the phone'). |
| `uint8_t` | get<br>This is a low-level function to read, and return, a byte from |

## I2cBus - Function Summary

| | |
|---|---|
| | the bus. |
| boolean | put<br>This is a low-level function to write a byte across the bus. |

## I2C Slave

Allows your board to become an I2C Slave that can be called by an I2C Master board.

A slave can only talk when it is asked a question and must therefore listen for incoming messages to which it can reply with a response.

Project Designer allows you to set a unique I2C address for your board as well as defining the message size for incoming requests and outgoing responses.

WebbotLib allows you to define two functions in your code: a 'receive handler' and a 'transmit handler'.

When a master sends you a message WebbotLib will then call your 'receive handler' passing a pointer to the received message. You can then act upon what you have been asked to do - but you cannot send back any response yet.

If the master is expecting a response then WebbotLib will eventually call your 'transmit handler' allowing you to write your response into a buffer that WebbotLib will send back to the master.

Project Designer autonmatically adds the handlers to your main file and they look like this:

```
// A message has been received from an I2C master
void I2cSlave::receiveHandler(cBuffer* rx){
/* add code here to parse the message */
}
// Send response to the previous message
void I2cSlave::transmitHandler(cBuffer* tx){
/* add code here to write it out */
}
```

## IMU

An IMU (Inertial measurement unit) is a compound sensor which typically includes an accelerometer, gyro, and compass. The sensor values are often passed through a Kalman filter to reduce noise and are typically used by airborne vehicles.

### Example

Assuming you have called your sensor 'myIMU' in Project Designer then you can read it as follows:

```
// Read the device and store results
myIMU.read();
// Dump it out
cout << myIMU;
```

All IMU devices support the following functions:

| Function Summary | |
|---|---|
| ACCEL_TYPE | getAccelX<br>Returns the X axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getAccelY<br>Returns the Y axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getAccelZ<br>Returns the Z axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| GYRO_TYPE | getGyroX<br>Returns the X axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getGyroY<br>Returns the Y axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getGyroZ<br>Returns the Z axis rotational velocity in degrees per second at the time of the last read(). |
| COMPASS_TYPE | getBearing<br>Returns the compass bearing, in degrees, at the time of the last read(). |

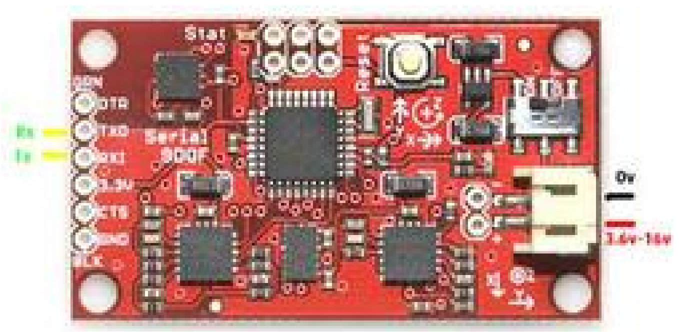
## Function Summary

| | |
|---|---|
| COMPASS_TYPE | getRoll<br>Returns the compass roll, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getPitch<br>Returns the compass pitch, in degrees, at the time of the last read(). |

The following devices are supported:

## Sparkfun Razor

The Sparkfun Razor 9DoF IMU board contains an accelerometer, gyros for roll, pitch and yaw and a magnetometer compass and communicates with your board via a TTL level UART, or directly to your PC with the addition of a (MAX232 type) level shifter.



Based on an 8MHz ATmel ATMega328 processor the firmware for this sensor can be programmed using this library - meaning that you can write your own code to be installed on the sensor. The supplied board contains firmware allowing it to be accessed via a serial terminal like Hyperterminal.

Suitable firmware for the board is available via many sources but this library currently supports firmware from Admin at the Society of Robots and so the sensor will need this firmware to be uploaded prior to use by this library. This can be found here: http://www.societyofrobots.com/robotforum/index.php?topic=11599.msg88205#msg88205

WebbotLib does not currently support the firmware supplied pre-installed on the board.

The Razor also provides the following functions:

| Function Summary | |
|---|---|
| | ledOn<br>Turns on the LED on the Razor board. |
| | ledOff<br>Turns off the LED on the Razor board. |
| | ledSet<br>Turn the LED on the Razor board either on or off. |

All IMU devices support the following functions:

| Imu - Function Summary | |
|---|---|
| ACCEL_TYPE | getAccelX<br>Returns the X axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getAccelY<br>Returns the Y axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getAccelZ<br>Returns the Z axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| GYRO_TYPE | getGyroX<br>Returns the X axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getGyroY<br>Returns the Y axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getGyroZ<br>Returns the Z axis rotational velocity in degrees per second at the time of the last read(). |
| COMPASS_TYPE | getBearing<br>Returns the compass bearing, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getRoll<br>Returns the compass roll, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getPitch<br>Returns the compass pitch, in degrees, at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |

## Sensor - Function Summary

| | |
|---|---|
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Pressure

Supports pressure measurement in Pascals (Pa).

Example

Assuming your device is called 'myPressure' in Project Designer then:-

```
// Read the sensor and store values
myPressure.read();

// Get the last read value
PRESSURE_TYPE val = myPressure.getPa();

// Print it out
cout << "Pressure:" << val;
```

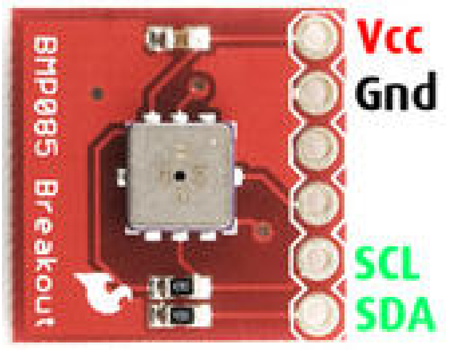
All pressure sensors support the following functions:

| Function Summary | |
|---|---|
| PRESSURE_TYPE | getPa<br>Returns the last read pressure in Pa. |
| PRESSURE_TYPE | getkPa<br>Returns the last read pressure in kPa. |
| double | relativeDepth<br>Returns the under water depth in meters relative to when the power was switched on. |
| double | altitude<br>Returns the current altitude in meters above sea level. |
| double | relativeAltitude<br>Returns the current altitude in meters above, or below, the location where the power was turned on. |

The following devices are supported:

## BMP085

Bosch I2C pressure and temperature sensor capable of measuring 30000 Pa to 110000 Pa. Equivalent to 500m below sea level up to 9,000m above sea level.



This must be connected to an I2C bus.

The BMP085 sensor also provides the following functions:

### Function Summary

| | |
|---|---|
| TEMPERATURE_TYPE | getCelsius<br>Returns the temperature in celsius. |

All pressure sensors support the following functions:

### Pressure - Function Summary

| | |
|---|---|
| PRESSURE_TYPE | getPa<br>Returns the last read pressure in Pa. |
| PRESSURE_TYPE | getkPa<br>Returns the last read pressure in kPa. |
| double | relativeDepth<br>Returns the under water depth in meters relative to when the power was switched on. |
| double | altitude<br>Returns the current altitude in meters above sea level. |
| double | relativeAltitude<br>Returns the current altitude in meters above, or below, the location where the power was turned on. |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## MPX5100A

Motorola pressure sensor returning 45mV per kPa.

This must be connected to an ADC pin.

All pressure sensors support the following functions:

| Pressure - Function Summary | |
|---|---|
| PRESSURE_TYPE | getPa<br>Returns the last read pressure in Pa. |
| PRESSURE_TYPE | getkPa<br>Returns the last read pressure in kPa. |
| double | relativeDepth<br>Returns the under water depth in meters relative to when the power was switched on. |
| double | altitude<br>Returns the current altitude in meters above sea level. |
| double | relativeAltitude<br>Returns the current altitude in meters above, or below, the location where the power was turned on. |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## RTC

Adds support for different real time clocks.

**Note:** if you are using your clock and getting a value back but it's not changing then try setting the clock to a new value. This will often make it wake up ! Project Designer adds a line of code to your appInitSoftware but it is commented out - after all - no point setting the clock to a fixed value every time you turn on your board....

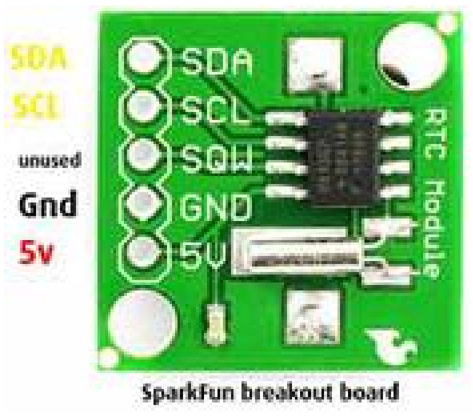
All real time clocks support the following functions:

| Function Summary | |
|---|---|
| | dump<br>Dump the current clock values to the standard output device. |
| | dumpTo<br>Dump the value of the clock to the specified output stream. |
| `boolean` | set<br>Attempt to set the clock to the specified date and time. |
| `DATE_TIME*` | read<br>Read the real time clock and return the values. |

The following devices are supported:

## DS1307 Real Time Clock

Adds support for the DS1307 real time clock.



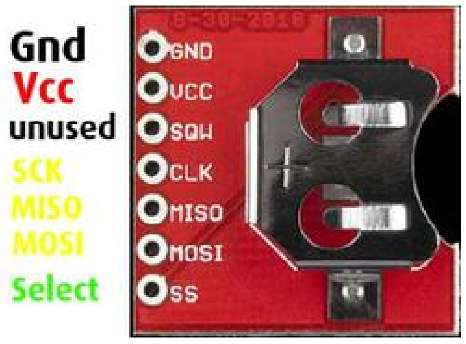
This IC is available as a board from suppliers such as SparkFun (shown) but can also be easily constructed using strip board as all the components, including the battery holder, are all available as 'through hole' components,

All real time clocks support the following functions:

| RTC - Function Summary | |
|---|---|
| | dump<br>Dump the current clock values to the standard output device. |
| | dumpTo<br>Dump the value of the clock to the specified output stream. |
| boolean | set<br>Attempt to set the clock to the specified date and time. |
| DATE_TIME* | read<br>Read the real time clock and return the values. |

## DS3234 Real Time Clock

Adds support for the DS3234 real time clock.



This IC is available as a board from suppliers such as SparkFun (shown) but can also be easily constructed using strip board as all the components, including the battery holder, are all available as 'through hole' components,

All real time clocks support the following functions:

| RTC - Function Summary | |
|---|---|
| | dump<br>Dump the current clock values to the standard output device. |
| | dumpTo<br>Dump the value of the clock to the specified output stream. |
| boolean | set<br>Attempt to set the clock to the specified date and time. |
| DATE_TIME* | read<br>Read the real time clock and return the values. |

## SPI

Adds support for SPI devices over a hardware or software bus.

Note that if you are using the hardware SPI interface then this may also be used by an ISP programmer such as the AVRISP MKII. This means that it is possible to create problems on the SPI bus because it is used by the programmer and by your own program. So as a security measure then I would recommend fitting a pull up resistor (anything from 4.7k ohm to 10k ohm) on each slave device going from their chip select (CS) line to the +ve power supply. This will keep the devices disabled whilst the programmer is being used.

The following devices are supported:

## Generic SPI Device

If you have an SPI device that is not otherwise listed in Project Designer then you should add it to your project as a Generic SPI Device.

This will then give you access to the different functions for communicating with the device.

An SPI device provides the following functions:

| SpiDevice - Function Summary | |
|---|---|
| | select<br>Selects the device as the target you want to communicate with,Only one device can be selected at a time and WebbotLib will make sure that when one device is selected that all other devices on the same SPI bus are de-selected first. |
| | write<br>Send one or more bytes to the device. |
| uint8_t | xfer<br>Performs a transfer by sending the specified data and returning the response, |
| uint8_t | get<br>Reads a byte from the device. |
| | read<br>Read one or more bytes from the device. |

## SPI Bus

Creates a hardware or software SPI bus to allow you to connect SPI devices.

Once added within Project Designer then you can choose from the available devices that can be connected to the bus.

An SPI bus provides the following functions:

| SpiBus - Function Summary | |
|---|---|
| SPI_CLOCK | getClockPrescaler<br>Get the current prescale value used to set the bus speed. |
| | setClockPrescaler<br>Set the prescaler value used to control the bus speed. |

# Stepper

Support for various stepper motor controllers.

## So what's a stepper motor?

Whilst a DC motor continues to rotate whilst any voltage is applied; a stepper motor contains electromagnets which are used to kick the motor around in individual 'steps' - or 'ticks'. To perform a 'step' then the electromagnets need to be energised in a particular sequence. It is only this 'change' that performs a 'step' - the continued application of the same current will produce no further steps. So to keep a stepper motor rotating then the electromagnets must continue to be energised in a given sequence.

All stepper motors will state the number of steps per revolution. ie a motor with 200 steps per revolution will mean that each step will turn through 360° / 200 ie 1.8° per step.

Since a 'step' only happens when we ask it to happen; and we know the angle each step represents then these **appear** to be ideal for giving accurate and precise turning angles and distance measurement.

Compare with a DC motor. Since we don't really know precisely the exact RPM of our DC motor at any given time then we use an encoder to tell us that information - and it is used in a closed loop feedback system to adjust the voltage, or PWM duty cycle, to control the DC motor more precisely.

On the other hand - a stepper motor is open loop. ie we don't need the encoder to say how fast it is turning because it turns through a fixed angle for every 'step' command we send it. - so if we added an encoder then it would only confirm what we already know.

Well that's the elementary theory - but, as usual, the real world sometimes gets in the way. For example: let's take a small stepper motor out of a floppy disk drive and attach it to the drive shaft of our 30 ton tank. Will it be guaranteed to move the tank track for every step => "No the tank is too heavy ie the motor doesn't have enough torque". Conversely lets take a great big heavy industrial stepper motor and give it step commands every millionth of a second. Will it rotate a million times a second => "No the motor can't move that fast - as it has a maximum RPM".;

We have learnt some examples of how a stepper motor can go wrong - but if the motor is running inside its limits of torque and RPM then it can provide a cheaper alternative to a DC motor with an encoder. Of course there is nothing to stop you adding an encoder to a stepper motor but it defeats the whole point some what. Spend more money on a better stepper motor and save the money by not requiring the encoder.

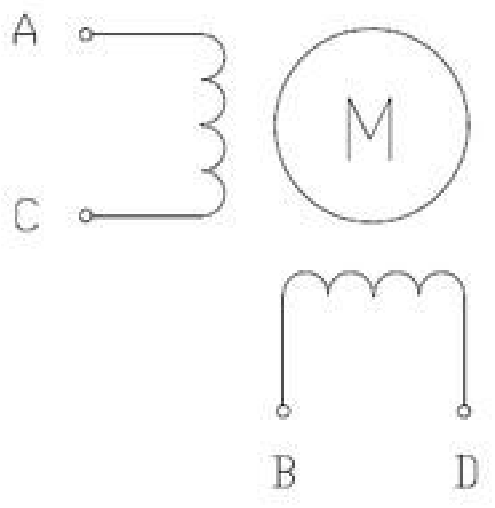
## Purchasing Decisions

When looking to buy a stepper motor then you need to check info such as whether it has a gearbox, A gearbox will improve the torque (moving power) but reduce the RPM (speed).

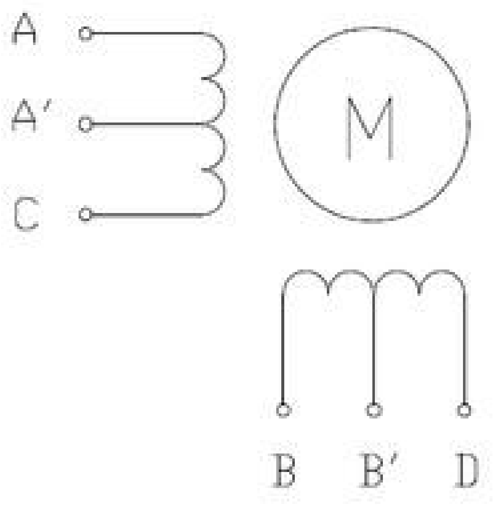
The next consideration is that there are two different categories of stepper motor: **bi-polar** and **uni-polar**.

If you have already purchased a motor, or re-cycled one from some old hardware, then you need to determine the category.

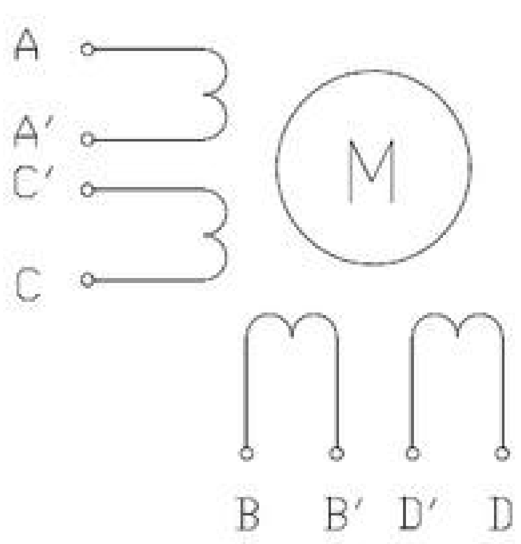
Generally: a bi-polar motor has 4 connections and a uni-polar has either 6 or 8 connections.



This is a bi-polar stepper motor



This is a 6 pin uni-polar stepper motor. By leaving the A' and B' centre taps unused then we can use the AC and BD terminals as if it was a bi-polar motor.



This is an 8 pin uni-polar stepper motor. By connecting A' to C' and B' to D' then we can use the AC and BD terminals as if it was a bi-polar motor.

A simple ohm-meter, on a low setting, can be used to discover which pin is which.

Note how a uni-polar can be transformed into a bi-polar motor; but not the other way around.

## Stepper Motor Specifications

The specifications of your motor should list the resistance of each coil (or 'phase' as they are also known). This will help you to use your multi-meter to work out which terminals are which if your data sheet isn't very revealing. If not: then you will have to play with your ohm meter to work it out.

Knowing the coil resistance then the data sheet may give a range of battery voltages that can be used or it may list a maximum current. Using Ohms law: Voltage = Current x Resistance then we know the coil **Resistance** so given either the current or voltage then we can calculate the other value.

Now that you know the voltage to be applied, and hence the current drawn, then you can find a suitable driver for that combination.

For example: if the coils are 2 Ohm and we have a 6V battery then the controller needs to be able to supply (Volts./ Resistance) = 6/2 = 3 Amps.

Check the torque of the motor you buy is capable of the mass it needs to control. If your motor is just rotating the 'head' of the robot then it wont need as much torque as if it was driving the whole robot across the ground.

Also check the number of steps per rotation, or angle per step, to make sure it gives the accuracy you need.

## Stepper Motor Controllers

The WebbotLib stepper motor driver doesn't mean that you can just connect the motor directly to the micro controller - they need way too much current. Hence you will also need one of the supported stepper motor controllers.

## Drive Modes

All stepper motor controllers provide a 'Full Step' mode. This mode provides the best torque but requires the highest current drain and gives a 1:1 ratio - ie it will give 200 steps for a motor listed as having 200 steps per revolution.

Most controllers can provide 'Half Step' mode. This doubles the number of steps per revolution but normally requires less current and hence there is less torque. NB it also means the maximum RPM is also halved.

Deluxe controllers go beyond 'Half Step' and provide 4, 8 or even 16 times the number of steps per revolution. Often called 'micro-stepping' but, as mentioned above, this positional accuracy comes at the expense of torque and RPM.

## Summary

So you should now understand how to connect your motor as well as the benefits/implications of using micro-stepping for greater positional accuracy at the expense of torque.

Most controller boards allow you to configure the 'micro-stepping' via some input pins. However: this is something that you will set once and not reconfigure whilst the robot is running. Hence this library leaves those connections down to you. Suffice it to say that if your motor has 200 steps/revolution and you configure the driver to use 'Half Stepping' to give 400 steps/revolution then you should just declare it in Project Designer as having 400 steps/revolution.

## Project Designer

When you create a stepper motor driver in Project Designer you need to specify the 'Maximum Step Frequency (Hz)' ie a value of 200 would mean that the maximum speed for the motors is set at 200 steps per second. Check the datasheets for the motors you are using to see if they specify this value. If they don't then you will need to find it by trial error. If the number is too low then you will be restricting the maximum speed of the motor and if it is too high then the motor may ignore some of the pulses as it is unable to keep up.

If you are using a variety of different motors then use the smallest value across the different motors - alternatively create a different driver for each motor model.

As mentioned earlier stepper motors are liable to slippage if they are not used within their capabilities. This normally happens because you are trying to turn it too fast for the load it is supporting or because you are accelerating/braking too fast (ie the wheels skid). Consequently: WebbotLib allows you to specify the maximum acceleration/braking to be used to try and avoid such slippage. When adding each motor in Project Designer you can specify the 'Acceleration speed increment' and the 'Acceleration speed interval' - these two settings combine to set the maximum acceleration.

For example: if the maximum step frequency is 200Hz (ie every 5ms) and we set the 'Acceleration speed increment' to 1 and the 'Acceleration speed interval' to 10 then the motor will increase by '1' every '10 x 5ms = 50ms'. So if the motor is currently at rest and we set a DRIVE_SPEED of 100 then it will take 5 seconds to get up to the final speed.

Setting the 'Acceleration speed increment' to 127 will give 'infinite' acceleration.

## Modes of Operation

WebbotLib allows you control the stepper motor in two different modes: continuous and one-shot.

In continuous mode you specify a DRIVE_SPEED and the motor accelerates, or decelerates, to the required speed. This allows you to use it just like any other DC motor and you can do this by using the functions, and example, shown in "*Actuators*" (see page 38) .

In one-shot' mode you specify how many steps you want the motor to turn. This will happen in the background and once completed the motor will stop. This is useful, for example, to make precise turns.

All stepper motors provide the following functions:

| Function Summary | |
|---|---|
| uint16_t | getStepsPerRevolution<br>Returns the number of steps required to rotate the motor through 360°. |
| DRIVE_SPEED | getActualSpeed<br>Returns the actual drive speed of the stepper motor. |
| uint16_t | getActualPosition<br>Returns the current position of the stepper motor between 0 and getStepsPerRevolution()-1. |
| | step<br>Tell the motor to move by the specified number of steps and then stop. |
| int16_t | getStepsRemaining<br>Returns the number of remaining steps from the last step() command. |

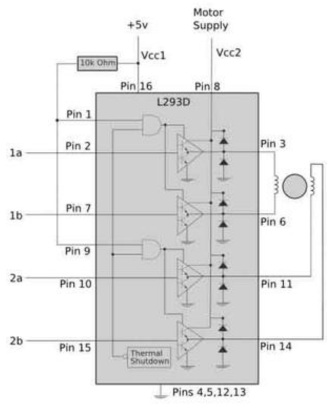
The following devices are supported:

## Generic Bipolar stepper motor

Adds support for a generic bipolar stepper motor controller such as the L293D or SN754410.

This uses 4 output pins to control the two coils as well as an optional output pin which is set high to enable the motor.

If you are using an L293D, or a pin compatible chip like the SN754410, then here is the schematic:-



The enable pins are pulled high using a 10k resistor just in case you choose not to use the enable logic. If you wish to be able to disconnect the motor (ie turn off the supplied current) then specify an enable output pin from your board and connect it to pins 1 and 9.

All stepper motors provide the following functions:

**StepperMotor - Function Summary**

| | |
|---|---|
| uint16_t | getStepsPerRevolution<br>Returns the number of steps required to rotate the motor through 360°. |
| DRIVE_SPEED | getActualSpeed<br>Returns the actual drive speed of the stepper motor. |
| uint16_t | getActualPosition<br>Returns the current position of the stepper motor between 0 and getStepsPerRevolution()-1. |
| | step<br>Tell the motor to move by the specified number of steps and then stop. |

| StepperMotor - Function Summary | |
|---|---|
| int16_t | getStepsRemaining<br>Returns the number of remaining steps from the last step() command. |

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Generic L297 stepper motor

Adds support for a generic L297 stepper motor controller.

All stepper motors provide the following functions:

| **StepperMotor - Function Summary** | |
|---|---|
| uint16_t | getStepsPerRevolution<br>Returns the number of steps required to rotate the motor through 360°. |
| DRIVE_SPEED | getActualSpeed<br>Returns the actual drive speed of the stepper motor. |
| uint16_t | getActualPosition<br>Returns the current position of the stepper motor between 0 and getStepsPerRevolution()-1. |
| | step<br>Tell the motor to move by the specified number of steps and then stop. |
| int16_t | getStepsRemaining<br>Returns the number of remaining steps from the last step() command. |

All actuators provide the following functions:

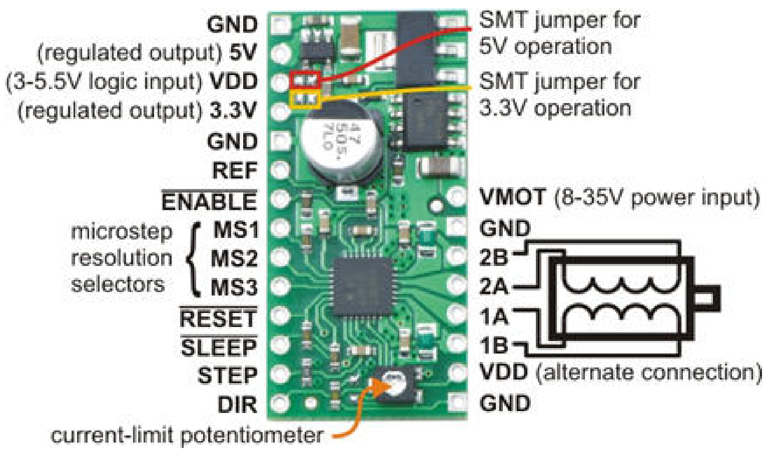
| **Actuator - Function Summary** | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |

## Actuator - Function Summary

| | |
|---|---|
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Pololu A4983 stepper motor

Adds support for a Pololu A4983 stepper motor controller.



This powerful controller can supply up to 2A per coil and contains a chopper circuit with a potentiometer to limit the current. Make sure you **read the manual** as to how to use this as its too complex for me to describe here.

You can either power the board from your micro controller regulated supply or you can power it from the same battery as you are using to power the motor.

To power it from your micro controller then attach one of the Vdd pins to the regulated supply on your micro controller. This should be the same voltage that is used by the micro controller. For example: the Axon has both a 5V regulated supply and a 3.3V regulated supply but, since the ATMega640 is powered from the 5V supply then you must connect Vdd to the 5V supply.

To power it from the motor battery then leave Vdd unconnected and instead you will need to short out one of the solder jumpers on the board. For example: the ATMega640 on an Axon uses 5V so you should short out the 5V jumper on the A4983 board.

All stepper motors provide the following functions:

| **StepperMotor - Function Summary** | |
|---|---|
| uint16_t | getStepsPerRevolution<br>Returns the number of steps required to rotate the motor through 360°. |
| DRIVE_SPEED | getActualSpeed<br>Returns the actual drive speed of the stepper motor. |
| uint16_t | getActualPosition<br>Returns the current position of the stepper motor between 0 and getStepsPerRevolution()-1. |
| | step<br>Tell the motor to move by the specified number of steps and then stop. |

| StepperMotor - Function Summary | |
|---|---|
| int16_t | getStepsRemaining<br>Returns the number of remaining steps from the last step() command. |

All actuators provide the following functions:

| Actuator - Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

## Storage

This folder contains code which can be used to store data to persistent storage.

This currently includes hardware such as micro SD cards and EEPROM devices.

The following devices are supported:

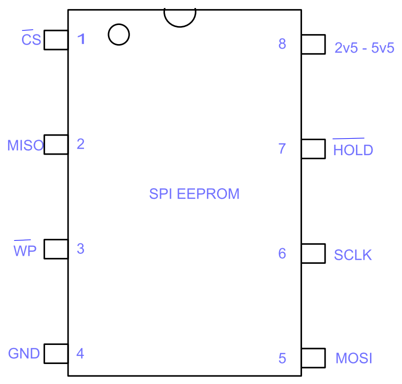
## SPI EEPROM

Provides generic support for EEPROM devices accessed over an SPI bus.

Note that these device are often advertised showing the capacity as the number of 'bits' they provide. Such as 512k bits. Since we are dealing in 8 bit bytes then you need to divide this figure by 8. So 512k bits => 64k bytes. Not sure why they are sold that way - maybe its because of 32 bit vs 16 bit vs 8 bit processors?

**So why would you need one?**

These devices are quite cheap (a dollar or two) and typically come in an 8 pin format and are therefore physically quite small but can store a reasonably large amount of data - so they are good for storing 'work in progress'. Your processor may come with some 'on board' EEPROM space but typically it will be quite small. So if you need more capacity then here is your solution.



The diagram shows a typical pin out for such a device but you may want to check the data sheet for the device you have purchased.

Besides a Ground and Supply voltage (typically 1.8v to 5.5v) you will also see the MISO, MOSI and SCLK pins which are the heart of the SPI bus (the MISO and MOSI pins may be labelled Data Out, or DO, and Data In, or DI, respectively). You will also see a chip select, or CS, pin that is used to select the device. So that accounts for 6 of the pins on the device, There are two others: HOLD and WP. HOLD can be connected to its neighbouring supply pin and WP can be connected to its neighbouring Ground pin.

There are a few other things you need to check from the datasheet before you can use the device:

- The "address" size
- The "page" size

The address size may be 1, 2, 3 or 4 bytes (ie 8, 16, 24 or 32 bits) and is normally dictated by the number of bytes of storage:

- 1 byte can address up to 256 bytes
- 2 bytes can address up to 64Kb
- 3 bytes can address up to 16 Mb
- 4 bytes can address up to 4 Gb

The page size is always a power of 2 and is normally 128, 256 or 512 bytes - it chops the device up into a number of similar sized pages. A single write to the device must all be in the same page. This value is passed to WebbotLib when creating the device so that your code doesn't have to worry about the restriction - WebbotLib 'just does it'.

In Project Designer you must must add the EEPROM to an existing "*SPI Bus*" (see page 231) .

An SPI EEPROM provides the following functions:

| SPI_Eeprom - Function Summary | |
|---|---|
| uint8_t | readByte<br>Reads a single byte from the EEPROM. |
| | readBytes<br>Read a number of bytes from the EEPROM into memory. |
| uint8_t | writeByte<br>Writes a single byte to the EEPROM. |
| | writeBytes<br>Writes a number of bytes from memory to the EEPROM. |
| EEPROM_ADDR | totalBytes<br>Returns the total number of bytes the EEPROM can store. |

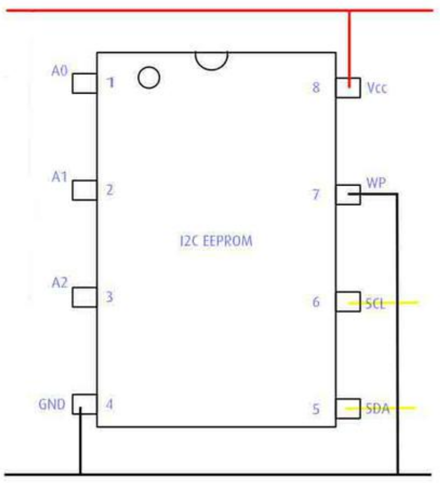
## I2C EEPROM

Provides generic support for EEPROM devices accessed over an I2C bus.

Note that these device are often advertised showing the capacity as the number of 'bits' they provide. Such as 512k bits. Since we are dealing in 8 bit bytes then you need to divide this figure by 8. So 512k bits => 64k bytes. Not sure why they are sold that way - maybe its because of 32 bit vs 16 bit vs 8 bit processors?

**So why would you need one?**

These devices are moderately cheap (say five dollars) and typically come in an 8 pin format and are therefore physically quite small but can store a moderate amount of data - so they are good for storing 'work in progress'. Your processor may come with some 'on board' EEPROM space but typically it will be quite small. So if you need more capacity then here is your solution.



The diagram shows a typical pin out for such a device but you may want to check the data sheet for the device you have purchased.

Besides a Ground and Supply voltage you will also see the SDA and SCL pins which are the heart of the I2C bus.

You will also see a write protect pin, WP, which can be connected to Ground so that the chip is 'writable'.

The only other pins are those labelled A0, A1 and A2.

In 'simple' terms these can be connected to Vcc or Gnd to modify the I2C address of the device starting at its default address of 0xC0 where the resultant address is:

1 1 0 0 a2 a1 a0 0

So "in theory" these 3 bits allow you to connect 8 devices with addresses: 0xC0, 0xC2, 0xC4, 0xC6, 0xC8, 0xCA and 0xCE.

Unfortunately, its not that simple ... although WebbotLib takes care of the complexities for you.

There are a few other things you need to check from the datasheet before you can use the device:

- The "address" size
- The "page" size

The address size may be 1, 2, 3 or 4 bytes (ie 8, 16, 24 or 32 bits) and is normally dictated by the number of bytes of storage:

- 1 byte can address up to 256 bytes
- 2 bytes can address up to 64Kb
- 3 bytes can address up to 16 Mb
- 4 bytes can address up to 4 Gb

However: you may have,say, a 1024 byte EEPROM which therefore needs 10 bits to specify an address. In this case: it uses 1 byte (8 bits) for the memory address and the remaining two bits are placed into A0 and A1 - meaning that each chip has 4 x I2C addresses as if it was 4 separate 256 byte chips. In this case the A0 and A1 pins should be left unconnected.

But don't worry:- the WebbotLib Project Designer will show you examples.

The page size is always a power of 2 - it chops the device up into a number of similar sized pages. A single write to the device must all be in the same page. This value is passed to WebbotLib when creating the device so that your code doesn't have to worry about the restriction - WebbotLib 'just does it'.

An I2C EEPROM provides the following functions:

| **I2C_Eeprom - Function Summary** | |
|---|---|
| uint8_t | readByte<br>Reads a single byte from the EEPROM. |
| | readBytes<br>Read a number of bytes from the EEPROM into memory. |
| uint8_t | writeByte<br>Writes a single byte to the EEPROM. |
| | writeBytes<br>Writes a number of bytes from memory to the EEPROM. |
| EEPROM_ADDR | totalBytes<br>Returns the total number of bytes the EEPROM can store. |

## SD Card

Provides storage on an SD card or the phased out MMC card.

An SD card is like a miniature EEPROM accessed via the SPI interface. They are also available as micro SD cards which, physically, are much smaller. You may be familiar with them as they provide the 'memory' for your digital camera or your hand held phone.

Here is a typical pin out of the bare card:-

```
  _________________
 /  1 2 3 4 5 6 78 | <- view of SD card looking at contacts
/ 9                | Pins 8 and 9 are present only on SD cards and not MMC cards
|        SD Card   |
|                  |
/\/\/\/\/\/\/\/\/\/\
1 - CS   (chip select)          - wire to any available I/O pin
2 - DIN  (data in, card<-host)  - wire to SPI MOSI pin
3 - VSS  (ground)               - wire to ground
4 - VDD  (power)                - wire to 3.3V power
5 - SCLK (data clock)           - wire to SPI SCK pin
6 - VSS  (ground)               - wire to ground
7 - DOUT (data out, card->host) - wire to SPI MISO pin
```

The cards are available in various capacities from a few Mb to umpteen Gb - so can certainly hold more data than your processor can hold in memory.

**Why would I need one?** These guys can store a lot of info - and like an EEPROM their data is 'saved' even when the power is off. So they are great for storing any kind of logging or data acquisition. But, unlike an EEPROM, they are removable and can be physically removed and plugged into other things - including your computer!

If we were to make the analogy that these devices are like mini bank cards then we also need a mini ATM to plug them into so that they can be read. (No worries - they don't know your banking info!)

The ATM will not only provide power (normally 3v3) but may also provide a digital IO output pin to indicate if a card has been inserted (called 'card detect').

Here is an example from SparkFun
http://www.sparkfun.com/commerce/product_info.php?products_id=204

Lots of other boards are available and some can even generate the 3v3 supply via a regulator and may be powered from, say, +5v.

But the beauty of these devices is that you can also plug them into a USB carrier and plug that into your computer and use it like a memory stick. For example: http://www.sparkfun.com/commerce/product_info.php?products_id=8698

Ok - enough advertising!

In Project Designer you must must add the SD Card to an existing "*SPI Bus*" (see page 231) .

WebbotLib allows you to access these cards in one of two modes:

1. Stand alone
2. Computer format

In 'stand alone' format your code will be smaller but all you can do is read/write from/to a given 512 byte sector block number on the card. This 'custom' format means that the card will NOT be recognised if it is plugged into a computer (unless it is re-formated). Unsurprisingly:- if you try doing it then the computer may report all sorts of errors. However: you could write a small program on your micro-controller to dump the card content over a UART to your PC.

In 'computer format' the generated code will be bigger but will mean that you can transfer the card to your computer and it will be able to read it and vice versa. ie it becomes a removable hard disk drive. However: before it can be used then you must format it from your computer. I suggest using a program such as: http://www.sdcard.org/consumers/formatter/ to do that.

The mode can be set in Project Designer via the 'Formatted for PC use?' checkbox. If ticked then it will use 'computer format' otherwise it will use 'stand alone' format.

For 'stand alone' format then you are limited to the read and write functions.

For 'computer format' Project Designer will create a variable called 'disk_xxxx' where 'xxxx' is the name you have given the card in Project Designer. This allows you to manipulate files on the disk using the functions in "*Disk*" (see page 542) .

An SD Card supports the following functions:

| **SdCard - Function Summary** | |
|---|---|
| `boolean` | readSectors<br>Read one or more 512 byte sectors from the card into memory. |
| `boolean` | writeSectors<br>Write one or more 512 byte sectors from memory to the card. |
| `uint32_t` | totalSectors<br>Returns the total number of 512 byte sectors on the card. |

## Temperature

Supports temperature measurement.

All readings from this library are in 'celsius' but can be converted into 'fahrenheit'.

### Example

Assuming your device is called 'myTemp' in Project Designer then:-

```
// Read the sensor and store values
myTemp.read();

// Get the last read value
TEMPERATURE_TYPE val = myTemp.getCelsius();

// Print it out
cout << "Temperature:" << val;
```

All temperature sensors support the following functions.

| Function Summary | |
|---|---|
| TEMPERATURE_TYPE | getCelsius<br>Returns the temperature from the last read() in celsius. |
| TEMPERATURE_TYPE | Temperature::getFahrenheit<br>Convert a temperature from celsius to fahrenheit. |

The following devices are supported:

## Phidget Temperature Sensor

Phidget temperature sensor.

This must be connected to an ADC pin and can return temperatures from -40 to +125 degrees celsius.

For an example of how to read it see the example for "*Temperature*" (see page 250)

All temperature sensors support the following functions.

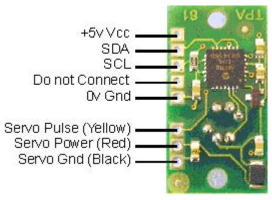
| **Temperature - Function Summary** | |
|---|---|
| TEMPERATURE_TYPE | getCelsius<br>Returns the temperature from the last read() in celsius. |
| TEMPERATURE_TYPE | Temperature::getFahrenheit<br>Convert a temperature from celsius to fahrenheit. |

All sensors provide the following functions

| **Sensor - Function Summary** | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## TPA81 Thermal Array

Devantech thermal array.



This must be connected to the I2C bus and a 5V power source. Note that this library does not support the in-built servo driver.

### Note

The getCelsius() function will return the ambient temperature but you can access each of the individual sensor readings using the function getSensorCelsius().

### Example

Assuming you have called the device 'myTPA81' in Project Designer then you can use the sensor as follows:-

```
// Read the device and store values
myTPA81.read();

// Output the ambient temperature
TEMPERATURE_TYPE ambientC = myTPA81.getCelsius();
TEMPERATURE_TYPE ambientF = Temperature::toFahrenheit(ambientC);
cout << "Ambient: " << ambientC << "C " << ambientF << "F\n";

// Output each individual reading
for(int inx = 0; inx < 8 ; inx++){
    cout << "Sensor " << inx << "=" << myTPA81.getSensorCelsius(inx) + '\n';
}
```

The TPA81 also provides the following function:

| Function Summary | |
|---|---|
| TEMPERATURE_TYPE | getSensorCelsius<br>Returns the temperature, in celsius, from one of the individual temperature sensors. |

All temperature sensors support the following functions.

| Temperature - Function Summary | |
|---|---|
| TEMPERATURE_TYPE | getCelsius<br>Returns the temperature from the last read() in celsius. |
| TEMPERATURE_TYPE | Temperature::getFahrenheit<br>Convert a temperature from celsius to fahrenheit. |

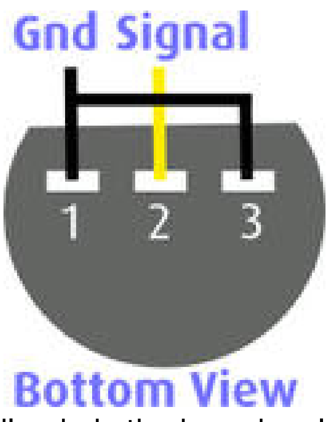
All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

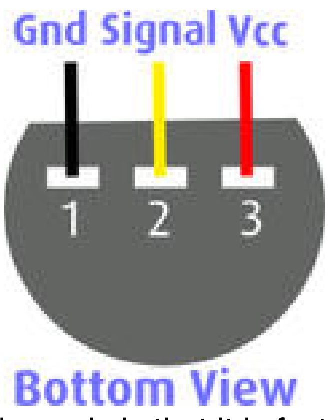
## Maxim DS18B20

Adds support for the Maxim DS18B20 thermometer.

This must be added to the ('Dallas') one wire bus.

The thermometer can either be connected using two pins: 

whereby the two outer pins are both connected to ground and the middle pin is the bus signal pin. This is called 'parasitic' mode since the thermometer uses the signal pin to power itself. The drawback of this mode is that it takes longer to read a temperature value.

Alternatively it can connected using three pins: 

whereby the thermometer has a seperate power line. The advantage of this mode is that it is faster to read a temperature value at the price of an extra 'wire'.

Note that you can change between the two connection methods without recompiling your code since WebbotLib asks the thermometer which of the two wiring methods have been used.

All one wire devices provide the following functions:

| **Function Summary** | |
|---|---|
| `boolean` | exists<br>Searches the bus to discover whether this device is plugged in. |

| Function Summary | |
|---|---|
| boolean | found<br>Tests whether this device was found when the bus was initialised. |
| | dumpROM<br>Dumps the complete ROM ID of the device to the specified output stream. |

All temperature sensors support the following functions.

| Temperature - Function Summary | |
|---|---|
| TEMPERATURE_TYPE | getCelsius<br>Returns the temperature from the last read() in celsius. |
| TEMPERATURE_TYPE | Temperature::getFahrenheit<br>Convert a temperature from celsius to fahrenheit. |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Voltage

Supports voltage measurement.

All readings from this library are in 'volts' regardless of whether it is AC or DC. AC voltages are positive whereas DC voltages can be positive or negative.

### Example

Assuming you have called the sensor 'myVolts' in Project Designer then any voltage sensor can be accessed as follows:-

```
// Read the sensor and store the value
myVolts.read();

// Access the value
VOLTAGE_TYPE v = myVolts.getVolts();
cout << "Voltage=" << v;

// Or dumped out using
cout << myVolts;
```

All voltage sensors provide the following functions:

| Function Summary | |
|---|---|
| VOLTAGE_TYPE | getVolts<br>Returns the voltage at the time of the last read(). |

The following devices are supported:

## Phidget Voltage Sensor

Phidget DC voltage sensor measuring from -30v to +30v.

This must be connected to an ADC pin.

### Example

For an example see "*Voltage*" (see page 256) .

All voltage sensors provide the following functions:

| Voltage - Function Summary | |
|---|---|
| VOLTAGE_TYPE | getVolts<br>Returns the voltage at the time of the last read(). |

All sensors provide the following functions

| Sensor - Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

## Reference

This section details the core C functions in WebbotLib that can be called from your application by including the relevant header file.
This gives you access to some of the more advanced features in the library such as timers, pin change interrupts, scheduling as well as mathematical help with matrices and PID functions.

## buffer.h

Defines a 'first in first out circular' buffer which can be written to and read from by different tasks.

The 'front' of the buffer is where you normally read data from, whereas the 'back' is where you write data to. If the front is equal to the end then there is no data in the buffer.

| Standard Function Summary | |
|---|---|
| | bufferInit(cBuffer* buffer, void *start, size_t size)<br>Before the buffer can be used it must be initialised by this function. |
| boolean | bufferGet(cBuffer* buffer,uint8_t *rtn)<br>Gets the next byte from the buffer. |
| boolean | bufferPut(cBuffer* buffer, uint8_t data)<br>Writes the byte into the buffer. |
| size_t | bufferFreeSpace(const cBuffer* buffer)<br>Returns the number of bytes available in the buffer. |
| size_t | bufferBytesUsed(const cBuffer* buffer)<br>Return the number of bytes currently stored in the buffer. |
| boolean | bufferQueue(cBuffer* buffer, const void* src, size_t len)<br>Write a block of bytes to the buffer it has enough capacity to support the whole block. |

| Advanced Function Summary | |
|---|---|
| | bufferDump(cBuffer* buffer, size_t numbytes)<br>Deletes numbytes from the front of the buffer. |
| uint8_t | bufferGetAtIndex(const cBuffer* buffer, size_t index)<br>This allows you to look ahead in the buffer and will return the byte at index from the buffer but will leave the byte in the buffer. |
| | bufferFlush(cBuffer* buffer)<br>Removes all data from the buffer |
| boolean | bufferIsFull(const cBuffer* buffer)<br>Returns TRUE if the buffer is full or FALSE if not. |

## Standard Function Detail

### bufferInit

`bufferInit(cBuffer* buffer, void *start, size_t size)`

Before the buffer can be used it must be initialised by this function.

Parameters:

buffer - The cBuffer being initialised

start - The byte array to be used for the buffer

size - The number of bytes in the buffer

For example you could initialise a 30 byte buffer as follows:-

```
cBuffer myBuffer; // create the buffer
char data[30]; // create the array to be used by the buffer
// Initialise myBuffer to use the data array
bufferInit(&myBuffer,data,sizeof(data));
```

### bufferGet

`boolean bufferGet(cBuffer* buffer,uint8_t *rtn)`

Gets the next byte from the buffer.

If the buffer is empty this will return FALSE otherwise it will return TRUE and store the next available byte at the address specified.

### bufferPut

`boolean bufferPut(cBuffer* buffer, uint8_t data)`

Writes the byte into the buffer. If this returns FALSE then the buffer was full and the byte was not written.

### bufferFreeSpace

`size_t bufferFreeSpace(const cBuffer* buffer)`

Returns the number of bytes available in the buffer.

Note that if the buffer is being read from, or written to, by an interrupt service routine then this number may become out of date almost immediately.

### bufferBytesUsed

`size_t bufferBytesUsed(const cBuffer* buffer)`

Return the number of bytes currently stored in the buffer.

### bufferQueue

`boolean bufferQueue(cBuffer* buffer, const void* src, size_t len)`

Write a block of bytes to the buffer it has enough capacity to support the whole block.

This will return TRUE if all of the data has been written to the queue or FALSE if the buffer is unchanged because it did not enough space to copy the whole block.

## Advanced Function Detail

### bufferDump

`bufferDump(cBuffer* buffer, size_t numbytes)`

Deletes **numbytes** from the front of the **buffer.** This is like calling bufferGet(cBuffer* buffer,uint8_t *rtn) numbytes times.

---

### bufferGetAtIndex

`uint8_t bufferGetAtIndex(const cBuffer* buffer, size_t index)`

This allows you to look ahead in the buffer and will return the byte at **index** from the **buffer** but will leave the byte in the buffer.

If the value of **index** exceeds the current length of the buffer then the behaviour is undefined.

---

### bufferFlush

`bufferFlush(cBuffer* buffer)`

Removes all data from the buffer

---

### bufferIsFull

`boolean bufferIsFull(const cBuffer* buffer)`

Returns TRUE if the buffer is full or FALSE if not.

## clock.h

Use the clock for delays and to test for time intervals.

### Standard Function Summary

| | |
|---|---|
| `TICK_COUNT` | clockGetus()<br>Get the current time in µS. |
| `boolean` | clockHasElapsed(TICK_COUNT usStart, TICK_COUNT usWait)<br>Test if a given duration has elapsed. |
| | clockWaitms(TICK_COUNT ms)<br>Pause for the given number of milliseconds. |
| | clockWaitus(TICK_COUNT us)<br>Pause for the given number of microseconds. |

### Advanced Function Summary

| | |
|---|---|
| `boolean` | clockHasElapsedGetOverflow(TICK_COUNT usStart, TICK_COUNT usWait, TICK_COUNT* overflow)<br>Similar to 'clockHasElapsed' but returns the number of µS left to go if the duration has not elapsed, or the number of µS we have exceeded the duration if is has elapsed. |

### Standard Function Detail

#### clockGetus

`TICK_COUNT clockGetus()`

Get the current time in µS.

Note that this number will wrap around so a later reading may give a smaller value.

This happens every 0xffffffff or 4,294,967,295 microseconds ie every 4295 seconds or every 72 minutes (approx).

This means that the longest time difference you can sense by subtracting two values is about 72 minutes - this should not be a problem since if you are trying to do that then your program is probably wrong.

Note that, even with wrap around, you can always subtract two values to get a duration (so long as it's less than 72 minutes) ie:

```
TICK_COUNT start = clockGetus();
... do something ...
TICK_COUNT end = clockGetus();
TICK_COUNT duration = end - start; // the number of uS up to a maximum of 72
minutes
```

NB if this is called from app_init then it will have unknown results as the clock has not yet been created.

## clockHasElapsed

`boolean clockHasElapsed(TICK_COUNT usStart, TICK_COUNT usWait)`

Test if a given duration has elapsed.

Returns TRUE if it has or FALSE if not.

Whilst I wouldn't recommend doing this: here is an example that pauses for 10mS.

```
TICK_COUNT start = clockGetus(); // Get the start time
TICK_COUNT wait = 10000; // 10ms = 10000us
while(clockHasElapsed(start, wait)==FALSE){
    ... still waiting ...
}
```

NB if this is called from app_init then it will have unknown results as the clock has not yet been created.

## clockWaitms

`clockWaitms(TICK_COUNT ms)`

Pause for the given number of milliseconds.

So to wait for 1 second we could use:-

```
clockWaitms(1000);
```

NB The granularity of the clock is such that you should not assume that this waits for exactly 1 second- but rather that it waits for 'at least' one second.

NB if this is called from app_init then it will have unknown results as the clock has not yet been created.

## clockWaitus

`clockWaitus(TICK_COUNT us)`

Pause for the given number of microseconds.

So to wait for 100 microseconds we could use:-

```
clockWaitus(100);
```

NB The granularity of the clock is such that you should not assume that this waits for exactly 100 micro seconds- but rather that it waits for 'at least' that time.

NB if this is called from app_init then it will have unknown results as the clock has not yet been created.

## Advanced Function Detail

### clockHasElapsedGetOverflow

`boolean clockHasElapsedGetOverflow(TICK_COUNT usStart, TICK_COUNT usWait, TICK_COUNT* overflow)`

Similar to 'clockHasElapsed' but returns the number of µS left to go if the duration has not elapsed, or the number of µS we have exceeded the duration if is has elapsed.

For example:

```
TICK_COUNT start = clockGetus();
TICK_COUNT wait = 10000; // wait for 10ms
TICK_COUNT overflow;
while(clockHasElapsedGetOverflow(start, wait, &overflow)==FALSE){
    // There are still 'overflow' µS left
}
// overflow has the number of µS in excess of 'wait' that we have actually
waited for
```

NB if this is called from app_init then it will have unknown results as the clock has not yet been created.

## core.h

This file is always included by your project.

It defines some standard macros and data types and also provides the main entry point for your program.

| Standard Function Summary | |
|---|---|
| `int16_t` | interpolate(int16_t value, int16_t minVal, int16_t maxVal, int16_t minRtn, int16_t maxRtn)<br>This will interpolate a value from one range of values into its equivalent in another range of signed numbers by using the following parameters:-value - The value from range 1minVal,MaxVal - specify the start and end of range 1minRtn,maxRtn - specify the start and end of range 2The function will convert 'value' into its equivalent in range 2. |
| `uint16_t` | interpolateU(int16_t value, int16_t minVal, int16_t maxVal, uint16_t minRtn, uint16_t maxRtn)<br>This will interpolate a value from one range of values into its equivalent in another range of unsigned numbers by using the following parameters:-value - The value from range 1minVal,MaxVal - specify the start and end of range 1minRtn,maxRtn - specify the start and end of range 2The function will convert 'value' into its equivalent in range 2. |
| `double` | interpolatef(double value, double minVal, double maxVal, double minRtn, double maxRtn)<br>This will interpolate a floating point value from one range of values into its equivalent in another range of signed numbers by using the following parameters:-value - The value from range 1minVal,MaxVal - specify the start and end of range 1minRtn,maxRtn - specify the start and end of range 2The function will convert 'value' into its equivalent in range 2. |
| `uint32_t` | isqrt(uint32_t x)<br>Performs a fast square root function on a whole number - returning the nearest whole number answer and without requiring the floating point library. |
| | delay_cycles(uint32_t __cycles)<br>Pause for the given number of clock cycles. |

## Standard Function Summary

| | |
|---|---|
| | delay_ms(uint32_t ms)<br>Pause for the given number of ms. |
| | delay_us(uint32_t us)<br>Pause for the given number of µs. |
| `uint64_t` | MS_TO_CYCLES(ms)<br>Convert milliseconds into a number of cycles. |
| `uint64_t` | US_TO_CYCLES(us)<br>Convert microseconds into a number of cycles. |
| | PRINTF(stream,format, args...)<br>Output a formatted string to any Stream. |

## Advanced Function Summary

| | |
|---|---|
| | memoryDump(FILE* stream,const void* data, size_t offset, size_t len)<br>Dumps an area of memory in tabular format showing both hexadecimal and ASCII values. |

## Standard Function Detail

### interpolate

`int16_t interpolate(int16_t value, int16_t minVal, int16_t maxVal, int16_t minRtn, int16_t maxRtn)`

This will interpolate a value from one range of values into its equivalent in another range of signed numbers by using the following parameters:-

value - The value from range 1

minVal,MaxVal - specify the start and end of range 1

minRtn,maxRtn - specify the start and end of range 2

The function will convert 'value' into its equivalent in range 2.

Mathematically it will return:

```
minRtn+((value-minVal)*(maxRtn-minRtn)/(maxVal-minVal))
```

Example: assume that you have a value in a variable called 'theValue' that stores a value in the range 0 to 2048 and you want to convert this into the range -512 to +512 and store the new value into a variable called 'newValue'. Then you could write:-

```
int16_t newValue = interpolate(theValue, 0, 2048, -512, 512);
```

Here are some examples of what would be returned:-

- if theValue=0 then it returns -512
- if theValue=2048 then it returns +512
- if theValue=1024 then it returns 0

Note that no range checking is performed. So if theValue contained twice the input limit then it will return twice the output limit. ie if theValue=4096 then the returned value will be 1024.

If you want to limit the values to make sure they are within a given range then use the CLAMP command in libdefs.h. For example:-

```
uint16_t newValue = interpolate(theValue, 0, 2048, -512, 512);
newValue = CLAMP(newValue,-512,512); // limit the answer to be in the range -
512 to 512
```

---

### interpolateU

`uint16_t interpolateU(int16_t value, int16_t minVal, int16_t maxVal, uint16_t minRtn, uint16_t maxRtn)`

This will interpolate a value from one range of values into its equivalent in another range of unsigned numbers by using the following parameters:-

value - The value from range 1

minVal,MaxVal - specify the start and end of range 1

minRtn,maxRtn - specify the start and end of range 2

The function will convert 'value' into its equivalent in range 2.

Mathematically it will return:

```
minRtn+((value-minVal)*(maxRtn-minRtn)/(maxVal-minVal))
```

Example: assume that you have a value in a variable called 'theValue' that stores a value in the range 0 to 2048 and you want to convert this into the range 300 to 800 and store the new value into a variable called 'newValue'. Then you could write:-

```
uint16_t newValue = interpolateU(theValue, 0, 2048, 300, 800);
```

Note that no range checking is performed. So if theValue contained twice the input limit then it will return twice the output limit. ie if theValue=4096 then the returned value will be 1600.

If you want to limit the values to make sure they are within a given range then use the CLAMP command in libdefs.h. For example:-

```
uint16_t newValue = interpolateU(theValue, 0, 2048, 300, 800);
newValue = CLAMP(newValue,300,800); // limit the answer to be in the range
300 to 800
```

---

### interpolatef

`double interpolatef(double value, double minVal, double maxVal, double minRtn, double maxRtn)`

This will interpolate a floating point value from one range of values into its equivalent in another range of signed numbers by using the following parameters:-

value - The value from range 1

minVal,MaxVal - specify the start and end of range 1

minRtn,maxRtn - specify the start and end of range 2

The function will convert 'value' into its equivalent in range 2.

Mathematically it will return:

```
minRtn+((value-minVal)*(maxRtn-minRtn)/(maxVal-minVal))
```

Example: assume that you have a value in a variable called 'theValue' that stores a value in the range 0 to 20.48 and you want to convert this into the range -5.12 to +5.12 and store the new value into a variable called 'newValue'. Then you could write:-

```
double newValue = interpolatef(theValue, 0, 20.48, -5.12, 5.12);
```

Here are some examples of what would be returned:-

- if theValue=0 then it returns -5.12
- if theValue=20.48 then it returns +5.12
- if theValue=10.24 then it returns 0.0

Note that no range checking is performed. So if theValue contained twice the input limit then it will return twice the output limit. ie if theValue=40.96 then the returned value will be 10.24.

If you want to limit the values to make sure they are within a given range then use the CLAMP command in libdefs.h. For example:-

```
double newValue = interpolatef(theValue, 0, 20.48, -5.12, 5.12);
newValue = CLAMP(newValue,-5.12,5.12); // limit the answer to be in the range
-5.12 to 5.12
```

---

## isqrt

`uint32_t isqrt(uint32_t x)`

Performs a fast square root function on a whole number - returning the nearest whole number answer and without requiring the floating point library.

## delay_cycles

`delay_cycles(uint32_t __cycles)`

Pause for the given number of clock cycles.

Note that the actual delay will vary depending upon the speed of the processor. In order to remove this dependency it is recommended that you use either the MS_TO_CYCLES or US_TO_CYCLES macro to convert an actual time into the number of clock cycles. For example: to delay by 100uS you should write:

```
delay_cycles(US_TO_CYCLES(100));
```

It is recommended that you use this function, rather than delay_ms or delay_us, if the delay is a fixed value and especially if the delay period is short. The reason being that the delay_ms and delay_us routines will first have to do some multiplications and divisions in order to convert the delay into cycles and the overhead of these calculations can sometimes take longer than the delay you require. Using the MS_TO_CYCLES and US_TO_CYCLES macros for a fixed delay means that the compiler will perform these calculations at compile time and thereby reduce the overhead at runtime.

## delay_ms

`delay_ms(uint32_t __ms)`

Pause for the given number of ms.

This is similar to clockWaitms but will also work in app_init where the clock is not available.

## delay_us

`delay_us(uint32_t __us)`

Pause for the given number of µs.

This is similar to clockWaitus but will also work in app_init where the clock is not available.

## MS_TO_CYCLES

`uint64_t MS_TO_CYCLES(ms)`

Convert milliseconds into a number of cycles.

Because the cpu speed is only known when building an end-user program then this macro is not available if you are writing a new library function.

Also see: delay_ms, delay_cycles

## US_TO_CYCLES

```
uint64_t US_TO_CYCLES(us)
```

Convert microseconds into a number of cycles.

Because the cpu speed is only known when building an end-user program then this macro is not available if you are writing a new library function.

Also see: delay_us, delay_cycles

## PRINTF

```
PRINTF(stream,format, args...)
```

Output a formatted string to any "*Stream*" (see page 365) .

The new 'streams' functionality actually provides much more friendly ways to output data ( such as "*print*" (see page 365) ) where you don't need to understand the link required between the format string and the data type you are showing. However: they have certain functional limits - eg they wont allow you to print a number right-aligned within a given field width. Hence this function is provided as a low-level alternative for those familiar with C format strings and also for those people who are migrating their code away from the old WebbotLib 'rprintf' commands.

When using this function the formatting string is automatically stored in program memory.

Note that due to the way C works then neither the compiler, nor the runtime, can verify that the % entries in the format string match the data type of the given parameters. Consequently: if you attempt to print out various variables in one go and there is a mismatch between the format string and the parameters that follow then C will print out what looks like garbage. If in doubt, or to debug a problem, then only output one value per call.

## Advanced Function Detail

### memoryDump

```
memoryDump(FILE* stream,const void* data, size_t offset, size_t len)
```

Dumps an area of memory in tabular format showing both hexadecimal and ASCII values.

The first parameter specifies where you want the output dumping to.

The second parameter specifies the memory address, the third is the byte offset starting at that address, and the last parameter specifies the number of bytes to dump out.

So if you have an array such as:

```
char buffer[50];
```

Then you can dump out its contents to the standard output using:-

```
memoryDump(stdout, buffer, 0, 50);
```

## <avr/eeprom.h>

Most AVR micro controllers contain some 'on-chip' EEPROM. This is memory that remembers its content even after the power is switched off.

This makes it a useful option if:

1. You have some configuration settings that you want to restore each time the robot is turned on
2. You are using a neural network and you want to retain what has been 'learned'
3. Or simply because you are running out of regular memory and want to move some of your variables or tables into EEPROM to free up some standard memory.

So in some regards you can think of it as a small, on-board, hard drive.

If you need more persistent storage than your micro processor contains then WebbotLib also supports the addition of external EEPROMs and sdCards via "*Storage*" (see page 243)

As well as the above benefits there are, of course, some down sides to using EEPROM memory.

1. It is slower to access than standard memory
2. To use data from EEPROM it first needs to be read into some standard memory. Then if you change its value you must remember to save it back out to EEPROM in order to preserve the value between power ups.
3. There are times (see later) when you don't want your program to restart with the old values

So how do I use EEPROM? Well first I should say that the code documented here is supplied with the compiler and wasn't written by me. All my library does is to automatically include this support if you are compiling for a device that contains any EEPROM. The functions detailed in this section have only been included to make the documentation more complete.

The first thing you need to do is decide which variables you want to store in EEPROM and then add the keyword EEMEM. For example:-

```
int8_t EEMEM myVar = 10;
```

This tells the compiler to make space for your variable in the EEPROM but you then have to change every place where you read or write to the variable.

To access the variable you must read it into a memory variable using 'eeprom_read_byte' or 'eeprom_read_word' (see details of these calls in the following pages). If you modify the value and want to write it back into EEPROM so that it is remembered you must then call the matching 'eeprom_write_byte' or 'eeprom_write_word'.

That covers simple numeric types but what about things like arrays and strings? These can be defined in the same way by pre-fixing their definition with EEMEM but you need to use the 'eeprom_read_block' and 'eeprom_write_block' calls.

Now that we have a basic understanding of how to access the EEPROM data we will now look at some of the potential pitfalls and suggest some better ways of working.

Assuming that you are storing more than one variable in EEPROM then you have to be very carefull when you are modifying your program. When the compiler finds data you want storing in EEPROM that it assigns them a location in the order that it comes across them. So assuming your program contains the following variables:-

```
int8_t EEMEM myVar1 = 10;
int8_t EEMEM myVar2 = 20;
```

These initial values will be placed into your hex file and assuming that your programmer is set up to transfer the eeprom data then these variables will all have been loaded into the EEPROM and all is ok.

Now lets add a new variable in the middle:-

```
int8_t EEMEM myVar1 = 10;
int8_t EEMEM myVar3 = 30;
int8_t EEMEM myVar2 = 20;
```

When you send this to your programmer, and it is set up to transfer the EEPROM data, then the new variables will be transferred across successfully but any changes to myVar1 and myVar2 will be lost.

The big problem happens when you don't send the new EEPROM data across (say because you wanted to keep the values of myVar1 and myVar2 from the last time you ran the program). The problem is that the new myVar3 is occupying the slot in EEPROM that used to be occupied by myVar2 and so will pickup its last written value. Equally: myVar2 is now in a previously unused location and so will pickup some random value.

So the lesson is that you add, edit, or delete any EEPROM variables then you will need to re-program the new EEPROM data into the micro controller and lose the previous values.

Some developers choose to write routines that save and restore all of the EEPROM variables in one go. They do this because it can be a bit of a pain to remember that for each EEMEM you have to change every access to the variable and also because if the variables are accessed frequently it can slow the program down. The easiest way of doing this is to create a structure or 'struct' (google is your friend!) that defines the data you want saving to EEPROM. So we could change the previous code to look like this:-

Step 1 - create a new data type called 'SAVED_DATA' that contains the data that needs to be held in eeprom:-

```
typedef struct s_eeprom {
    int8_t myVar1;
    int8_t myVar2;
    int8_t myVar3;
} SAVED_DATA;
```

Step 2 - create an EEMEM variable so that space is allocated in EEPROM along with initial values

```
SAVED_DATA EEMEM eeprom = { 10,30, 20};
```

Step 3 - create your standard memory variables as you would do normally:-

```
int8_t myVar1;
int8_t myVar2;
int8_t myVar3;
```

Step 4 - write a routine to read the data out of EEPROM and then unpack it into your variables:-

```
void restore(void){
    // Create a memory block with the same structure as the eeprom
    SAVED_DATA temp;
    // Read the data from eeprom into the 'temp' version in memory
    eeprom_read_block( &temp, &eeprom, sizeof(SAVED_DATA));
    // Now copy the variables out into their proper slots
    myVar1 = temp.myVar1;
    myVar2 = temp.myVar2;
    myVar3 = temp.myVar3;
}
```

Step 5 - write a routine to write the data to EEPROM:-

```
void save(void){
    // Create a memory block with the same structure as the eeprom
    SAVED_DATA temp;
    // Gather all variables to be saved into the structure
    temp.myVar1 = myVar1;
    temp.myVar2 = myVar2;
    temp.myVar3 = myVar3;
    // Now write the block out to eeprom
    eeprom_write_block(&temp, &eeprom, sizeof(SAVED_DATA));
}
```

The benefit of this approach is that you can write your program as if there was no EEPROM at all. Once you decide that a given variable needs to be stored in EEPROM you can just add it to the SAVED_DATA definition and add a line of code to each of save and restore to move the variable. Obviously in order for this to work you will need to call the 'restore' function somewhere at the start of your code to get the values from EEPROM. The program then runs at full speed with these variables in standard memory. It is up to you as to when you choose to save the values to EEPROM and you could do this in your main loop every 20 seconds say or when an event occurs such as a button being pressed.

Thats just one strategy you could use - its certainly not the only strategy - and you may have a better one!

Note that this file is automatically included if your micro processor has some on-board EEPROM.

## Standard Function Summary

| | |
|---|---|
| `uint8_t` | eeprom_read_byte (const uint8_t * __p)<br>Read a byte from EEPROM. |
| | eeprom_write_byte (uint8_t * __p, uint8_t __value)<br>Writes a byte to EEPROM. |
| `uint16_t` | eeprom_read_word(const uint16_t* __p)<br>Read a word (ie a 16 bit number) from EEPROM. |
| | eeprom_write_word(uint16_t* __p, uint16_t __value)<br>Writes a word (ie a 16 bit number) to EEPROM. |
| `uint32_t` | eeprom_read_dword(const uint32_t* __p)<br>Read a double word (ie a 32 bit number) from EEPROM. |
| | eeprom_write_dword(uint32_t* __p, uint32_t __value)<br>Writes a double word (ie a 32 bit number) to EEPROM. |
| | eeprom_read_block (void * __dst, const void * __src, size_t __n)<br>Read a block of data from EEPROM into standard memory. |
| | eeprom_write_block (const void * __src, void * __dst, size_t __n)<br>Writes a block of data to EEPROM from standard memory. |

## Standard Function Detail

### eeprom_read_byte

`uint8_t eeprom_read_byte (const uint8_t *__p)`

Read a byte from EEPROM.

Assuming you have a global variable declared as:-

```
uint8_t EEMEM myVar;
```

Then you can read its current value into memory inside a function by calling:-

```
uint8_t currentValue = eeprom_read_byte(&myVar);
```

---

### eeprom_write_byte

`eeprom_write_byte (uint8_t *__p, uint8_t __value)`

Writes a byte to EEPROM.

Assuming you have a global variable declared as:-

```
uint8_t EEMEM myVar;
```

Then you can assign it a value of 10 by calling:-

```
eeprom_write_byte(&myVar, 10);
```

---

## eeprom_read_word

`uint16_t eeprom_read_word(const uint16_t* __p)`

Read a word (ie a 16 bit number) from EEPROM.

Assuming you have a global variable declared as:-

```
uint16_t EEMEM myVar;
```

Then you can read its current value into memory inside a function by calling:-

```
uint16_t currentValue = eeprom_read_word(&myVar);
```

---

## eeprom_write_word

`eeprom_write_word(uint16_t* __p, uint16_t __value)`

Writes a word (ie a 16 bit number) to EEPROM.

Assuming you have a global variable declared as:-

```
uint16_t EEMEM myVar;
```

Then you can assign it a value of 10 by calling:-

```
eeprom_write_word(&myVar, 10);
```

---

## eeprom_read_dword

`uint32_t eeprom_read_dword(const uint32_t* __p)`

Read a double word (ie a 32 bit number) from EEPROM.

Assuming you have a global variable declared as:-

```
uint32_t EEMEM myVar;
```

Then you can read its current value into memory inside a function by calling:-

```
uint32_t currentValue = eeprom_read_dword(&myVar);
```

---

## eeprom_write_dword

`eeprom_write_dword(uint32_t* __p, uint32_t __value)`

Writes a double word (ie a 32 bit number) to EEPROM.

Assuming you have a global variable declared as:-

```
uint32_t EEMEM myVar;
```

Then you can assign it a value of 10 by calling:-

```
eeprom_write_dword(&myVar, 10);
```

## eeprom_read_block

`eeprom_read_block (void *__dst, const void *__src, size_t __n)`

Read a block of data from EEPROM into standard memory.

The first parameter is the address in standard memory that you want to read the data to, and the second parameter is the address of the EEPROM variable. The third parameter is the number of bytes to be copied.

Note that since these are both memory addresses then they are normally prefixed with an '&' symbol.

For example if you had a string variable declared as follows:-

```
char EEMEM eepromString[10];
```

Then you could read it into memory inside a function as follows:-

```
char ramString[10];
eeprom_read_block( &ramString[0], &eepromString[0], 10);
```

## eeprom_write_block

`eeprom_write_block (const void * __src, void *__dst, size_t __n)`

Writes a block of data to EEPROM from standard memory.

The first parameter is the address in standard memory of the data you want to write, and the second parameter is the address of the EEPROM area that you want to write to. The third parameter is the number of bytes to be copied.

Note that since these are both memory addresses then they are normally prefixed with an '&' symbol.

For example if you had a string variable declared as follows:-

```
char EEMEM eepromString[10];
```

Then you could write the EEPROM with the value 'Webbot' as follows:-

```
eeprom_write_block( "Webbot", &eepromString[0], 10);
```

## errors.h

"*All programs run properly and never come across an unexpected situation*" - said the inexperienced programmer!

The problem with microcontrollers is that you cannot assume that they have a screen. So how can they tell you that something has gone wrong? A computer can give you the '*blue screen of death*' or a '*This program has terminated unexpectedly*' message so at least you know something has gone wrong. But how do we do that on something without a screen? Luckily most boards come with at least one LED that the program can control.

So this library uses one of these LEDs to indicate when there is a problem. If the board has no LEDs then we can't help! Each system file will register the LED to be used for errors - assuming that one is present on the board. Once an error is reported via 'setError' then this LED will start flashing.

Each error has its own unique number. This library only uses negative error numbers - but your code should only use positive error numbers but obviously you can re-use a given error number from one project to the next.

Library errors make the LED blink at twice the speed as your own errors but they both have a 2 second gap to signal the start of the count. Counting the blinks will give you the error number.

Once an error has been reported and the LED starts blinking then any future calls to setError will be ignored - ie the LED only reports the first error. Fix it and then you may find others.

This file defines each of the error numbers used by the library itself. The file gives details on each of the errors and, once you get an error, you should read the entry as it doesn't always mean that the problem is in my code - it may be that your code has called mine with incorrect parameters.

| Standard Function Summary | |
|---|---|
| | setError(ERROR_CODE err)<br>Indicate that an error has happened. |
| ERROR_CODE | getError()<br>Returns the current error code or 0 if there is no error. |
| | setErrorLog(Writer log)<br>Specify an alternative location to send error messages to. |

### Standard Function Detail

#### setError

```
setError(ERROR_CODE err)
```

Indicate that an error has happened.

The parameter specifies the error number.

## getError

`ERROR_CODE getError()`

Returns the current error code or 0 if there is no error.

You may decide that if there is an error that you want your robot to stop and shutdown once an error situation has been met. You could achieve this in your main appControl loop as follows:-

```
#include "errors.h"
void appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    if(getError()!=0){
        // We've got an error. So stop all motors.
    }else{
        // There are no errors - so perform one iteration of the main loop
    }
}
```

## setErrorLog

`setErrorLog(Writer log)`

Specify an alternative location to send error messages to.

The status led can be used to flash error messages but sometimes the micro-controller board either has no on board LED or the board is buried in a shell and so the LED cannot be seen.

This function allows you to specify an alternative routine to display the error message.

This could be a UART (which has previously been initialised to the required baud rate) using:-

setErrorLog(uart1SendByte);

## libdefs.h

Defines various macros and datatypes that are used in the rest of the library.

| Standard Function Summary | |
|---|---|
| | MIN(a,b)<br>This macro takes two numbers as its arguments and will return the smallest one ie the one closest to negative infinity. |
| | MAX(a,b)<br>This macro takes two numbers as its arguments and will return the largest one ie the one closest to positive infinity. |
| | ABS(x)<br>This macro takes one number as its arguments and will return the absolute value. |
| | CLAMP(val, min, max)<br>A macro to clamp a value to be within a given range. |
| `uint16_t` | get_uint16(const void* buf,size_t offset)<br>Returns a 16 bit integer from a buffer. |
| `int16_t` | get_int16(const void* buf,size_t offset)<br>Returns a 16 bit signed integer from a buffer. |
| `uint32_t` | get_uint32(const void* buf,size_t offset)<br>Returns a 32 bit integer from a buffer. |
| | set_uint16(const void* buf,size_t offset,uint16_t data)<br>Stores a 16 bit integer into a buffer. |
| | set_uint32(const void* buf,size_t offset,uint32_t data)<br>Stores a 32 bit integer into a buffer. |

| Advanced Function Summary | |
|---|---|
| | CRITICAL_SECTION<br>Bracket a section of code that shoud not be interrupted. |
| | CRITICAL_SECTION_START<br>Deprecated |
| | CRITICAL_SECTION_END<br>Deprecated |

## Advanced Function Summary

| | |
|---|---|
| `uint8_t` | READMEMBYTE(rom,char_ptr) This macro will read a byte either from ROM or from RAM given the following parameters:-rom - If FALSE it reads from RAM, if TRUE then it assumes the byte was declared to be in ROMchar_ptr - The address of the byte you want to read. |

## Standard Function Detail

### MIN

`MIN(a,b)`

This macro takes two numbers as its arguments and will return the smallest one ie the one closest to negative infinity.

### MAX

`MAX(a,b)`

This macro takes two numbers as its arguments and will return the largest one ie the one closest to positive infinity.

### ABS

`ABS(x)`

This macro takes one number as its arguments and will return the absolute value. ie if the number starts with a '-' then it will change it to a '+'.

### CLAMP

`CLAMP(val, min, max)`

A macro to clamp a value to be within a given range.

You can use this macro to make sure that a values lies within a given range. For example: if you have a variable called 'value' and you want to make sure that its contents is in the range 10 to 99 then you can use:

```
value = CLAMP(value, 10, 99);
```

If the initial value is less than 10 then it will be set to 10. If it is greater than 99 then it will be set to 99.

### get_uint16

`uint16_t get_uint16(const void* buf,size_t offset)`

Returns a 16 bit integer from a buffer.

This assumes the data is stored as least significant byte then most significant byte.

The parameters specify a memory array and a byte offset into that array.

---

### get_int16

```
int16_t get_int16(const void* buf,size_t offset)
```

Returns a 16 bit signed integer from a buffer.

This assumes the data is stored as least significant byte then most significant byte.

The parameters specify a memory array and a byte offset into that array.

---

### get_uint32

```
uint32_t get_uint32(const void* buf,size_t offset)
```

Returns a 32 bit integer from a buffer.

This assumes the data is stored from least significant byte to most significant byte.

The parameters specify a memory array and a byte offset into that array.

---

### set_uint16

```
set_uint16(const void* buf,size_t offset,uint16_t data)
```

Stores a 16 bit integer into a buffer.

This assumes the data is stored as least significant byte then most significant byte.

The parameters specify a memory array, a byte offset into that array, and the value to be written.

---

### set_uint32

```
set_uint32(const void* buf,size_t offset,uint32_t data)
```

Stores a 32 bit integer into a buffer.

This assumes the data is stored from least significant byte thorugh to most significant byte.

The parameters specify a memory array, a byte offset into that array, and the value to be written.

## Advanced Function Detail

### CRITICAL_SECTION

`CRITICAL_SECTION`

Bracket a section of code that shoud not be interrupted.

Since most ATMegas are 8 bit devices then you must be careful when accessing any memory variable that is greater than 8 bit in size as it will require more than one instruction to read the variable. If the variable is not changed by any interrupt routine then all is OK.

However: if the variable is changed by an interrupt routine then you must do several things:

1. Mark the variable as 'volatile' so that the compiler knows that it always needs to read the current value rather than keeping old copies lying around in registers.
2. Any foreground (ie non-interrupt service) routine that reads the variable must make sure that it is not changed by an interrupt whilst it is being read.

To show you what I mean then assume you have a 16 bit number that is also changed in an interrput then you would need to declare it as:

```
volatile uint16_t myNumber;
```

Since a 16 bit number is two bytes then it requires two instructions to read it (low byte first, then high byte). Lets assume it currently holds the number 255 (ie High byte=0, low byte=255).

Assume we try to read it like this:

```
uint16_t myCopy = myNumber;
```

Then the compiler will generate code to read the low byte first (ie 255) then the high byte (ie 0) but since it requires two instructions then there is nothing to stop an interrupt happening in the middle. Lets assume that the interrupt routine adds one to the variable:

```
myNumber++;
```

so that the low byte is now 0 and the high byte is now 1.

This causes a major problem as our main code has read the old low byte (255) and then reads the new high byte (1) - so the answer it gets is 1*256 + 255 = 511. Of course this is neither the old number (255) nor the new number (256) - its a mixture of both!!!!

You can prevent this by using:

```
CRITICAL_SECTION{
    uint16_t myCopy = myNumber;
}
```

During the bracketed section then all interrupts will be disabled meaning that you read the proper values but this code should be kept as small, ie quick, as possible. If you leave interrupts disabled for too long then you may miss key events such as a character arriving on a UART etc.

## CRITICAL_SECTION_START

`CRITICAL_SECTION_START`

**Deprecated** - use CRITICAL_SECTION instead

## CRITICAL_SECTION_END

`CRITICAL_SECTION_END`

**Deprecated** - use CRITICAL_SECTION instead

## READMEMBYTE

`uint8_t READMEMBYTE(rom,char_ptr)`

This macro will read a byte either from ROM or from RAM given the following parameters:-

rom - If FALSE it reads from RAM, if TRUE then it assumes the byte was declared to be in ROM

char_ptr - The address of the byte you want to read.

## pid.h

Implements a PID (Proportional, Integral, Derivative) control loop.

A PID control loop allows you to optimise how 'something' changes from A to B. The 'something' in question can be anything which has a 'measurable goal' and a 'measurable progress'.

As an example: you may want your motors to turn at a given RPM (measurable goal) and so long as you have some encoders then you can measure how quickly they are currently turning (measurable progress). A PID allows you to reach the required RPM as quickly as possible. And if your robot starts going up a hill then the RPM will drop and so the robot will apply more power to the motors to try to keep the speed constant.

But this could apply to many other 'measurable' things where the robot can directly effect the output. Servo locations, speed, acceleration, bearing and position to name but a few. However: it is not suitable for goal seeking where the robot cannot directly effect the result such as maze solving. Although this has a measurable goal (the destination) and a measurable progress (the distance from the goal) there is no magical action the robot can make in order to get closer as it depends on lots of other things it cannot control (like where there are walls!).

A 'practical' example of a control loop: consider a tank moving over rough ground. It spots a target and can work out the gun barrel angle to fire a shell to hit the target. But the tank barrel is heavy, so moves slowly, and the rough ground makes the problem worse. How do we keep the barrel at the correct location?

The theoretical programming use of PID is very easy: tell it your goal once, and then keep updating it with your current progress. ie tell the PID the angle you want for the barrel and then keep updating it with the current true position (taking into account the rough ground). The PID will output the amount of force to use to lift/lower the barrel.

However: the 'tuning' of your PID system is partly manual trial-and-error or can be based on certain calculations like Ziegler–Nichols. These tunings are critical to get to the goal in the optimum time but they depend on all sorts of things - ie the weight of the tank barrel, the maximum torque of the motors that move it etc etc. So sorry .... no quick fix. You have to tune the PID to your set up.

Don't despair: there is lots of web info about PID on the net. Such as:-

http://en.wikipedia.org/wiki/PID_controller

There is also lots of info about the best way to tune your PID.

WebbotLib provides a PID that can be used in two situations. Standard and circular.

What's a circular system. Well think of a tank turret. It can move clockwise or anti-clockwise and both movements will end up getting to the correct place - a circular PID will choose the best way to go.

We provide two constructors. For a standard PID use:-

```
#include "pid.h"
PID myPID = MAKE_PID(kP,kI,kD, il, outMin, outMax);
```

or for a circular PID:-

```
#include "pid.h"
PID myPID = MAKE_CIRCULAR_PID(kP,kI,kD, il, outMin, outMax, inMin, inMax);
```

The kp, kI, kD values are the standard constants used by the PID theory.

il - is the maximum value of the integral component and is used to stop the integral from swamping the system.

outMin/outMax - are the range of values you want to be returned by the PID. If you are using it to drive a motor, for example, then the returned speed would be need to be between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX.

For a circular PID the additional inMin/inMax values are the range of input values that represent a full circle. If the input value was from an encoder then these values would be the encoder values that represent a position of 0° and 360°.

## Standard Function Summary

| | |
|---|---|
| `void` | pidSetTarget(PID* pid,float setPoint)<br>Specify the goal, or 'set point' for the PID. |
| `float` | pidSetActual(PID* pid,float actual)<br>Updates the PID with the current measurement and returns the output value. |

## Standard Function Detail

### pidSetTarget

```
void pidSetTarget(PID* pid,float setPoint)
```

Specify the goal, or 'set point' for the PID.

This is the value you want to achieve.

### pidSetActual

```
float pidSetActual(PID* pid,float actual)
```

Updates the PID with the current measurement and returns the output value.

The output value is the value that is used to 'drive' the 'thing' you are measuring.

## pinChange.h

Allow callback routines to be attached to IOPins that are called when the pin changes.

This relies upon hardware interrupts and is not provided by the ATMega8, ATMega32 or ATMega128. The ATMega168, ATMeag328P, ATMega640 and ATMega2560 provide the feature on 24 of the pins. The ATMega2561 only provides the feature on 8 pins. Check the datasheet for pins labelled PCINTnn. Trying to use this file when compiling for devices that do not provide the feature will result in a compilation error: "This device does not support Pin Change interrupts".

Note: it does not support the alternative INT pins - only the PCINT pins.

The interrupts occur whenever the pin changes from high to low or from low to high and will happen whether the pin is an input pin or an output pin. This allows us to monitor incoming pulses and is used to support other library functions such as Sensors/Encoder/quadrature.h.

A simple logging routine could be created that sets up callbacks for each I/O pin and logs each pin change over the UART to a PC. This could be done without changing anything in your main program and could be removed at a later date once your program is debugged.

### Standard Function Summary

| | |
|---|---|
| | pin_change_attach(const IOPin* io,PinChangeCallback callback, void* data)<br>Attach a callback routine. |
| | pin_change_dettach(const IOPin* io)<br>Remove any callback routine from the given IOPin. |

### Standard Function Detail

#### pin_change_attach

```
pin_change_attach(const IOPin* io,PinChangeCallback callback, void*
data)
```

Attach a callback routine.

Each pin can only have one callback routine attached at a time and will be called via an interrupt so it should be as fast as possible.

An error is generated if the IOPin doesn't have a hardware pin change interrupt associated with it or if there is already a callback attached.

The first parameter is the IOPin to attach the callback to.

The second parameter is the address of your callback routine.

The third parameter is a user defined pointer to some data that can be passed into the routine when it is called. If you don't need it then set it to 'null'.

The callback routine itself should have the following signature:

```
void myCallback(const IOPin* io,boolean val, volatile void* data)
```

Where the first parameter is the pin that has changed, the second parameter is the new value for the pin (TRUE=high, FALSE=low), and the last parameter is the 'data' value you specified when attaching the callback.

The same callback routine can be attached to as many pins as you like.

Example - to create a callback that is called when B4 changes:

```
#include "pinChange.h"

void myCallback(const IOPin* io,boolean val, volatile void* data){
    ... the pin has changed ...
}

// Set up the callback in my initialisation code
void appInitHardware(){
    pin_change_attach(B4, &myCallback, null);
}
```

## Practical pin change example

Assume that we have two push button switches on B4 and B5 and we want to count how many times each one has been pushed or released.

All of the following solutions will need the buttons to be defined and initialised - so they will all start with this code snippet:

```
#include "switch.h"
#include "iopin.h"
#include "pinChange.h"
SWITCH button1 = MAKE_SWITCH(B4);
SWITCH button2 = MAKE_SWITCH(B5);
volatile int count1; // Changes for button1
volatile int count2; // Changes for button2
void appInitHardware(){
    SWITCH_init(&button1);
    SWITCH_init(&button2);
    count1 = 0;
    count2 = 0;
}
```

Solution 1 - have a call back for each switch:-

```
void myCallback1(const IOPin* io,boolean val, volatile void* data){
    count1 = count1 + 1;
}
void myCallback2(const IOPin* io,boolean val, volatile void* data){
    count2 = count2 + 1;
}
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    pin_change_attach(button1.pin, &myCallback1, null);
    pin_change_attach(button2.pin, &myCallback2, null);
}
```

Solution 2. Wait a minute! The call backs are doing the same thing but to a different variable. So we could just have one call back.

```
void myCallback(const IOPin* io,boolean val, volatile void* data){
    if( io == button1.pin){
        count1 = count1 + 1;
    }else{
        count2 = count2 + 1;
    }
}
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    pin_change_attach(button1.pin, &myCallback, null);
    pin_change_attach(button2.pin, &myCallback, null);
}
```

Solution 3. The above code is messy in that if we add more switches then we have to remember to change appInitSoftware and the call back routine to add the new switches. So this solution will make use of the last parameter to pin_change_attach to tell the call back routine which variable should be changed:-

```
void myCallback(const IOPin* io,boolean val, volatile void* data){
    // We know that the 'data' is a pointer to an 'int' variable
    int* ptr = (int*)data;
// Increment the integer
    *ptr = *ptr + 1;
}
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    pin_change_attach(button1.pin, &myCallback, &count1);
    pin_change_attach(button2.pin, &myCallback, &count2);
}
```

This is easier to maintain as we can add as many new switches as we like without having to change the call back routine.

## pin_change_dettach

`pin_change_dettach(const IOPin* io)`

Remove any callback routine from the given IOPin.

## rprintf.h

There are times when you need to create a formatted message that is made of a mixture of text and numbers. Normally this is used for logging purposes to log messages to a PC or LCD display.

This module provides various functions to allow you to do that - using the standard C 'printf' format. The specification is quite wide and includes the output of real numbers (ie numbers with something to the right of the decimal point).

When generating your code in Project Designer you can select from three different options for printf support. These give a trade off between functionality and the amount of program space required.

Since we don't have a screen, unlike a PC say, then we need to tell the rprintf module where the text output is to be sent by default - and this is done in the code generation dialogue box in Project Designer. Leaving the option blank will call all of the output to be discarded.

NOTE: these functions are retained for now - but you should consider using "*PRINTF(stream,format, args...)*" (see page 270) or the stream output commands like "*print*" (see page 365) instead.

**Standard Function Summary**

| | |
|---|---|
| `Writer` | rprintfInit(Writer writer)<br>Tells rprintf where to send its output to. |
| | OUTPUT(Writer writer)<br>Brackets a piece of code whose rprintf output should be sent to a specific place. |
| | rprintfChar(unsigned char c)<br>Output a single character. |
| | rprintfCharN(unsigned char c, size_t repeat)<br>Print a character a given number of times. |
| | rprintfStr(char str[])<br>Print a string that is stored in RAM. |
| | rprintfStrLen(char str[], size_t start, size_t len)<br>Print a section of string that is stored in RAM. |
| | rprintfProgStr(const prog_char str[])<br>Output a constant string stored in program memory. |

## Standard Function Summary

| | |
|---|---|
| | rprintfProgStrM(string)<br>Output a fixed string from program memory. |
| | rprintfCRLF()<br>Move the output to a new line. |
| | rprintfu04(unsigned char data)<br>Output the low 4 bits of 'data' as one hexadecimal character |
| | rprintfu08(unsigned char data)<br>Output the parameter as two hexadecimal characters. |
| | rprintfu16(unsigned short data)<br>Output the parameter as 4 hexadecimal digits. |
| | rprintfu32(unsigned long data)<br>Outputput the parameter as an 8 digit hexadecimal number. |
| | rprintfNum(char base, char numDigits, boolean isSigned, char padchar, long n)<br>Allows you to print a number in any base using different padding characters. |
| | rprintf(format, args...)<br>Output a formatted string to the output. |

## Advanced Function Summary

| | |
|---|---|
| `char` | hexchar(char x)<br>Convert binary to hexadecimal digit. |
| | rprintfFloat(char numDigits, double x)<br>Prints a floating point number. |
| | rprintfMemoryDump(const void* data, size_t offset, size_t len)<br>Dumps an area of memory in tabular format showing both hexadecimal and ASCII values. |

## Standard Function Detail

### rprintfInit

```
Writer rprintfInit(Writer writer)
```

Tells rprintf where to send its output to.

This has now been replaced by "*OUTPUT(Writer writer)*" (see page ) which is a much more forgiving way of making sure that the previous state is restored.

The parameter can specify the address of any function that takes a character as an argument. The most usual case is that characters are sent via a UART to the PC or a serial LCD. Note that if you are using a UART then you should have initialised it to the correct baud rate using uartInit(UART* uart, BAUD_RATE baud) before calling any of the rprintf functions.

To specify that data will be send to UART0 at 9600 you can use the following code:-

```
uartInit(UART0, 9600);
rprintfInit(&uart0SendByte);
```

If you want to use a different UART then just change the number in the line above. For example: if your controller has a UART2 then you can send info to it by using:

```
uartInit(UART2, 9600);
rprintfInit(&uart2SendByte);
```

If you want to write your code that will receive the output from rprintf functions rather than using a UART then you need to create a Writer. This can be done using the MAKE_WRITER macro which expects one parameter which is the name of your routine. The routine will receive the character in a variable called 'byte'. For example: lets say we want to create a Writer that converts everything to upper case before sending to UART1 then we could write:-

```
MAKE_WRITER(upperCaseWriter){
    // Convert the byte to upper case
    if( byte >= 'a' && byte <='z'){
        byte -='a';
        byte += 'A';
    }
    return uart1SendByte(byte);
}
```

You can then tell rprintf to use this routine by:

```
rprintfInit(&upperCaseWriter);
```

Note that if your program keeps changing where rprintf sends its output then it is good practise to make the start of your routine save the current rprintf destination, and then to restore it at the end of your routine. We can do this by saving the value returned by rprintfInit as this is the previous rprintf writer. So for example: say we have some code that will use our 'upper case' writer defined above - but during the rest of our application we want rprintf to go somewhere else. We should write the code as follows:-

```
// Change to our upper case writer and save the previous one
Writer old = rprintfInit(&upperCaseWriter);
... do stuff ....
... all rprintf output will be converted to upper case ...
... and sent out to UART1 ...
// Restore the previous writer
rprintfInit(old);
... Any rprintf output now goes to wherever it was going before ...
```

## OUTPUT

`OUTPUT(Writer writer)`

Brackets a piece of code whose rprintf output should be sent to a specific place.

Previously you would have temporarily changed the rprintf destination with code like this:-

```
// Change to our upper case writer and save the previous one
Writer old = rprintfInit(&upperCaseWriter);
... do stuff ....
... all rprintf output will be converted to upper case ...
... and sent out to UART1 ...
// Restore the previous writer
rprintfInit(old);
... Any rprintf output now goes to wherever it was going before ...
```

This can now be made simpler:-

```
// Change to our upper case writer and save the previous one
OUTPUT(&upperCaseWriter){
    ... do stuff ....
    ... all rprintf output will be converted to upper case ...
.   ... and sent out to UART1 ...
} // restores rprintf to where it was before
... Any rprintf output now goes to wherever it was going before ...
```

This new format is also safe in that the rprintf destination is automatically restored however you exit the loop. For example:-

```
// Change to our upper case writer and save the previous one
OUTPUT(&upperCaseWriter){
    return;    // EVEN THIS WILL RESTORE THE RPRINTF LOCATION !!
} // restores rprintf to where it was before
... Any rprintf output now goes to wherever it was going before ...
```

## rprintfChar

`rprintfChar(unsigned char c)`

Output a single character.

This will translate any 'linefeed' characters into 'carriage return' + 'line feed' sequences.

## rprintfCharN

`rprintfCharN(unsigned char c, size_t repeat)`

Print a character a given number of times.

For example to print a dashed line you could use:

```
rprintfCharN('-',60);
```

## rprintfStr

`rprintfStr(char str[])`

Print a string that is stored in RAM.

For example:

```
rprintfStr("Hello World");
```

This is not ideal since the fixed text 'Hello World' will actually eat into our valuable RAM space.

## rprintfStrLen

`rprintfStrLen(char str[], size_t start, size_t len)`

Print a section of string that is stored in RAM.

This will print out 'length' characters starting at the 'start' position of the string.

For example:

```
rprintfStrLen("Hello World",6,4);
```

will print out 'Worl'.

## rprintfProgStr

`rprintfProgStr(const prog_char str[])`

Output a constant string stored in program memory.

This function is normally only used when the text to be printed is not a fixed value. For example: you may have a function that takes a string as one of its arguments:

```
void myFunc(const char PROGMEM text[]){
    // .. do something ..
    rprintfProgStr(text); // output the text
}
```

Another common use is where there is a piece of text that you are outputting from lots of different places. For example you may be output "OK" in lots of places in your code. In this case we just want to define the text in one place and then reference it from all the places where we output the message. We could do that as follows:

```
const char PROGMEM msg[] = "OK";
```

and whenever we want to output the message we can just write:

```
rprintfProgStr(msg);
```

Also see: rprintfProgStrM

## rprintfProgStrM

`rprintfProgStrM(string)`

Output a fixed string from program memory.

This will store the string in program memory and use rprintfProgStr(const prog_char str[]) to output it. For example:

```
rprintfProgStrM("OK");
```

## rprintfCRLF

`rprintfCRLF()`

Move the output to a new line.

Sends a carriage return and line feed to the output device.

## rprintfu04

`rprintfu04(unsigned char data)`

Output the low 4 bits of 'data' as one hexadecimal character

## rprintfu08

`rprintfu08(unsigned char data)`

Output the parameter as two hexadecimal characters.

### rprintfu16

`rprintfu16(unsigned short data)`

Output the parameter as 4 hexadecimal digits.

---

### rprintfu32

`rprintfu32(unsigned long data)`

Outputput the parameter as an 8 digit hexadecimal number.

---

### rprintfNum

`rprintfNum(char base, char numDigits, boolean isSigned, char padchar, long n)`

Allows you to print a number in any base using different padding characters.

base - The number base you want to use. eg 10 = decimal, 16=hexadecimal

numdigits - The number of digits you want to pad the number out to

isSigned - Set to TRUE if the number is a signed number, ele FALSE

padchar - The character to use on the left hand side to pad the number out to the appropriate number of digits. Normally a space ' '.

n - The value to be printed.

---

### rprintf

`rprintf(format, args...)`

Output a formatted string to the output.

The capability of this function is effected by the 'printf' option you have selected during code generation.

The 'minimal' option allows you to print whole numbers, characters and strings. 'double's and 'float's will be output as '?'.

The 'floating point' option enables the full functionality including floating point numbers.

Here are some examples:

```
#include "rprintf.h"
int16_t num1 = 1234; // a standard 'signed int'
int32_t num2 = 1234567; // a standard 'long int'
int16_t width = 5;
rprintf("num1=%d num2=%ld", num1, num2); // prints 'num1=1234 num2=1234567'
// If we want a 5 digit number with leading zeros...
rprintf("num1=%05d",num1); // prints 'num1=01234'
// Or if the field width needs to be variable then
rprintf("num1=%0*d", width, num1); // prints 'num1=01234'
```

When using this function the formatting string is automatically stored in program memory.

Note that due to the way C works then neither the compiler, nor the runtime, can verify that the % entries in the format string match the data type of the given parameters. Consequently: if you attempt to print out various variables in one go and there is a mismatch between the format string and the parameters that follow then C will print out what looks like garbage. If in doubt, or to debug a problem, then only output one value per call to rprintf.

Here is a list of the various '%' options available in the format string along with the data type which should be passed as a parameter:

- %s Expects a 'char *', 'unsigned char*' or 'uint8_t *'
- %d Expects an 'int', 'int8_t' or an 'int16_t'
- %ld Expects a 'long' or 'int32_t'
- %u Expects an 'uint8_t' or an 'uint16_t'
- %lu Expects an 'unsigned long' or 'uint32_t'
- %o Expects an 'uint8_t' or an 'uint16_t' and outputs the number in octal (base 8)
- %lo Expects an 'unsigned long' or 'uint32_t' and outputs the number in octal (base 8)
- %x Expects an 'uint8_t' or an 'uint16_t' and outputs the number in hexadecimal (base 16)
- %lx Expects an 'uint32_t' and outputs the number in hexadecimal (base 16)
- %% Expects no parameter and just outputs the '%' character
- %c Expects a 'char', 'signed char', 'int8_t', 'int16_t', 'int'
- %f Expects a 'float' or 'double'

## Advanced Function Detail

### hexchar

`char hexchar(char x)`

Convert binary to hexadecimal digit.

Convert the low four digits of the parameter into a hexadecimal character.

---

### rprintfFloat

`rprintfFloat(char numDigits, double x)`

Prints a floating point number.

This is only available if you add the line:-

```
#define RPRINTF_FLOAT
```

prior to including this file.

This is necessary as it will bring in the floating point number library which, if not used by the rest of your code, will add quite a large payload.

This function will convert a floating point number into a string equivalent 'numDigits' long.

---

## rprintfMemoryDump

`rprintfMemoryDump(const void* data, size_t offset, size_t len)`

Dumps an area of memory in tabular format showing both hexadecimal and ASCII values.

The first parameter specifies the memory address, the second is the byte offset starting at that address, and the last parameter specifies the number of bytes to dump out.

So if you have an array such as:

```
char buffer[50];
```

Then you can dump out its contents using:-

```
rprintfMemoryDump(buffer, 0, 50);
```

## scheduler.h

Provides a scheduler mechanism whereby code can be queued up to be called at a later date. NB the scheduler timer cannot be used to time/measure fast events of less than 1ms ie certainly not to send pulses to a servo or for PWM to motors. However it is fine for less demanding applications - such as regular sampling in a Kalman filter.

The scheduler can only queue up a maximum number of jobs. The default is 0 jobs so that this code is only included if needed. To change the setting then, in Project Designer, edit the 'clock' and enter a number in the 'Scheduled jobs' box. This number represents the maximum number of jobs that can be scheduled at any one time. If this number is too small then you will get a 'Scheduler Exhausted' error at run time.

| Advanced Function Summary | |
|---|---|
| | scheduleJob(SchedulerCallback callback, SchedulerData data, TICK_COUNT start, TICK_COUNT delay)<br>Queue up a function to be called at a later date. |

### Advanced Function Detail

#### scheduleJob

`scheduleJob(SchedulerCallback callback, SchedulerData data, TICK_COUNT start, TICK_COUNT delay)`

Queue up a function to be called at a later date. The scheduler only calls your function once - so if you want it to keep running then it will need to keep calling this function to queue itself up again.

This function takes the following parameters:-

callback - The function to be called at a later date. This method must have the following parameters (void * data, TICK_COUNT lasttime, TICK_COUNT overflow).

data - the data structure to be passed back into the data parameter of the callback function

start - The TICK_COUNT, in microseconds, when the current delay starts from. See examples below.

delay - The number of microseconds after 'start' when the callback routine should be called.

Lets assume we only want to callback a function once, after 1000 microseconds from now, and that function is called myCallback then here is the code:-

```
void myCallback(MyData * data, TICK_COUNT lasttime, TICK_COUNT overflow){
    // lasttime is the microsecond time of when I should have been called
    // overflow is the number of microseconds of error between lasttime
    // and when actually called
    .... do what you want to do ....
}
// Then in some other code you can queue up the callback using
scheduleJob(&myCallback, someData, clockGetus(),1000);
```

If you want the code to keep being called back then you will need to change the callback function to keep re-scheduling the job. For example:

```
void myCallback(MyData * data, TICK_COUNT lasttime, TICK_COUNT overflow){
    // lasttime is the microsecond time of when I should have been called
    // overflow is the number of microseconds of error between lasttime
    // and when actually called
    .... do what you want to do ....
    ... and queue up the job again 1000 microseconds after it should have
been called ...
    scheduleJob(&myCallback, someData, lasttime,1000);
}
```

or

```
void myCallback(MyData * data, TICK_COUNT lasttime, TICK_COUNT overflow){
    // lasttime is the microsecond time of when I should have been called
    // overflow is the number of microseconds of error between lasttime
    // and when actually called
    .... do what you want to do ....
    ... and queue up the job again 1000 microseconds from now ...
    scheduleJob(&myCallback, someData, clockGetus(),1000);
}
```

The difference between these last two examples is that the first will start the callback routine 1000 microseconds after the previous call back **started** whereas the second will start the callback routine 1000 microseconds after the previous call back has **finished.**

## timer.h

Provides low level help with timers and compare channels.

If you are trying to measure time delays or get a current 'timer clock' value then you should be looking at "*clock.h*" (see page 262) . The functions here are for much lower-level use!

Timers, in general, are probably the most complex subject when dealing with microprocessors and trying to create code which can port from one device to another especially with the added complexity of different cpu speeds. Timers have different modes: but each timer may only support some of those modes, and each timer may have several channels/pins that are linked to it.

This library attempts to de-mystify some of the above by providing a more intelligent API than other libraries.

The library is aware of the AVR device you are using as well as the clock speed. The library also knows which timers are available for your selected device as well as the modes and channels that it supports.

At the end of the day the 'beginner' doesn't need to worry about this file. There are higher level functions elsewhere to achieve given tasks. So if all of this is going 'over the top of your head' then don't worry - read on...

| **Standard Function Summary** | |
|---|---|
| | timerOverflowAttach(const Timer* timer, TimerCallback callback, void* user_data )<br>Register an overflow callback. |
| | timerOverflowDetach(const Timer* timer)<br>Remove any overflow callback for the given timer. |
| | timerCaptureAttach(const Timer* timer, TimerCallback callback, void* user_data, boolean risingEdge )<br>Register a capture callback. |
| | timerCaptureDetach(const Timer* timer)<br>Remove any capture callback for the given timer. |
| `boolean` | timerSupportsCompare(const Timer* timer)<br>Does the timer have any compare units?Return TRUE if the timer has at least one compare unit. |
| `uint8_t` | timerNumberOfCompareUnits(const Timer* timer)<br>Returns the number of compare units provided by this timer. |

## Standard Function Summary

| | |
|---|---|
| `uint16_t` | timerGetPrescaler(const Timer* timer)<br>Returns the current clock prescaler value for this timer. |
| `boolean` | timerIsInUse(const Timer* timer)<br>Test if a given timer is currently in use. |
| `boolean` | timerIs16bit(const Timer* timer)<br>Is the given timer a 16 bit timer?Return TRUE if it is a 16 bit timer, or FALSE if it is an 8 bit timer. |
| | compareAttach(const TimerCompare* channel, TimerCompareCallback callback, uint16_t threshold, void* data )<br>Attach a callback function to a compare unit. |
| | compareDetach(const TimerCompare* compare)<br>Removes any callback function from the given compare unit. |
| `const TimerCompare*` | compareFromIOPin(const IOPin* pin)<br>Find the compare unit that can be used to toggle the specified IOPin. |
| `const IOPin*` | compareGetPin(const TimerCompare* channel)<br>Returns the IOPin associated with a given compare unit or null if there isn't one or it has no header pin on this board. |
| `uint16_t` | compareGetThreshold(const TimerCompare* channel)<br>Returns the threshold value for the given compare unit. |
| | compareSetThreshold(const TimerCompare* channel, uint16_t threshold)<br>Changes the threshold value for the compare unit. |
| `const Timer*` | compareGetTimer(const TimerCompare* compare)<br>Returns the Timer that this compare unit is associated with. |
| `boolean` | compareIsInUse(const TimerCompare* channel)<br>Tests if a given compare unit is currently in use. |
| `CHANNEL_MODE` | compareGetOutputMode(const TimerCompare* channel)<br>Find the mode used for toggling the PWM output pin of a compare channel when the compare threshold is met. |
| | compareSetOutputMode(const TimerCompare* channel, CHANNEL_MODE mode)<br>Sets the output mode of the compare unit. |

## Standard Function Summary

| | |
|---|---|
| `uint8_t` | NUMBER_OF_TIMERS<br>A constant value representing the number of Timers in the microprocessor. |

## Advanced Function Summary

| | |
|---|---|
| | timerOverflowClearInterruptPending(const Timer* timer)<br>Clears the timers 'overflow interrupt pending' flag. |
| `boolean` | timerOverflowIsInterruptPending(const Timer* timer)<br>Test if an overflow interrupt is pending for the given timer. |
| `boolean` | timerIsCaptureSupported(const Timer* timer)<br>Returns TRUE if this timer supports Input Capture. |
| | timerCaptureClearInterruptPending(const Timer* timer)<br>Clears the timers 'capture interrupt pending' flag. |
| `boolean` | timerCaptureIsInterruptPending(const Timer* timer)<br>Test if a capture interrupt is pending for the given timer. |
| `const IOPin*` | timerGetCapturePin(const Timer* timer)<br>Returns the IOPin used for input capture. |
| `const TimerCompare*` | timerGetCompare(const Timer* timer, uint8_t channel)<br>Find the compare unit for a given timer. |
| `TIMER_MODE` | timerGetMode(const Timer*timer)<br>Returns the current mode for a timer. |
| | timerSetMode(const Timer* timer, TIMER_MODE mode)<br>Set the operating mode for the given timer. |
| `boolean` | timerIsModeSupported(const Timer* timer, TIMER_MODE mode)<br>Test if a timer supports a given mode of operation. |
| `uint16_t` | timerGetBestPrescaler(const Timer* timer, uint16_t repeat_ms)<br>Find the optimum prescaler value for the given timer in order to measure the given number of ms. |
| `boolean` | timerSupportsPrescaler(const Timer* timer, uint16_t prescaler)<br>Does the timer support the given prescaler value? |
| `uint16_t` | timerGetCounter(const Timer* timer)<br>Returns the current counter value for the timer. |

## Advanced Function Summary

| | |
|---|---|
| | timerSetPrescaler(const Timer* timer, uint16_t prescaler)<br>Set the clock prescaler value for the given timer. |
| `uint16_t` | timerGetTOP(const Timer* timer)<br>Returns the current value of TOP for the given timer. |
| | timerOff(const Timer* timer)<br>Tuen off the specified timer. |
| | compareClearInterruptPending(const TimerCompare* channel)<br>Clears the 'interrupt pending' flag for the compare unit. |
| `boolean` | compareIsInterruptPending(const TimerCompare* channel)<br>Test if the compare unit has set the 'interrupt pending' flag. |

## Standard Function Detail

### timerOverflowAttach

```
timerOverflowAttach(const Timer* timer, TimerCallback callback, void*
user_data )
```

Register an overflow callback.

This registers a function that is called every time this timer overflows from TOP to BOTTOM. The final parameter is an optional pointer to some user defined data. The library doesn't chage this data, it is purely there for your own use and may be 'null' if there is none.

Each timer can only have one registered overflow callback routine at a time. If you need to change the callback (unlikely?) then you should use timerOverflowDetach(const Timer* timer) to remove any existing callback prior to calling this routine.

---

### timerOverflowDetach

```
timerOverflowDetach(const Timer* timer)
```

Remove any overflow callback for the given timer.

---

### timerCaptureAttach

```
timerCaptureAttach(const Timer* timer, TimerCallback callback, void*
user_data, boolean risingEdge )
```

Register a capture callback.

This registers a function that is called every time the capture input pin for this timer goes low->high (if last parameter is TRUE) or high->low (if last parameter is FALSE) . The third parameter is an optional pointer to some user defined data. The library doesn't chage this data, it is purely there for your own use and may be 'null' if there is none.

Each timer can only have one registered capture callback routine at a time. If you need to change the callback (unlikely?) then you should use timerCaptureDetach(const Timer* timer) to remove any existing callback prior to calling this routine.

---

### timerCaptureDetach

```
timerCaptureDetach(const Timer* timer)
```

Remove any capture callback for the given timer.

---

### timerSupportsCompare

```
boolean timerSupportsCompare(const Timer* timer)
```

Does the timer have any compare units?

Return TRUE if the timer has at least one compare unit.

---

### timerNumberOfCompareUnits

```
uint8_t timerNumberOfCompareUnits(const Timer* timer)
```

Returns the number of compare units provided by this timer.

Typically this will return a number between 0 and 3.

---

### timerGetPrescaler

```
uint16_t timerGetPrescaler(const Timer* timer)
```

Returns the current clock prescaler value for this timer.

---

### timerIsInUse

```
boolean timerIsInUse(const Timer* timer)
```

Test if a given timer is currently in use.

---

### timerIs16bit

```
boolean timerIs16bit(const Timer* timer)
```

Is the given timer a 16 bit timer?

Return TRUE if it is a 16 bit timer, or FALSE if it is an 8 bit timer.

---

### compareAttach

```
compareAttach(const TimerCompare* channel, TimerCompareCallback
callback, uint16_t threshold, void* data )
```

Attach a callback function to a compare unit.

Only one callback function can be registered per compare unit at a time - see compateDetach.

The callback function is called once the timer value reaches the threshold value. Note that if this value is greater than the value of TOP (as returned by timerGetTOP(const Timer* timer) ) then this will never happen and the callback will never be called.

The first parameter specifies which compare unit.

The second parameter is the address of your function that will be called. This function should have the following signature:

```
void myCallBack(const TimerCompare *timer_compare, void* data);
```

If you don't want a function to be called then use: &nullTimerCompareCallback

The third parameter specifies the value used to trigger the compare match.

The last parameter is for your own use and is often used to pass a reference to some data that needs to be passed into your callback function.

---

## compareDetach

`compareDetach(const TimerCompare* compare)`

Removes any callback function from the given compare unit.

---

## compareFromIOPin

`const TimerCompare* compareFromIOPin(const IOPin* pin)`

Find the compare unit that can be used to toggle the specified IOPin.

Compare units can be used purely for timing purposes but more often they are used to output waveforms (such as PWM) on a given pin. Each compare unit is therefore associated with a given IO pin. This function allows you to locate the compare unit for a given IOPin. It will return 'null' if there is no compare unit associated.

Note that on some devices the same pin is used by more than one timer. In this case priority is given to the 16 bit timer.

For example on an Axon then calling: compareFromIOPin(H4) will return Timer 4 Channel B. However calling: compareFromIOPin(B7) should return Timer 0 Channel A but since this has no header pin on the Axon then it will return 'null'.

---

### compareGetPin
`const IOPin* compareGetPin(const TimerCompare* channel)`

Returns the IOPin associated with a given compare unit or null if there isn't one or it has no header pin on this board.

Also see:compareFromIOPin(const IOPin* pin) for the reverse action

### compareGetThreshold
`uint16_t compareGetThreshold(const TimerCompare* channel)`

Returns the threshold value for the given compare unit.

### compareSetThreshold
`compareSetThreshold(const TimerCompare* channel, uint16_t threshold)`

Changes the threshold value for the compare unit.

### compareGetTimer
`const Timer* compareGetTimer(const TimerCompare* compare)`

Returns the Timer that this compare unit is associated with.

### compareIsInUse
`boolean compareIsInUse(const TimerCompare* channel)`

Tests if a given compare unit is currently in use.

This is useful if you want to use a compare unit for timing purposes rather than generating a waveform on an IOPin. You can iterate through all the timers and compare units to locate one that is not currently in use.

### compareGetOutputMode
`CHANNEL_MODE compareGetOutputMode(const TimerCompare* channel)`

Find the mode used for toggling the PWM output pin of a compare channel when the compare threshold is met.

This will return one of the following values:

CHANNEL_MODE_DISCONNECT - if the output pin is left unchanged

CHANNEL_MODE_TOGGLE - if the output pin is toggled

CHANNEL_MODE_NON_INVERTING - if the output pin is set low

CHANNEL_MODE_INVERTING - if the output pin is set high

### compareSetOutputMode

`compareSetOutputMode(const TimerCompare* channel, CHANNEL_MODE mode)`

Sets the output mode of the compare unit.

The following modes are supported:

```
CHANNEL_MODE_DISCONNECT,
CHANNEL_MODE_TOGGLE,
CHANNEL_MODE_NON_INVERTING,
CHANNEL_MODE_INVERTING
```

### NUMBER_OF_TIMERS

`uint8_t NUMBER_OF_TIMERS`

A constant value representing the number of Timers in the microprocessor.

This can be used to iterate through the list of timers as follows:-

```
for(int8_t t=0; t<NUMBER_OF_TIMERS-1; t++){
const Timer * timer = &pgm_Timers[t];
}
```

## Advanced Function Detail

### timerOverflowClearInterruptPending

`timerOverflowClearInterruptPending(const Timer* timer)`

Clears the timers 'overflow interrupt pending' flag.

Normally only required when reconfiguring a timer so don't use it unless you understand what this means!

### timerOverflowIsInterruptPending

`boolean timerOverflowIsInterruptPending(const Timer* timer)`

Test if an overflow interrupt is pending for the given timer.

It only makes sense to call this when interrupts are disabled or in an interrupt service routine as, if we are in the foreground, then the interrupt will trigger and we will never be able to test for it. This routine will return TRUE if an overflow interrupt is pending and will be triggered when interrupts are re-enabled.

### timerIsCaptureSupported

`boolean timerIsCaptureSupported(const Timer* timer)`

Returns TRUE if this timer supports Input Capture.

### timerCaptureClearInterruptPending

`timerCaptureClearInterruptPending(const Timer* timer)`

Clears the timers 'capture interrupt pending' flag.

Normally only required when reconfiguring a timer so don't use it unless you understand what this means!

---

### timerCaptureIsInterruptPending

`boolean timerCaptureIsInterruptPending(const Timer* timer)`

Test if a capture interrupt is pending for the given timer.

It only makes sense to call this when interrupts are disabled or in an interrupt service routine as, if we are in the foreground, then the interrupt will trigger and we will never be able to test for it. This routine will return TRUE if a capture interrupt is pending and will be triggered when interrupts are re-enabled.

---

### timerGetCapturePin

`const IOPin* timerGetCapturePin(const Timer* timer)`

Returns the IOPin used for input capture.

If the timer does not support input capture mode, or the IOPin is not available on your system, then this will return null.

---

### timerGetCompare

`const TimerCompare* timerGetCompare(const Timer* timer, uint8_t channel)`

Find the compare unit for a given timer.

Each timer typically has between 0 and 3 compare units.

The first parameter specifies the timer (eg TIMER0, TIMER1 etc) and the second parameter is the compare unit number from 0 up to 'timerNumberOfCompareUnits(const Timer* timer) - 1'.

---

### timerGetMode

`TIMER_MODE timerGetMode(const Timer*timer)`

Returns the current mode for a timer.

The modes are:

```
TIMER_MODE_NORMAL,
TIMER_MODE_PWM8_PHASE_CORRECT,
TIMER_MODE_PWM9_PHASE_CORRECT,
TIMER_MODE_PWM10_PHASE_CORRECT,
TIMER_MODE_CTC_OCR,
TIMER_MODE_PWM8_FAST,
TIMER_MODE_PWM9_FAST,
TIMER_MODE_PWM10_FAST,
TIMER_MODE_PWM_PHASE_FREQ_ICR,
TIMER_MODE_PWM_PHASE_FREQ_OCR,
TIMER_MODE_PWM_PHASE_CORRECT_ICR,
TIMER_MODE_PWM_PHASE_CORRECT_OCR,
TIMER_MODE_CTC_ICR,
TIMER_MODE_13_RESVD,
TIMER_MODE_PWM_FAST_ICR,
TIMER_MODE_PWM_FAST_OCR
```

### timerSetMode

`timerSetMode(const Timer* timer, TIMER_MODE mode)`

Set the operating mode for the given timer.

Only use this if you know that the timer is not currently in use ie 'timerIsInUse' returns FALSE. For a list of possible modes see timerGetMode(const Timer*timer)

### timerIsModeSupported

`boolean  timerIsModeSupported(const Timer* timer, TIMER_MODE mode)`

Test if a timer supports a given mode of operation.

Some timers support all modes whilst others only support a subset of the possible modes.

### timerGetBestPrescaler

`uint16_t timerGetBestPrescaler(const Timer* timer, uint16_t repeat_ms)`

Find the optimum prescaler value for the given timer in order to measure the given number of ms.

This function takes into account the different prescale options for the given timer and whether it is a 16 bit or 8 bit timer and will return the smallest prescale value capable of measuring the given delay taking into account the clock speed of the microprocessor.

This will return a value such as 1, 16, 64, 128 etc.

### timerSupportsPrescaler

`boolean  timerSupportsPrescaler(const Timer* timer, uint16_t prescaler)`

Does the timer support the given prescaler value?

### timerGetCounter

```
uint16_t timerGetCounter(const Timer* timer)
```

Returns the current counter value for the timer.

This always returns a 16 bit value but for an 8 bit timer the top byte will always be 0x00. This function should not be used to calculate durations as the actual timings will vary according to clock speeds, prescalers etc and there are other more friendly methods to this. It's main use is in other library functions when setting compare thresholds - see uartsw.c for an example.

### timerSetPrescaler

```
timerSetPrescaler(const Timer* timer, uint16_t prescaler)
```

Set the clock prescaler value for the given timer.

### timerGetTOP

```
uint16_t timerGetTOP(const Timer* timer)
```

Returns the current value of TOP for the given timer.

This is used by library routines such as the motor drivers in order to calculate the required compare unit value for a given duty cycle.

### timerOff

```
timerOff(const Timer* timer)
```

Tuen off the specified timer.

Use this with extreme caution! If you have used 'timerIsInUse' to locate an unused timer, and then return it to the timer pool via this call then all is ok. But don't suddenly turn off another timer at random since it may be used for motor control or for the clock.

### compareClearInterruptPending

```
compareClearInterruptPending(const TimerCompare* channel)
```

Clears the 'interrupt pending' flag for the compare unit.

This is normally only used when interrupts are switched off for example when initialising the timer.

### compareIsInterruptPending

```
boolean compareIsInterruptPending(const TimerCompare* channel)
```

Test if the compare unit has set the 'interrupt pending' flag.

This is only of use when interrupts are turned off - otherwise the interrupt happens and the flag is cleared. So if this function returns TRUE then it means that as soon as interrupts are re-enabled then the callback routine will be called.

## Gait

Adds support for using gaits in both design mode and free running mode.

## Gait/GaitDesigner.h

Allows you to connect your servos to a computer running Gait Designer (downloadable from http://webbot.org.uk).

To use the gait designer you need to define all of your servos as normal. For example on my Brat biped I have split the servos into two banks - one for the left left and another for the right leg:-

```
#define UART1_RX_BUFFER_SIZE 40
#include <sys/axon2.h>
#include <Gait/GaitDesigner.h>
#include <servos.h>
#include <rprintf.h>

SERVO servo1 = MAKE_SERVO(false,B5,1500,650);
SERVO servo2 = MAKE_SERVO(false,B6,1500,650);
SERVO servo3 = MAKE_SERVO(false,B7,1500,650);
static SERVO_LIST PROGMEM bank1_list[] = {&servo1,&servo2,&servo3};
SERVO_DRIVER bank1 = MAKE_SERVO_DRIVER(bank1_list);

SERVO servo4 = MAKE_SERVO(true,E3,1500,650);
SERVO servo5 = MAKE_SERVO(true,E4,1500,650);
SERVO servo6 = MAKE_SERVO(true,E5,1500,650);
static SERVO_LIST PROGMEM bank2_list[] = {&servo4,&servo5,&servo6};
SERVO_DRIVER bank2 = MAKE_SERVO_DRIVER(bank2_list);
```

The gait designer however expects a single list of all the servos to be controlled. This can be done by:-

```
ACTUATOR_LIST PROGMEM all[] = {&servo1.actuator,&servo2.actuator,&servo3.actuator,
   &servo4.actuator,&servo5.actuator,&servo6.actuator };
```

The order of the servos in this list should remain the same once you start using Gait Designer. However you can continue to change the servo banks or whether you are using hardware or software PWM.

The final step is to create an object that is used to talk to the Gait Designer program on the computer specifying the list of all the servos and the uart and baud rate:-

```
GAIT_DESIGNER gait = MAKE_GAIT_DESIGNER(all, UART1, (BAUD_RATE)115200);
```

In your appInitHardware you need to initialise the servos and rprintf just as normal. But you also need to initialise the gait object:-

```
void appInitHardware(void){
  servoPWMInit(&bank1);
  servoPWMInit(&bank2);
  gaitDesignerInit(&gait);
  rprintfInit(&uart1SendByte);
}
```

The final step is to regularly call the gait object from your main loop so that incoming messages get processed:-

```
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
  gaitDesignerProcess(&gait);
  return 0;
}
```

You are now ready to run the Gait Designer application to design the gait. Once you have finished the design you can export it from Gait Designer by using the File | Export menu option. This will create an H file containing all of the animations in the design.

You are now ready to use "*GaitRunner.h*" (see page ) so that the robot can play the gait without the connection to the PC.

## Standard Function Summary

| | |
|---|---|
| `void` | gaitDesignerInit(GAIT_DESIGNER* gait)<br>Initialise the gait program. |
| `void` | gaitDesignerProcess(GAIT_DESIGNER* gait)<br>Process commands from the computer. |

## Standard Function Detail

### gaitDesignerInit

`void gaitDesignerInit(GAIT_DESIGNER* gait)`

Initialise the gait program.

This will initialise the uart to the correct baud rate and start listening to commands from the Gait Designer program on the computer.

Call this once - from your appInitHardware

---

### gaitDesignerProcess

`void gaitDesignerProcess(GAIT_DESIGNER* gait)`

Process commands from the computer.

You should call this method from your main loop. It will process all new messages from the computer and move the servos accordingly.

## Gait/GaitRunner.h

Allows you to embed the gait produced by Gait Designer into your program so that it can be used without the Gait Designer application.

You will need to make the following changes to the code example shown in "*GaitDesigner.h*" (see page ) :-

#include the gait file produced by Gait Designer using the File | Export menu option.

Looking at this file you will that it defines a constant (G8_ANIM_xxxx) for each animation in the gait and these number are used when you want to play a gait. Equally it defines a constant (G8_LIMB_xxxx) for each limb/joint/servo in your model.

Replace this line of code:

```
GAIT_DESIGNER gait = MAKE_GAIT_DESIGNER(all, UART1, (BAUD_RATE)115200);
```

with the following:

```
G8_RUNNER gait = MAKE_G8_RUNNER(all,animations);
```

The last parameter is the list of animations in the H file produced from Gait Designer.

In appInitHardware replace this line of code:

```
gaitDesignerInit(&gait);
```

with the following:

```
gaitRunnerInit(&gait);
```

In appControl replace this line of code:

```
gaitDesignerProcess(&gait);
```

with the following:

```
gaitRunnerProcess(&gait);
```

Alternatively you may choose to use the scheduler to call gaitRunnerProcess at regular intervals via interrupts.

The only missing code is the use of gaitRunnerPlay (described later) to start playing a given animation.

## Standard Function Summary

| | |
|---|---|
| G8_RUNNER | MAKE_G8_RUNNER<br>Create a standalone gait runner. |
| | gaitRunnerInit(G8_RUNNER* runner)<br>Initialise the G8_RUNNER object. |
| boolean | gaitRunnerProcess(G8_RUNNER* runner)<br>Tell the servos to move to their next position. |
| | gaitRunnerPlay(G8_RUNNER* runner, uint8_t animation, int16_t loopSpeed, DRIVE_SPEED speed, int16_t repeatCount<br>Start playing a given animation. |
| boolean | gaitRunnerIsPlaying(const G8_RUNNER* runner)<br>Return TRUE if an animation is still playing or FALSE if it has stopped. |
| | gaitRunnerStop(G8_RUNNER* runner)<br>Force the current animation to stop once it has reached the last frame of the animation. |
| int16_t | gaitRunnerRepeatCount(const G8_RUNNER* runner)<br>Returns the number of loops remaining before the animation ends. |
| | gaitRunnerSetSpeed(G8_RUNNER* runner, DRIVE_SPEED speed )<br>Sets the current playback speed of the animation. |
| DRIVE_SPEED | gaitRunnerGetSpeed(const G8_RUNNER* runner )<br>Returns the current playback speed for the animation. |

## Advanced Function Summary

| | |
|---|---|
| | gaitRunnerSetDelta(G8_RUNNER* runner, uint8_t limbNumber, DRIVE_SPEED speed )<br>Allows you to modify the gait in real time. |
| DRIVE_SPEED | gaitRunnerGetDelta(const G8_RUNNER* runner, uint8_t limbNumber)<br>Returns the last value set by |

## Standard Function Detail

### MAKE_G8_RUNNER

`G8_RUNNER MAKE_G8_RUNNER`

Create a standalone gait runner.

The first parameter is the ACTUATOR_LIST containing all of the servos you want to control.

The second parameter is the list of possible animations in the gait file exported from Gait Designer. This is called 'animations' - unless you have modified the file by hand.

---

### gaitRunnerInit

`gaitRunnerInit(G8_RUNNER* runner)`

Initialise the G8_RUNNER object.

This is normally called in appInitHardware.

---

### gaitRunnerProcess

`boolean gaitRunnerProcess(G8_RUNNER* runner)`

Tell the servos to move to their next position.

You can call this from your main loop (appControl) or you may use the scheduler to call it at regular intervals in the background. Note that since servos are only sent a pulse every 20ms then there is no point in using the scheduler to call this function more frequently than every 20ms.

The function will return TRUE if an animation is still playing or FALSE if it has stopped.

---

### gaitRunnerPlay

`gaitRunnerPlay(G8_RUNNER* runner, uint8_t animation, int16_t loopSpeed, DRIVE_SPEED speed, int16_t repeatCount`

Start playing a given animation.

The 'animation' parameter selects the animation you want to play. These are defined at the top of the file created by Gait Designer and all start with G8_ANIM_.

The 'loopSpeed' sets the overall speed of the animation. It must be a value greater than 'speed' and up to 32,767.

The 'speed' parameter is the animation playback speed and can be between -127 and 127. A negative speed will playback the animation in reverse. The higher the value then the faster the animation will play back. A value of zero will cause the animation to freeze at its current position. The speed can be changed at any time.

The 'repeatCount' parameter specifies how many loops of the animation you want to play. A value of zero will cause the animation to play continuously until you tell it to stop. Note that if the 'speed' is negative then the repeat count should also be negative.

The precise relationship between 'loopSpeed' and 'speed' is that one loop of the animation will take:

((65.536 * loopSpeed) / speed) milliseconds

A loopSpeed of 32767 and a speed of 1 will mean that a single loop of the animation will take about 35.8 minutes !!

A loopSpeed of 5000 will mean you can vary the loop time from 327 seconds (speed=1) to 2.58 seconds (speed=127).

## gaitRunnerIsPlaying

```
boolean gaitRunnerIsPlaying(const G8_RUNNER* runner)
```

Return TRUE if an animation is still playing or FALSE if it has stopped.

NB if you set the animation to play at a speed of zero then the animation is still playing - but the servos will be frozen at their current position.

## gaitRunnerStop

```
gaitRunnerStop(G8_RUNNER* runner)
```

Force the current animation to stop once it has reached the last frame of the animation.

## gaitRunnerRepeatCount

```
int16_t gaitRunnerRepeatCount(const G8_RUNNER* runner)
```

Returns the number of loops remaining before the animation ends.

A return value of zero means that the animation will keep playing until you call gaitRunnerStop.

## gaitRunnerSetSpeed

```
gaitRunnerSetSpeed(G8_RUNNER* runner, DRIVE_SPEED speed )
```

Sets the current playback speed of the animation. See gaitRunnerPlay for a description

## gaitRunnerGetSpeed

```
DRIVE_SPEED gaitRunnerGetSpeed(const G8_RUNNER* runner )
```

Returns the current playback speed for the animation.

## Advanced Function Detail

### gaitRunnerSetDelta

`gaitRunnerSetDelta(G8_RUNNER* runner, uint8_t limbNumber, DRIVE_SPEED speed )`

Allows you to modify the gait in real time.

The second parameter is the limb number. These are defined in the file exported by Gait Designer and all start with G8_LIMB_.

The third parameter allows you to adjust the value for this limb. The default value is zero.

This command can be used, for example, to take the values from an accelerometer and adjust the value of, say, the ankles of our walking robot so that if it is on a slope then the ankles move to compensate.

If a gait is currently playing then this new value will be picked up automatically when you next call gaitRunnerProcess. If no gait is running and the robot is just standing still then the servos are updated immediately.

---

### gaitRunnerGetDelta

`DRIVE_SPEED gaitRunnerGetDelta(const G8_RUNNER* runner, uint8_t limbNumber)`

Returns the last value set by "*gaitRunnerSetDelta(G8_RUNNER* runner, uint8_t limbNumber, DRIVE_SPEED speed )*" (see page )

## Maths

The Maths folder contains various helper routines for more complex mathematical processing such as Vectors and Matrices.

## Maths/Vector2D.h

Support for two dimensional (2D) vectors.

A 2D vector just has an X and a Y value and no Z value - ie no 'height'. For land vehicles, or water surface vessels, this is normally enough as the height is out of your control. The height will always be 'sea level' or whatever the ground height is at X,Y. Tanks can't fly !!

A 2D vector is the distance in X and Y between two points. For example: if you plot the point (10,12) on a piece of graph paper then it is 10 units to the right of the origin and 12 units above the origin. The two points are the origin (0,0) and the point (10,12). So a Point is the 2D vector from the origin to the point - ie draw a line from the origin to the point and that is the vector.

Vectors can also be used to store a heading. In this case the first point is the current location of the vehicle and the second point is the location that the vehicle will have reached after a given time interval. For example: a vector of (1,0) indicates that the vehicle is heading to the right and its x position will increase by 1 every time interval. However a vector of (3,0) indicates that the vehicle is still heading to the right but it is now travelling 3 times faster.

So a vehicle could use two vectors: one to hold its point position from the origin, and a second to hold the current heading.

Since a vector is a line between two points then it has a finite length. Vectors can be 'normalised' which means that the direction of the line stays the same but the length is always equal to one. Consider the above examples of a vehicle heading of (1,0) and one of (3,0). They are both heading in the same direction but at different speeds. The length of the first heading is SQRT(1x1 + 0x0) = 1 and the length of the second is SQRT(3x3 + 0x0) = 3. We can normalise these headings by dividing the X,Y values by the length ie (1,0)=>(1/1,0/1) = (1,0) and the second is (3,0)=>(3/3,0/3) = (1,0). Their normalised values are the same (1,0). This means that you can store a heading and a speed where the heading is a normalised vector and the speed is a scaling factor.

How can I use this in practice?

Well lets assume you have a GPS and the previous position is (10,8) and the next position is (13,8) then the heading is the difference between the two ie new position - old position = (13,8) - (10,8) = (3,0). So the speed (length of 3,0) is 3 and the normalised heading is (1,0).

Vectors also have some other useful functions. The most obvious being that you can very easily/quickly calculate the angle between two vectors. Lets assume our robot is at RobotX,RobotY and is heading in direction (1,0) and that there is another object located at OtherX,OtherY. We can create a vector to the other object relative to the robot which will be (OtherX-RobotX, OtherY-RobotY). We now have two vectors both of which have the robot as origin: ie the robots heading and a line from the robot to other thing. We can now use the supplied function to calculate the angle between the two vectors. If this angle is 0° then the object is 'dead-ahead'; but if it is less than -90° or greater than 90° then the other object is 'behind' the robot.

Of course the other object may be another robot with its own heading and speed. By using vectors we can also find out if, and when, the two robots are going to collide if they continue at their current heading and speed. This allows your robot to start taking evasive action now.

When you get into the mind set of having multiple robots - where each broadcasts its current position, heading and speed to the other robots then it opens all sort of possibilities. For example: how about some quadro-copters that fly in a flock like birds - ie they all follow the leader but as they twist and turn to avoid obstacles then the leader changes. Or a fixed leader robot that other robots follow.

Such 'behaviour' routines are well documented and may well get into WebbotLib at some stage.

In summary: vectors are very simple to use and can achieve complex tasks

A C++ implementation is available via the "*Vector2D*" (see page 591) class.

## Standard Function Summary

| | |
|---|---|
| `VECTOR2D` | MAKE_VECTOR2D(x,y)<br>Construct a new vector with the given x,y values. |
| `double` | vector2d_GetX(const VECTOR2D* vector)<br>Returns the X value of the vector. |
| `double` | vector2d_GetY(const VECTOR2D* vector)<br>Returns the Y value of the vector. |
| | vector2d_SetX(VECTOR2D* vector, double x)<br>Set the X value of an exiting vector. |
| | vector2d_SetY(VECTOR2D* vector, double y)<br>Set the Y value of an exiting vector. |
| | vector2d_Set(VECTOR2D* vector, double x, double y)<br>Overwrite the vector with new values. |

## Standard Function Summary

| | |
|---|---|
| `double` | vector2d_Length(const VECTOR2D* vector)<br>Get the length of the vector. |
| | vector2d_Normalise(VECTOR2D* dst,const VECTOR2D* src)<br>Normalise the vector so that it has a length of 1. |
| | vector2d_Add(VECTOR2D* dst,const VECTOR2D* src)<br>Add two vectors ie dst = dst + src |
| | vector2d_Subtract(VECTOR2D* dst,const VECTOR2D* src)<br>Subtract two vectors. |
| | vector2d_Copy(VECTOR2D* dst,const VECTOR2D* src)<br>Set a vector to be a copy of another vector. |
| | vector2d_Scale(VECTOR2D* v,double scale)<br>Multiply both the X and Y values of the vector by the given scale factor. |
| `double` | vector2d_AngleRadians(const VECTOR2D* v1, const VECTOR2D* v2)<br>Return the angle, in radians, between two vectors. |

## Advanced Function Summary

| | |
|---|---|
| `double` | vector2d_LengthSquared(const VECTOR2D* vector)<br>Return the length squared of the vector. |
| `double` | vector2d_DotProduct(const VECTOR2D* v1, const VECTOR2D* v2)<br>Return the dot product of the two vectors 'v1' and 'v2'. |

## Standard Function Detail

### MAKE_VECTOR2D

`VECTOR2D MAKE_VECTOR2D(x,y)`

Construct a new vector with the given x,y values.

For example to create a vector with the values (10,5) then:-

```
VECTOR2D myVector = MAKE_VECTOR2D(10,5);
```

### vector2d_GetX

`double vector2d_GetX(const VECTOR2D* vector)`

Returns the X value of the vector.

Example:

```
VECTOR2D myVector = MAKE_VECTOR2D(10,5); // Create vector
double x = vector2d_GetX(&myVector); // Get x value ie 10 in this case
```

## vector2d_GetY

`double vector2d_GetY(const VECTOR2D* vector)`

Returns the Y value of the vector.

Example:

```
VECTOR2D myVector = MAKE_VECTOR2D(10,5); // Create vector
double y = vector2d_GetY(&myVector); // Get y value ie 5 in this case
```

## vector2d_SetX

`vector2d_SetX(VECTOR2D* vector, double x)`

Set the X value of an exiting vector.

Example:

```
VECTOR2D myVector = MAKE_VECTOR2D(10,5);
vector2d_SetX(&myVector, 12); // Change to (12,5)
```

## vector2d_SetY

`vector2d_SetY(VECTOR2D* vector, double y)`

Set the Y value of an exiting vector.

Example:

```
VECTOR2D myVector = MAKE_VECTOR2D(10,5);
vector2d_SetY(&myVector, 6); // Change to (10,6)
```

## vector2d_Set

`vector2d_Set(VECTOR2D* vector, double x, double y)`

Overwrite the vector with new values.

Example:

```
VECTOR2D myVector; // Create vector with undefined values
vector2d_Set(&myVector, 10, 5);// Vector is now (10,5)
```

## vector2d_Length

`double vector2d_Length(const VECTOR2D* vector)`

Get the length of the vector.

Mathematically this is SQRT(x*x + y*y)

## vector2d_Normalise

`vector2d_Normalise(VECTOR2D* dst,const VECTOR2D* src)`

Normalise the vector so that it has a length of 1.0

The first parameter specifies the vector to store the result, and the second parameter specifies the vector to be normalised. Both parameters can refer to the same vector if you are happy to loose the details of the original vector.

## vector2d_Add

`vector2d_Add(VECTOR2D* dst,const VECTOR2D* src)`

Add two vectors ie dst = dst + src

## vector2d_Subtract

`vector2d_Subtract(VECTOR2D* dst,const VECTOR2D* src)`

Subtract two vectors. dst = dst - src;

## vector2d_Copy

`vector2d_Copy(VECTOR2D* dst,const VECTOR2D* src)`

Set a vector to be a copy of another vector. dst = src.

Example:

```
VECTOR2D vec1 = MAKE_VECTOR2D(10,5); // Init vec1
VECTOR2D vec2; // Create vec2 with unkown values
vector2d_Copy(&vec2, &vec1); // Set vec2 to vec1 ie (10,5)
```

## vector2d_Scale

`vector2d_Scale(VECTOR2D* v,double scale)`

Multiply both the X and Y values of the vector by the given scale factor.

Example:

Assuming we know that

```
VECTOR2D robotPosition = MAKE_VECTOR2D(10,5); // current location
VECTOR2D robotHeading = MAKE_VECTOR2D(1,0); // The normalised heading
double speed = 7; // The speed of the robot in, say, cm/second
```

We can estimate what the robot position will be in 5 seconds using:

```
VECTOR2D newPosition; // Create a new vector
VECTOR2D movement; // Create a new vector
vector2d_Copy(&newPosition, &robotPosition); // new = current
vector2d_Copy(&movement, &robotHeading); // Get heading
vector2d_Scale(&movement, speed * 5); // Heading = Heading * speed * 5
seconds
vector2d_Add(&newPosition, &movement); // Estimated position after 5 seconds
```

### vector2d_AngleRadians

`double vector2d_AngleRadians(const VECTOR2D* v1, const VECTOR2D* v2)`

Return the angle, in radians, between two vectors.

The returned value will be in the range 0 to PI.

To convert the radians value into degrees then multiply it by (180.0 / PI)

## Advanced Function Detail

### vector2d_LengthSquared

`double vector2d_LengthSquared(const VECTOR2D* vector)`

Return the length squared of the vector.

Mathematically this is: x*x + y*y

Sometimes you may only be comparing relative lengths of different vectors - in which case you can use this function instead of the real length and thus avoid the overhead of the additional square root function to convert to the real length.

### vector2d_DotProduct

`double vector2d_DotProduct(const VECTOR2D* v1, const VECTOR2D* v2)`

Return the dot product of the two vectors 'v1' and 'v2'.

The dot product is: Length(v1) * Length(v2) * cos(angle between them)

If 'v1' and 'v2' have been normalised then their lengths are 1.0 and so the above simplifies to: cos(angle between them)

## Maths/Vector3D.h

Support for three dimensional (3D) vectors.

For an elementary explanation of vectors see Vector2D.h only it is extended into three dimensions rather than two - so ideal for an airborne vehicle.

A C++ implementation is available via the "*Vector3D*" (see page 596) class.

| Standard Function Summary | |
|---|---|
| `VECTOR3D` | MAKE_VECTOR3D(x,y,z)<br>Construct a new vector with the given x,y,z values. |
| `double` | vector3d_GetX(const VECTOR3D* vector)<br>Returns the X value of the vector. |
| `double` | vector3d_GetY(const VECTOR3D* vector)<br>Returns the Y value of the vector. |
| `double` | vector3d_GetZ(const VECTOR3D* vector)<br>Returns the Z value of the vector. |
| | vector3d_SetX(VECTOR3D* vector, double x)<br>Set the X value of an exiting vector. |
| | vector3d_SetY(VECTOR3D* vector, double y)<br>Set the Y value of an exiting vector. |
| | vector3d_SetZ(VECTOR3D* vector, double y)<br>Set the Z value of an exiting vector. |
| | vector3d_Set(VECTOR3D* vector, double x, double y, double z)<br>Overwrite the vector with new values. |
| `double` | vector3d_Length(const VECTOR3D* vector)<br>Get the length of the vector. |
| | vector3d_Normalise(VECTOR3D* dst,const VECTOR3D* src)<br>Normalise the vector so that it has a length of 1. |
| | vector3d_Add(VECTOR3D* dst, const VECTOR3D* src)<br>Add two vectors ie dst = dst + src |
| | vector3d_Subtract(VECTOR3D* dst, const VECTOR3D* src)<br>Subtract two vectors. |

| Standard Function Summary | |
|---|---|
| | vector3d_Copy(VECTOR3D* dst, const VECTOR3D* src)<br>Set a vector to be a copy of another vector. |
| | vector3d_Scale(VECTOR3D* v,double scale)<br>Multiply the X, Y and Z values of the vector by the given scale factor. |
| | vector3d_CrossProduct(VECTOR3D*result, const VECTOR3D* v1, const VECTOR3D* v2)<br>Calculate the vector cross product. |
| `double` | vector3d_AngleRadians(const VECTOR3D* v1, const VECTOR3D* v2)<br>Return the angle, in radians, between two vectors. |

| Advanced Function Summary | |
|---|---|
| `double` | vector3d_LengthSquared(const VECTOR3D* vector)<br>Return the length squared of the vector. |
| `double` | vector3d_DotProduct(const VECTOR3D* v1, const VECTOR3D* v2)<br>Return the dot product of the two vectors 'v1' and 'v2'. |

## Standard Function Detail

### MAKE_VECTOR3D

`VECTOR3D MAKE_VECTOR3D(x,y,z)`

Construct a new vector with the given x,y,z values.

For example to create a vector with the values (10,5,7) then:-

```
VECTOR3D myVector = MAKE_VECTOR3D(10,5,7);
```

---

### vector3d_GetX

`double vector3d_GetX(const VECTOR3D* vector)`

Returns the X value of the vector.

Example:

```
VECTOR3D myVector = MAKE_VECTOR3D(10,5,7); // Create vector
double x = vector3d_GetX(&myVector); // Get x value ie 10 in this case
```

---

## vector3d_GetY

`double vector3d_GetY(const VECTOR3D* vector)`

Returns the Y value of the vector.

Example:

```
VECTOR3D myVector = MAKE_VECTOR3D(10,5,7); // Create vector
double y = vector3d_GetY(&myVector); // Get y value ie 5 in this case
```

## vector3d_GetZ

`double vector3d_GetZ(const VECTOR3D* vector)`

Returns the Z value of the vector.

Example:

```
VECTOR3D myVector = MAKE_VECTOR3D(10,5,7); // Create vector
double y = vector3d_GetZ(&myVector); // Get z value ie 7 in this case
```

## vector3d_SetX

`vector3d_SetX(VECTOR3D* vector, double x)`

Set the X value of an exiting vector.

Example:

```
VECTOR3D myVector = MAKE_VECTOR3D(10,5,7);
vector3d_SetX(&myVector, 12); // Change to (12,5,7)
```

## vector3d_SetY

`vector3d_SetY(VECTOR3D* vector, double y)`

Set the Y value of an exiting vector.

Example:

```
VECTOR3D myVector = MAKE_VECTOR3D(10,5,7);
vector3d_SetY(&myVector, 6); // Change to (10,6,7)
```

## vector3d_SetZ

`vector3d_SetZ(VECTOR3D* vector, double y)`

Set the Z value of an exiting vector.

Example:

```
VECTOR3D myVector = MAKE_VECTOR3D(10,5,7);
vector3d_SetZ(&myVector, 6); // Change to (10,5,6)
```

## vector3d_Set

`vector3d_Set(VECTOR3D* vector, double x, double y, double z)`

Overwrite the vector with new values.

Example:

```
VECTOR3D myVector; // Create vector with undefined values
vector3d_Set(&myVector, 10, 5, 7);// Vector is now (10,5,7)
```

## vector3d_Length

`double vector3d_Length(const VECTOR3D* vector)`

Get the length of the vector.

Mathematically this is SQRT(x*x + y*y + z*z)

## vector3d_Normalise

`vector3d_Normalise(VECTOR3D* dst,const VECTOR3D* src)`

Normalise the vector so that it has a length of 1.0

The first parameter specifies the vector to store the result, and the second parameter specifies the vector to be normalised. Both parameters can refer to the same vector if you are happy to loose the details of the original vector.

## vector3d_Add

`vector3d_Add(VECTOR3D* dst, const VECTOR3D* src)`

Add two vectors ie dst = dst + src

## vector3d_Subtract

`vector3d_Subtract(VECTOR3D* dst, const VECTOR3D* src)`

Subtract two vectors. dst = dst - src;

## vector3d_Copy

`vector3d_Copy(VECTOR3D* dst, const VECTOR3D* src)`

Set a vector to be a copy of another vector. dst = src.

Example:

```
VECTOR3D vec1 = MAKE_VECTOR2D(10,5,7); // Init vec1
VECTOR3D vec2; // Create vec2 with unknown values
vector3d_Copy(&vec2, &vec1); // Set vec2 to vec1 ie (10,5,7)
```

## vector3d_Scale

`vector3d_Scale(VECTOR3D* v,double scale)`

Multiply the X, Y and Z values of the vector by the given scale factor.

Example:

Assuming we know that

```
VECTOR3D robotPosition = MAKE_VECTOR2D(10,5,7); // current location
VECTOR3D robotHeading = MAKE_VECTOR2D(1,0,0); // The normalised heading
double speed = 7; // The speed of the robot in, say, cm/second
```

We can estimate what the robot position will be in 5 seconds using:

```
VECTOR3D newPosition; // Create a new vector
VECTOR3D movement; // Create a new vector
vector3d_Copy(&newPosition, &robotPosition); // new = current
vector3d_Copy(&movement, &robotHeading); // Get heading
vector3d_Scale(&movement, speed * 5); // Heading = Heading * speed * 5
seconds
vector3d_Add(&newPosition, &movement); // Estimated position after 5 seconds
```

## vector3d_CrossProduct

`vector3d_CrossProduct(VECTOR3D*result, const VECTOR3D* v1, const VECTOR3D* v2)`

Calculate the vector cross product.

The first parameter is the address of an existing Vector3D to store the result and the remaining parameters are the two vectors to calculate the cross product from.

The resultant vector is at 90° to the two vectors.

This is very handy in a 3 dimensional world as it allows you to determine the complete orientation of a body given only two vectors - ie it can be used to calculate the missing third vector.

## vector3d_AngleRadians

`double vector3d_AngleRadians(const VECTOR3D* v1, const VECTOR3D* v2)`

Return the angle, in radians, between two vectors.

The returned value will be in the range 0 to PI.

To convert the radians value into degrees then multiply it by (180.0 / PI)

## Advanced Function Detail

### vector3d_LengthSquared

```
double vector3d_LengthSquared(const VECTOR3D* vector)
```

Return the length squared of the vector.

Mathematically this is: x*x + y*y + z*z

Sometimes you may only be comparing relative lengths of different vectors - in which case you can use this function instead of the real length and thus avoid the overhead of the additional square root function to convert to the real length.

---

### vector3d_DotProduct

```
double vector3d_DotProduct(const VECTOR3D* v1, const VECTOR3D* v2)
```

Return the dot product of the two vectors 'v1' and 'v2'.

The dot product is: Length(v1) * Length(v2) * cos(angle between them)

If 'v1' and 'v2' have been normalised then their lengths are 1.0 and so the above simplifies to: cos(angle between them)

## Maths/Matrix3D.h

A 3D matrix allows you to transform a Vector3D ie a point in 3D space. Commonly it is used to scale, translate, or rotate the point around the X, Y, and/or Z axes.

Matrices can also be multiplied together to produce a combination of the above transforms. You could create a matrix for a rotation around the X axis, and another for rotation around the Y axis, and yet another for a translation. These matrices can be multiplied together to produce a matrix that does all three transforms in one go and this is useful if you want to transform more than one point because you then only have to apply the resultant matrix to each point.

There is a lot of info 'out there' about matrices, eg Wikipedia etc, so I'm not going to give the whole maths lesson.

A C++ implementation is available via the "*Matrix3D*" (see page 602) class.

| **Standard Function Summary** | |
|---|---|
| `MATRIX3D` | MAKE_IDENTITY_MATRIX3D<br>Create an identity matrix. |
| | matrix3d_Copy(MATRIX3D* dst, const MATRIX3D* src)<br>Copy one matrix to another ie dst = src. |
| | matrix3d_Multiply(MATRIX3D* dst, const MATRIX3D* src)<br>Multiply two matrices together. |
| | matrix3d_SetRotateX(MATRIX3D* matrix, double radians)<br>Set the matrix to be a rotation around the X axis by the specified number of radians. |
| | matrix3d_SetRotateY(MATRIX3D* matrix, double radians)<br>Set the matrix to be a rotation around the Y axis by the specified number of radians. |
| | matrix3d_SetRotateZ(MATRIX3D* matrix, double radians)<br>Set the matrix to be a rotation around the Z axis by the specified number of radians. |
| | matrix3d_SetScale(MATRIX3D* matrix, double scale)<br>Set the matrix to the given scaling factor in X, Y and Z. |
| | matrix3d_Transform(VECTOR3D* dst, const VECTOR3D* src, const MATRIX3D* matrix)<br>Transform a VECTOR3D by the given matrix. |

## Standard Function Summary

| | |
|---|---|
| | matrix3d_Add(MATRIX3D* dst, const MATRIX3D* src)<br>Add two matrices together. |
| | matrix3d_Zero(MATRIX3D* matrix)<br>Sets all elements of the matrix to 0. |
| | matrix3d_Unity(MATRIX3D* matrix)<br>Sets all elements of the matrix to 0 but the diagonals to 1 :- |
| | matrix3d_Subtract(MATRIX3D* dst, const MATRIX3D* src)<br>Subtract two matrices. |

## Advanced Function Summary

| | |
|---|---|
| `double` | matrix3d_Determinant(const MATRIX3D* matrix)<br>Returns the determinant of the matrix. |
| | matrix3d_Set(MATRIX3D* matrix,double m00,double m01,double m02,double m10,double m11,double m12,double m20,double m21,double m22<br>Initialise a matrix with the given set of values. |

## Standard Function Detail

### MAKE_IDENTITY_MATRIX3D

`MATRIX3D MAKE_IDENTITY_MATRIX3D`

Create an identity matrix.

Example:

```
MATRIX3D identity = MAKE_IDENTITY_MATRIX3D();
```

An identity matrix has the value 1 along its diagonal and 0 everywhere else. It is called the identity matrix because if you multiply it with another matrix, or with a VECTOR3D, then it makes no changes.

---

### matrix3d_Copy

`matrix3d_Copy(MATRIX3D* dst, const MATRIX3D* src)`

Copy one matrix to another ie dst = src.

For example:

```
// Create an identity matrix
MATRIX3D identity = MAKE_IDENTITY_MATRIX3D();
// Create another matrix with unknown values
MATRIX3D another;
// another = identity
matrix3d_Copy(&another, &identity);
// Both now hold the identity matrix
```

## matrix3d_Multiply

`matrix3d_Multiply(MATRIX3D* dst, const MATRIX3D* src)`

Multiply two matrices together. dst = dst * src.

## matrix3d_SetRotateX

`matrix3d_SetRotateX(MATRIX3D* matrix, double radians)`

Set the matrix to be a rotation around the X axis by the specified number of radians.

## matrix3d_SetRotateY

`matrix3d_SetRotateY(MATRIX3D* matrix, double radians)`

Set the matrix to be a rotation around the Y axis by the specified number of radians.

## matrix3d_SetRotateZ

`matrix3d_SetRotateZ(MATRIX3D* matrix, double radians)`

Set the matrix to be a rotation around the Z axis by the specified number of radians.

## matrix3d_SetScale

`matrix3d_SetScale(MATRIX3D* matrix, double scale)`

Set the matrix to the given scaling factor in X, Y and Z.

## matrix3d_Transform

`matrix3d_Transform(VECTOR3D* dst, const VECTOR3D* src, const MATRIX3D* matrix)`

Transform a VECTOR3D by the given matrix.

## matrix3d_Add

`matrix3d_Add(MATRIX3D* dst, const MATRIX3D* src)`

Add two matrices together. dst = dst + src.

### matrix3d_Zero

`matrix3d_Zero(MATRIX3D* matrix)`

Sets all elements of the matrix to 0.

### matrix3d_Unity

`matrix3d_Unity(MATRIX3D* matrix)`

Sets all elements of the matrix to 0 but the diagonals to 1 :-

```
1,0,0
0,1,0
0,0,1
```

### matrix3d_Subtract

`matrix3d_Subtract(MATRIX3D* dst, const MATRIX3D* src)`

Subtract two matrices. dst = dst - src.

## Advanced Function Detail

### matrix3d_Determinant

`double matrix3d_Determinant(const MATRIX3D* matrix)`

Returns the determinant of the matrix.

### matrix3d_Set

`matrix3d_Set(MATRIX3D* matrix,double m00,double m01,double m02,double m10,double m11,double m12,double m20,double m21,double m22`

Initialise a matrix with the given set of values.

The resultant matrix is set to:-

[ m00 m01 m02 ]

[ m10 m11 m12 ]

[ m20 m21 m22 ]

## Maths/DCM.h

Defines the C++ class for a direction cosine matrix. For more info see "*DCM*" (see page 609) .

## Class Hierarchy

Most 'useful' stuff in WebbotLib is wrapped up in a C++ hierarchy.

This section contains the list of classes and the methods you can use.

## Class: Actuator
## Defined in: actuators.h

All actuators provide the following functions:

| Function Summary | |
|---|---|
| | setSpeed<br>Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX. |
| DRIVE_SPEED | getSpeed<br>Returns the value you specified in your previous call to setSpeed. |
| | connect<br>Connect the actuator to the drive system. |
| | disconnect<br>Disconnect the actuator from the drive system. |
| | setConnected<br>Connect or disconnect the actuator from the drive system. |
| boolean | isConnected<br>Test if an actuator is connected |
| boolean | isInverted<br>Test if an actuator is inverted |

### Function Details

### setSpeed

Set the required speed to a value between DRIVE_SPEED_MIN and DRIVE_SPEED_MAX.

#### Syntax

*actuator*.setSpeed(speed)
Where *actuator* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| DRIVE_SPEED | 'speed' | The required speed (for a motor) or position (for a servo). |

#### Returns

None

### Note
If the actuator has been disconnected then nothing will happen until it is reconnected.

### See Also
getSpeed

---

## getSpeed
Returns the value you specified in your previous call to setSpeed.

### Syntax
*actuator*.getSpeed()
Where *actuator* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
DRIVE_SPEED

### Note
The returned value is not the actual speed that the actuator is moving at - we have no way of knowing that without an encoder!

### See Also
setSpeed

---

## connect
Connect the actuator to the drive system.

### Syntax
*actuator*.connect()
Where *actuator* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

### See Also
disconnect
isConnected

---

## disconnect

Disconnect the actuator from the drive system. Once an actuator is disconnected it will stop responding to other commands until it is connected once again.

### Syntax

*actuator*.disconnect()
Where *actuator* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

None

### See Also

connect
isConnected

---

## setConnected

Connect or disconnect the actuator from the drive system.

### Syntax

*actuator*.setConnected(connected)
Where *actuator* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'connected' | TRUE if connected, FALSE if disconnected. |

### Returns

None

### Note

When an actuator is disconnected it stops receiving other commands until it is reconnected.

### See Also

connect
disconnect
isConnected

---

## isConnected

Test if an actuator is connected

### Syntax
*actuator*.isConnected()
Where *actuator* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
By 'connected' we mean 'connected to the drive system' rather than 'checking that the device has been plugged in' !

## isInverted
Test if an actuator is inverted

### Syntax
*actuator*.isInverted()
Where *actuator* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
You can only change the 'inverted' setting within Project Designer. Changing the setting will cause the actuator to turn in the opposite direction.

## Class: Servo
## Defined in: servos.h

All servos provide the following functions

### Function Summary

| | |
|---|---|
| | setConfig<br>Changes the servo centre position and range at runtime. |

### Function Details

### setConfig

Changes the servo centre position and range at runtime.

Allows you to change the configuration of your servo at any time. You may want to do this, for example, because you have stored the servo settings in EEPROM and want to revert to those values rather than the values specified in Project Designer.

#### Syntax

*servo*.setConfig(centre,range)
Where *servo* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint16_t | 'centre' | The pulse length, in microseconds, that makes the servo move to the centre position. For a modified servo it will cause the servo to stop rotating. The default value is 1500 (ie 1.5ms) |
| uint16_t | 'range' | Specifies, in microseconds, the maximum pulse width change about the centre position. The default is 500. |

#### Returns

None

#### Example

Assume we have already created a servo called 'servo1' then we can modify its centre position to 1450 with a range of 475 by calling:

```
servo1.setConfig(1450, 475);
```

## Class: DynamixelAX12
## Defined in: Servos/Dynamixel/AX12.h

All Dynamixel AX12 servos support the following functions:

| Function Summary | |
|---|---|
| uint8_t | getID<br>Returns the servo ID number. |
| uint16_t | read<br>Read the current settings from a servo. |
| DRIVE_SPEED | getActualSpeed<br>Following a successful read() this will return the current position or speed of the servo. |
| int8_t | getActualRPM<br>Following a successful read() this will return the current rotational speed in RPM (revolutions per minute). |
| int16_t | getActualLoad<br>Following a successful read() this will return the current load in the range -1023 to +1023. |
| uint8_t | getActualVolts<br>Following a successful read() this will return the voltage being supplied to the servo in 10ths of a volt. |
| uint8_t | getActualTemperature<br>Following a successful read() this will return the temperature of the servo in celsius. |
| boolean | getActualIsMoving<br>Following a successful read() this will return TRUE if the servo is actually moving or FALSE if not. |
| | dump<br>Dump the current status of a given servo. |
| | ledOn<br>Turns on the LED on the servo. |
| | ledOff<br>Turns off the LED on the servo. |

## Function Details

### getID

Returns the servo ID number.

#### Syntax

*dynamixelax12*.getID()
Where *dynamixelax12* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

uint8_t

#### Note

Each AX-12 servo must have a unique ID number.

---

### read

Read the current settings from a servo.

The return value is 0 if the command executed correctly and the data in the servos 'info' structure has been updated. A non-zero value indicates an error and is made up of a number of error bits ORed together:-

```
#define AX12_RECV_LEN 0x1000     /* we received an invalid length */
#define AX12_RECV_ID 0x800       /* we received an invalid servo id */
#define AX12_RECV_HEADER 0x400   /* we received an invalid header */
#define AX12_RECV_CHECKSUM 0x200 /* we received an invalid checksum */
#define AX12_RECV_TIMEOUT 0x100  /* we timed out when receiving */
#define AX12_ERROR_INSTRUCTION 0x40 /* undefined instruction */
#define AX12_ERROR_OVERLOAD 0x20 /* the maximum torque cannot drive the load
*/
#define AX12_ERROR_CHECKSUM 0x10 /* we sent the wrong checksum to the AX12 */
#define AX12_ERROR_RANGE 0x8     /* the instruction range was invalid */
#define AX12_ERROR_OVERHEAT 0x4  /* the servo is overheating */
#define AX12_ERROR_ANGLE 0x2     /* the angle is out of range */
#define AX12_ERROR_VOLTAGE 0x1   /* the voltage is out of range */
```

If the call is successful then the info structure is updated with the current position, speed, load, voltage, temperature and whether or not the servo is moving. These settings can be accessed using the other 'getActual' functions.

#### Syntax

*dynamixelax12*.read()
Where *dynamixelax12* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
uint16_t

### Example
Assuming your servo is called 'myServo' in Project Designer:

```
// Read info from the servo
uint16_t status = myServo.read();
if( status == 0){
    // Success
}else{
    if(status & AX12_RECV_TIMEOUT){
        cerr << "Timed out reading servo\n";
    }
}
```

### See Also
getActualIsMoving
getActualLoad
getActualRPM
getActualSpeed
getActualTemperature
getActualVolts

## getActualSpeed
Following a successful read() this will return the current position or speed of the servo.

The current actual position, or speed, of the servo in the range DRIVE_SPEED_MIN to DRIVE_SPEED_MAX. You can compare this in a feedback loop, or PID controller, against the last position you requested which can be recalled by calling servo.getSpeed().

### Syntax
*dynamixelax12*.getActualSpeed()
Where *dynamixelax12* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
DRIVE_SPEED

## getActualRPM

Following a successful read() this will return the current rotational speed in RPM (revolutions per minute).

### Syntax

*dynamixelax12*.getActualRPM()
Where *dynamixelax12* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

int8_t

---

## getActualLoad

Following a successful read() this will return the current load in the range -1023 to +1023.

Not sure what 'units' this is measured in. The difference between +ve and -ve values is to do with the direction that the servo is turning.

### Syntax

*dynamixelax12*.getActualLoad()
Where *dynamixelax12* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

int16_t

---

## getActualVolts

Following a successful read() this will return the voltage being supplied to the servo in 10ths of a volt.

### Syntax

*dynamixelax12*.getActualVolts()
Where *dynamixelax12* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

uint8_t

### Example
Assuming your servo is called 'servo1' in Project Designer then:-

```
if( servo1.read() == 0 ){
    // Got info
    uint8_t v = servo1.getActualVolts();
    cout << "Volts = " << v/10 << '.' << '0'+v%10;
}
```

## getActualTemperature
Following a successful read() this will return the temperature of the servo in celsius.

### Syntax
*dynamixelax12*.getActualTemperature()
Where *dynamixelax12* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
uint8_t

### Note
If the sensor gets too hot, by driving too big a load, then it may shut down.

## getActualIsMoving
Following a successful read() this will return TRUE if the servo is actually moving or FALSE if not.

### Syntax
*dynamixelax12*.getActualIsMoving()
Where *dynamixelax12* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
If you have looked at the other information returned by read() then you may think that this call is obsolete. ie we know where we have asked the servo to move to, and what its current position and speed are. Surely - the servo will be moving if its not in the 'correct' position?

Well if the load it is trying to move is too big then it may not actually be able to move it. This function allows you to see if the servo is actually moving or not under its own power. Of course the servo may be moving for other reasons - eg it cannot support the load you have placed on it and so it is slipping.

## dump

Dump the current status of a given servo.

The output will be sent to specified output device or, if not specified, to standard out.

### Syntax

*dynamixelax12*.dump(dest)
*dynamixelax12*.dump()
Where *dynamixelax12* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| FILE * | 'dest' | Where you want to dump the output to. |

### Returns

None

### Note

This will execute servo.read() and either print out the information returned or an error message.

### See Also

dumpAll

## ledOn

Turns on the LED on the servo.

### Syntax

*dynamixelax12*.ledOn()
Where *dynamixelax12* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

None

## ledOff

Turns off the LED on the servo.

### Syntax

*dynamixelax12*.ledOff()
Where *dynamixelax12* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

None

## Class: StepperMotor
## Defined in: Stepper/_stepper_common.h

All stepper motors provide the following functions:

| Function Summary | |
|---|---|
| uint16_t | getStepsPerRevolution<br>Returns the number of steps required to rotate the motor through 360°. |
| DRIVE_SPEED | getActualSpeed<br>Returns the actual drive speed of the stepper motor. |
| uint16_t | getActualPosition<br>Returns the current position of the stepper motor between 0 and getStepsPerRevolution()-1. |
| | step<br>Tell the motor to move by the specified number of steps and then stop. |
| int16_t | getStepsRemaining<br>Returns the number of remaining steps from the last step() command. |

## Function Details

### getStepsPerRevolution

Returns the number of steps required to rotate the motor through 360°.

#### Syntax

*steppermotor*.getStepsPerRevolution()
Where *steppermotor* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

uint16_t

#### Note

This is the fixed value that you entered in Project Designer.

## getActualSpeed

Returns the actual drive speed of the stepper motor.

### Syntax

*steppermotor*.getActualSpeed()
Where *steppermotor* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

DRIVE_SPEED

## getActualPosition

Returns the current position of the stepper motor between 0 and getStepsPerRevolution()-1. eg if the stepper motor has 200 steps per revolution then it will be a number between 0 and 199.

### Syntax

*steppermotor*.getActualPosition()
Where *steppermotor* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

uint16_t

### See Also

getStepsPerRevolution

## step

Tell the motor to move by the specified number of steps and then stop.

### Syntax

*steppermotor*.step(steps)
Where *steppermotor* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| int16_t | 'steps' | The number of steps you want to move by. +ve numbers will rotate one way and -ve numbers the other. |

### Returns
None

### Note
A positive number will make the motor turn clockwise, and a negative number will make the motor turn anti-clockwise.

The motor will accelerate in the required direction and then slow down as it approaches the goal.

### See Also
getStepsRemaining

---

## getStepsRemaining
Returns the number of remaining steps from the last step() command.

### Syntax
*steppermotor*.getStepsRemaining()
Where *steppermotor* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
int16_t

### Note
This can be used to test if a step() command has finished by checking if the returned value is zero.

If the motor was already moving in one direction and you then ask it to make 10 steps in the opposite direction then this function will start off by returning a value larger that 10. This is because it must first decelerate to a stop and hence it now needs to move more than 10 steps to go back to the position you requested.

### See Also
step

---

## Class: ServoDriver
## Defined in: servos.h

| Function Summary | |
|---|---|
| | connect<br>Connect a group of servos. |
| | disconnect<br>Disconnect a group of servos. |
| | setConnected<br>Connect or disconnect a group of servos. |
| | setSpeed<br>Set the speed for a group of servos. |
| | center<br>A 'robot-side' function that talks over a UART to a terminal program that helps you to configure the centre and range of any servos you are using. |

### Function Details

### connect

Connect a group of servos.

This is the equivalent of calling connect for each servo that is attached to the driver.

#### Syntax

*servodriver*.connect()

#### Parameters

None

#### Returns

None

---

### disconnect

Disconnect a group of servos.

This is the equivalent of calling disconnect for each servo that is attached to the driver.

### Syntax
*servodriver*.disconnect()

### Parameters
None

### Returns
None

---

## setConnected
Connect or disconnect a group of servos.

This is the equivalent of calling setConnected(boolean connected) for each servo that is attached to the driver.

### Syntax
*servodriver*.setConnected(connected)

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'connected' | TRUE if connected, FALSE if disconnected. |

### Returns
None

---

## setSpeed
Set the speed for a group of servos.

This is the equivalent of calling setSpeed for each servo that is attached to the driver so that all the servos have the same speed.

### Syntax
*servodriver*.setSpeed(speed)

### Parameters

| Type | Name | Description |
|---|---|---|
| DRIVE_SPEED | 'speed' | The required speed (for a motor) or position (for a servo). |

### Returns
None

## center
A 'robot-side' function that talks over a UART to a terminal program that helps you to configure the centre and range of any servos you are using.

### Syntax
*servodriver*.center()

### Parameters
None

### Returns
None

### Note
Because this is a mini-application then it will never return. If you have multiple banks of servos then you should center each bank individually.

### Example
Create your servos as usual in Project Designer and generate the project code setting the default output device to the uart that is used to talk to your PC.

Assuming that you have created a number of servos and that you have called them 'bank1':-

```
// This is the main loop
TICK_COUNT appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    bank1.center(); // Talk to the terminal program
    return 0;
}
```

From your terminal program (eg HyperTerminal) the following commands are available:

```
L - This will list all of the servos with their current centre and range
values
N - Select the next servo
P - Select the previous servo
C - Centring mode
R - Ranging mode
+ - Add 1 to the current centre or range value
* - Add 10 to the current centre or range value
- - Subtract 1 from the current centre or range value
/ - Subtract 10 from the current centre or range value
Any other key will display the above list
```

Once you have finished setting up the servos you should issue the L command and note down the centre and range values for each servo. Go back to Project Designer and enter these values in the servo definitions.

## Class: DynamixelAX12Driver
## Defined in: Servos/Dynamixel/AX12.h

The Dynamixel AX12 servo driver that communicates with the attached AX12 servos.

### Function Summary

| | |
|---|---|
| | dumpAll<br>Dump the status of ALL of the servos. |
| | begin<br>Marks the start of a group of commands. |
| | end<br>Marks the end of a command group. |
| | send<br>Send a sequence of bytes to one, or all, AX12 servos. |

### Function Details

### dumpAll

Dump the status of ALL of the servos.

You can specify where you want the output to go, otherwise it goes to the standard output stream.

This allows you to view a snapshot of all of the servos connected to one AX12 driver.

#### Syntax

*dynamixelax12driver*.dumpAll(dest)
*dynamixelax12driver*.dumpAll()

#### Parameters

| Type | Name | Description |
|---|---|---|
| FILE * | 'dest' | Where you want to dump the output to. |

#### Returns

None

#### Example

Assuming you have called the driver 'ax12_driver' in Project Designer then to dump to standard out:

```
ax12_driver.dumpAll(stdout);
```

## begin

Marks the start of a group of commands.

These servos are often used in humanoid robots and given the number of servos involved and time taken to update them all then you may notice each servo moving one after the other.

This command allows you to mark the start of a group of commands and would normally be placed at the start of your appControl loop. The commands are sent out to the servos in the main loop but they are only acted upon when the end() command is received at the end of your main loop.

### Syntax

*dynamixelax12driver*.begin()

### Parameters

None

### Returns

None

### Example

Assuming you have called the driver 'ax12_driver' and the servos 'servo1', 'servo2' etc then you can force the servos to update their new movement simultaneously as follows:-

```
ax12_driver.begin();
servo1.setSpeed(45);
servo2.setSpeed(-45);
servo3.setSpeed(68);
servo4.setSpeed(92);
ax12_driver.end();
```

## end

Marks the end of a command group.

### Syntax

*dynamixelax12driver*.end()

### Parameters

None

### Returns
None

### See Also
begin

## send
Send a sequence of bytes to one, or all, AX12 servos.

The bytes sent to the servos will be: 0xff, 0xff, id, your data, checksum.

Note that by using ax12_BROADCAST_ID as the id the message will be broadcast to ALL servos.

### Syntax
*dynamixelax12driver*.send(id,len,data)

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'id' | The ID of the servo you want to send the command to - or ax12_BROADCAST_ID for all. |
| size_t | 'len' | The number of bytes to be sent. |
| uint8_t * | 'data' | The start address of the data bytes to be sent. |

### Returns
None

### Example
Assuming you have called your driver 'ax12_driver' in Project Designer and that you want to set the baud rate of all servos to 115200 baud.

Since all of the servos should be working at the same baud rate then we will broadcast the message to all servos.

According to the servo manual we can see that the baud rate parameter is calculated as: (2000000/baud) - 1. For 115200 baud this = (2000000 / 115200) - 1 = 16.36 which, when rounded, is 16.

```
// The data to send: write, baud, 16
uint8_t data[] = { ax12_WRITE, ax12_BAUD, 16 };
// Broadcast to all servos
ax12_driver.send( ax12_BROADCAST_ID, sizeof(data), data );
```

NB changing the baud rate is not normally recommended - the main reason being that the UART will still be communicating at the old baud rate so the servos will stop responding. Yes - you could also change the uart to the new baud rate. It just gets very confusing if you don't keep a written note as to what baud rate, and what ID number, you have allocated to each servo and there is no way to reset them back to their defaults.

To make life easier the AX12.h file defines a set of macros for setting individual values on a servo. These macros all start with 'ax12_set_'.

For example to set the position of servo1 to a value of 100 then:

```
ax12_set_GOAL_POSITION(&servo1, 100);
```

However: since the AX12 is an actuator then it is recommended that you use the common actuator commands instead:

```
servo1.setSpeed(100);
```

## Class: Stream
## Defined in: Stream/Stream.h

A Stream supports the following functions:

**Function Summary**

| | |
|---|---|
| | print<br>Prints the value of a number, or a string, to the stream. |
| | print_P<br>Prints a string from program memory to the stream. |
| | println<br>Sends a carriage return line feed sequence to the stream to indicate the end of a line of text. |
| int | read<br>Reads the next byte from the device. |
| int | write<br>Writes a single byte to the output stream. |
| size_t | write<br>Writes out a sequence of bytes from a given position in RAM. |
| size_t | write_P<br>Writes out a sequence of bytes from a given position in program memory. |

**Function Details**

### print

Prints the value of a number, or a string, to the stream.

#### Syntax

*stream*.print(c)
*stream*.print(str)
*stream*.print(n)
*stream*.print(r)
*stream*.print(n,base)
*stream*.print(r,decPlaces)
*stream*.print(stream)
*stream*.print(sensor)

### Parameters

| Type | Name | Description |
|---|---|---|
| char | 'c' | The character to be sent to the stream. |
| char * | 'str' | The string, in RAM, that you want to print to the stream. |
| integer | 'n' | The integer number you want to print to the stream. |
| real | 'r' | The real number you want to print to the stream. |
| uint8_t | 'base' | The number base you want to use for the number. ie 10=decimal, 16=hexadecimal, 2=binary, 8=octal. |
| uint8_t | 'decPlaces' | The number of decimal places to be shown to the right of the decimal point. |
| Stream | 'stream' | Reference to another stream. |
| Sensor | 'sensor' | The name of the sensor that you want to dump to the stream. |

### Returns
None

### Note
When you run the code generation in Project Designer you can define the standard output and standard error destinations. WebbotLib will automatically define the following streams to those devices:-

cout - An output stream to the 'standard output' device.

cin - An input stream from the 'standard output' device.

err - An output stream to the 'standard error' device.

### Example

```
// Print stuff to the standard output device
cout.print("Hello World\r\n");
cout.print(12);
cout.print(12.3456); // default to 2 decimal places
cout.print(12.3456,4); // uses 4 decimal places
cout.print( 'A' );
// Copy all incoming data to standard out
cout.print( cin );
// Dump the sensor called 'myCompass'
cout.print( myCompass );
```

You can also call the print routine multiple times:

```
cout.print("The answer is ").print(12);
```

Streams also support the '<<' operator:

```
cout << "The answer is " << 12;
cout << "myCompass readings: " << myCompass;
```

### See Also

print_P
println

## print_P

Prints a string from program memory to the stream.

### Syntax

*stream*.print_P(pstr)

### Parameters

| Type | Name | Description |
|---|---|---|
| char * | 'pstr' | The string, in program memory, that you want to print to the stream. |

### Returns

None

### Example

Print the string from program memory to the standard output stream called cout:

```
cout.print_P( PSTR("Hello World\n") );
```

Note the use of PSTR to place the string into program memory.

### See Also

print

## println

Sends a carriage return line feed sequence to the stream to indicate the end of a line of text.

### Syntax

*stream*.println()

### Parameters

None

### Returns

None

## read

Reads the next byte from the device. This will return EOF, the end of file marker, if there are no characters available - otherwise it will return the next byte.

### Syntax

*stream*.read()

### Parameters

None

### Returns

int

### Example

Receive the next character from standard input and echo it back to standard output:

```
int ch = cin.read();
if(ch != EOF){
    cout.write(ch);
}
```

### See Also

write

## write

Writes a single byte to the output stream. The returned value is normally the same value as the byte just written but on some devices it can return EOF to indicate a write error. For eaxmple: writing a byte out to a file stored on a µSD card may return EOF if there is no available space.

### Syntax
*stream*.write(byte)

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'byte' | The byte to be written to the output stream. |

### Returns
int

### See Also
read

## write
Writes out a sequence of bytes from a given position in RAM. The returned value is the number of bytes actually written. This would only be less than the reqested number of bytes if there is an error writing to the stream.

### Syntax
*stream*.write(bytes,len)

### Parameters

| Type | Name | Description |
|---|---|---|
| void * | 'bytes' | The start address of the bytes to be written to the stream. |
| size_t | 'len' | The number of bytes to be transferred. |

### Returns
size_t

## write_P
Writes out a sequence of bytes from a given position in program memory. The returned value is the number of bytes actually written. This would only be less than the reqested number of bytes if there is an error writing to the stream.

### Syntax
*stream*.write_P(bytes,len)

### Parameters

| Type | Name | Description |
|---|---|---|
| void * | 'bytes' | The start address of the bytes to be written to the stream. |
| size_t | 'len' | The number of bytes to be transferred. |

### Returns

size_t

## Class: Speech
## Defined in: Audio/Text2Speech/Text2Speech.h

The speech synthesiser provides the following functions:

| Function Summary | |
|---|---|
| `uint8_t` | getPitch<br>Returns the current voice pitch. |
| | setPitch<br>Set the voice pitch. |

### Function Details

### getPitch

Returns the current voice pitch.

#### Syntax

*speech*.getPitch()
Where *speech* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

uint8_t

---

### setPitch

Set the voice pitch.

#### Syntax

*speech*.setPitch(pitch)
Where *speech* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'pitch' | The new voice pitch. |

#### Returns

None

### Note

The default voice pitch value is 7. Smaller values will result in a higher pitched (faster) voice. Larger values will result in a lowr pitch (slower) voice.

## Class: Uart
## Defined in: _uart_common.h

All UARTs provide the following functions:

| Function Summary | |
|---|---|
| | on<br>Re-initialise a UART after it has been turned off. |
| | off<br>Disables the UART once all pending transmissions have finished. |
| | attach<br>Registers a callback function that is called when a character is received by the UART. |
| | detach<br>Remove any associated call back function for this UART. |
| boolean | isBusy<br>Returns FALSE if the UART has nothing left to send. |
| | flushRx<br>Flushes the receive buffer - ie discards any received characters. |
| | flushTx<br>Flushes the transmit buffer - ie it discards any outstanding bytes. |
| boolean | isRxBufferEmpty<br>Tests if the receive buffer is empty. |

## Function Details

### on

Re-initialise a UART after it has been turned off.

#### Syntax

*uart*.on(baud)

### Parameters

| Type | Name | Description |
|---|---|---|
| BAUD_RATE | 'baud' | The required baud rate. |

### Returns

None

---

## off

Disables the UART once all pending transmissions have finished.

### Syntax

*uart*.off()

### Parameters

None

### Returns

None

---

## attach

Registers a callback function that is called when a character is received by the UART.

### Syntax

*uart*.attach(callback)
*uart*.attach(callback,param)

### Parameters

| Type | Name | Description |
|---|---|---|
| void (*rx_func)(unsigned char, Uart&, void*) | 'callback' | The address of the function to be called when a character is received by the UART. |
| void * | 'param' | A user defined value that is passed to the callback routine. If not specified this will be NULL. |

### Returns

None

### Example

```
void myReceive(unsigned char data, Uart& uart, void* param){
    // You have just received a byte in the 'data' parameter
    // You can now access the uart using 'uart.functionName()'
}
```

You can now ask the UART0 to call it when a byte is received by writing:-

UART0.attach(&myReceive, NULL);

Note the '&' character at the start of your function name.

Only one callback routine can be registered at any one time.

There are times when your call back routine also needs to know the device that is using the UART. For example it could be a Drone Cell device. The final parameter of the attach allows you to specify a value that is passed into the 'param' value of your call back function.

## detach

Remove any associated call back function for this UART.

### Syntax

*uart*.detach()

### Parameters

None

### Returns

None

## isBusy

Returns FALSE if the UART has nothing left to send.

If your main loop is continually sending data over the UART, even if you have a transmit buffer, then the buffer can quickly become full since the UART is running slower than your program. Further transmissions will cause your program to halt until there is space left in the transmit buffer.

Sometimes this is not what you want to do - perhaps you want your program to go at full speed and only send data when it's not going to slow down the program.

That's when you need this function

### Syntax

*uart*.isBusy()

### Parameters

None

### Returns

boolean

### Example

```
if( ! uart0.isBusy() ){
    // UART has nothing to do
    // so send your stuff out now
    uart0 << "Hello World\n");
}
```

Obviously if the UART has no transmit buffer and you are sending more than 1 byte, or the message length is longer than your transmit buffer size, then there will still be a pause whilst sending the message.

---

## flushRx

Flushes the receive buffer - ie discards any received characters.

### Syntax

*uart*.flushRx()

### Parameters

None

### Returns

None

---

## flushTx

Flushes the transmit buffer - ie it discards any outstanding bytes.

### Syntax

*uart*.flushTx()

### Parameters

None

### Returns
None

## isRxBufferEmpty
Tests if the receive buffer is empty.

Return TRUE if it is empty or FALSE if it contains at least one byte.

### Syntax
*uart*.isRxBufferEmpty()

### Parameters
None

### Returns
boolean

### Example
```
if(myUart.isRxBufferEmpty()){
    // We have no data to process. So do other things....
}else{
    // We have data to process so start reading it in
}
```

## Class: UartHW
## Defined in: uart.h

A hardware UART provides the following functions:

**Function Summary**

| | |
|---|---|
| | setPollingMode<br>Changes a hardware UART receive mode between interrupt driven and polling modes. |

**Function Details**

### setPollingMode

Changes a hardware UART receive mode between interrupt driven and polling modes.

UARTs normally work in interrupt driven mode and potentially have a receive queue to store all the received characters. However: when using half duplex mode you only expect to receive data after you have sent something ie you 'talk' then 'listen'. When running at very high baud rates (eg 1,000,000 baud) then the processing overhead of the interrupt handling plus the queue handling can be slow enough for you to loose incoming data. Hardware interrupts from other devices only make the problem worse.

In this scenario it is better to put the receiver into 'polling' mode and then poll the UART for each incoming byte. It may even be necessary to bracket the code with a CRITICAL_SECTION so that no other interrupts cause data to be lost. But you will definitely need to make sure that you have some kind of timeout just in case you don't receive a response as this would cause everything to 'die'.

This seems like a 'dangerous' thing to do but don't forget that we are doing it because the data rate is so high. Therefore the time it takes to receive the data is actually very short.

#### Syntax

*uarthw*.setPollingMode(polling)
Where *uarthw* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'polling' | TRUE for polling mode or FALSE for interrupt mode. |

#### Returns

None

## Example

If you want to use this in your own code then here is a half duplex example for 'myUart' which expects a 10 byte response:

```
// Put uart into polling mode for receive
myUart.setPollingMode(TRUE);
```

```
// Assume we have sent out some data to start with
uint8_t reply[10]; // Put the reply here
uint8_t replyPos=0; // Nothing received yet
int now = 0; // Create a variable to test for timeouts
boolean timeOut = FALSE; // We haven't timed out
```

```
// Wait till transmit has finished
while(myUart.isBusy()){
    breathe();
}
```

```
// Turn off hardware interrupts
CRITICAL_SECTION{
    while(replyPos < sizeof(reply)){
        int ch = myUart.read();
        if(ch != -1){
            reply[replyPos++] = (uint8_t)(ch & 0xff);
            now = 0; // Reset the timeout counter
        }else if(--now == 0){
            // Timeout after a while
            timeOut = TRUE;
            break;
        }
    }
} // Turn interrupts back on
```

```
if(timeOut){
    cout << "We timed out!";
}else{
    // We have got some data
}
```

## Class: DroneCell
## Defined in: Communication/DroneCell.h

The DroneCell supports the following functions:

| Function Summary | |
|---|---|
| | powerOn<br>Starts the power up sequence for the device. |
| | powerOff<br>Places the DroneCell into a low power standby mode. |
| uint8_t | getSignalStrength<br>Get the current signal strength. |
| boolean | isRegistered<br>Test if the DroneCell is currently registered with a cell phone network operator. |
| uint8_t | messageWaiting<br>Test if a message has been received. |
| const char * | getOperator<br>Get the name of the current network operator. |
| boolean | readMessage<br>Attempt to read an incoming message from a particular SIM card slot. |
| boolean | deleteMessage<br>Delete the message in the specified SIM card slot. |
| char * | getMessagePhoneNumber<br>Having successfully read a message this will return the senders phone number. |
| char * | getMessageBody<br>Having successfully read a message this will return the body of the text message. |
| boolean | initSMS<br>Attempts to place the DroneCell into a mode where you can send an SMS text message. |
| boolean | sendSMS<br>Attempts to send the SMS text message. |

## Function Details

### powerOn

Starts the power up sequence for the device.

This is called automatically when your micro controller starts up and toggles the RESET line of the DroneCell to begin its power up sequence. Consequently~ you won't normally need to call this unless you have used the 'powerOff' command to put it in low power stand by mode and want to wake it up again.

#### Syntax

*dronecell*.powerOn()
Where *dronecell* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

#### Note

Note that this function will return immediately but it may take 5 seconds or more for the device to be up and running.

---

### powerOff

Places the DroneCell into a low power standby mode.

#### Syntax

*dronecell*.powerOff()
Where *dronecell* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

#### See Also

powerOn

---

### getSignalStrength

Get the current signal strength.

The returned value should be interpreted as follows:-

```
0 = none
1= -111dBm
2..30 = -109dBm ... -53dBm
31 = -51dBm or better
99 = not known
```

### Syntax

*dronecell*.getSignalStrength()
Where *dronecell* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

uint8_t

### Example

Assuming your device is called myDroneCell in Project Designer then:-

```
uint8_t signalStrength = myDroneCell.getSignalStrength();
```

## isRegistered

Test if the DroneCell is currently registered with a cell phone network operator.

Return FALSE if not, TRUE if yes.

### Syntax

*dronecell*.isRegistered()
Where *dronecell* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

boolean

## messageWaiting

Test if a message has been received.

This will return 0 if there is no message waiting, otherwise it is the SIM slot message index that should be passed into readMessage() to read the contents of the message.

### Syntax

*dronecell*.messageWaiting()
Where *dronecell* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

uint8_t

### Note

Having read the message you will probably need to delete it. Failure to do so will mean that the SIM card will eventually become full.

---

## getOperator

Get the name of the current network operator.

The return value is a string which may be blank if the device is still trying to find a network, otherwise it is the name of the network operator being used.

### Syntax

*dronecell*.getOperator()
Where *dronecell* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

const char *

### Example

Example (assuming you have called your device myDroneCell in Project Designer):

```
// Write name of the operator to the output
cout << "Operator=" << myDroneCell.getOperator();
```

---

## readMessage

Attempt to read an incoming message from a particular SIM card slot.

The SIM card slots are normally numbered 1 to 10 but you would normally be using the number returned from 'messageWaiting'.

The function will return FALSE if the specified slot is empty, invalid, or is not an incoming message.

If the function returns TRUE then two member string variables are set: one holds the phone number of the sender, and the other is the message body.

### Syntax

*dronecell*.readMessage(msgIndex)
Where *dronecell* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'msgIndex' | The index number of the message . |

### Returns

boolean

### Note

Once you have proessed an incoming message then you should delete it to free up space on the SIM card.

### Example

Assuming your device is called myDroneCell in Project Designer:

```
// Get next received msg no
uint8_t msgNo = myDroneCell.messageWaiting();

if(msgNo != 0){
    cout << "Message No.: " << msgNo;
    if(myDroneCell.readMessage(msgNo)){
        cout << "Msg: From " << myDroneCell.getMessagePhoneNumber();
        cout << "=" << myDroneCell.getMessageBody();
    }
}
```

### See Also

getMessagePhoneNumber
getMessageBody
deleteMessage

## deleteMessage

Delete the message in the specified SIM card slot.

This function will return FALSE if the slot is invalid or if the device is currently busy.

### Syntax

*dronecell*.deleteMessage(msgIndex)
Where *dronecell* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'msgIndex' | The index number of the message . |

### Returns

boolean

## getMessagePhoneNumber

Having successfully read a message this will return the senders phone number.

### Syntax

*dronecell*.getMessagePhoneNumber()
Where *dronecell* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

char *

### See Also

readMessage

## getMessageBody

Having successfully read a message this will return the body of the text message.

### Syntax

*dronecell*.getMessageBody()
Where *dronecell* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

char *

### See Also

readMessage

## initSMS

Attempts to place the DroneCell into a mode where you can send an SMS text message.

This function will return FALSE if the DroneCell is too busy doing something else, or TRUE if it is now ready for you to start entering the text body of your message.

When entering the text body of the message you can treat the DroneCell as an output stream and use the various print and '<<' functions to output the text.

Once your message has been created you must call sendSMS to try and send the text message.

### Syntax

*dronecell*.initSMS(phoneNumber)
Where *dronecell* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| const char * | 'phoneNumber' | The phone number you want to send the SMS message to. |

### Returns

boolean

### Note

The phone number is in the international format ie '+', country code ('1'=USA), followed by the rest of the number excluding any '0' prefix.

Note that if your code doesn't check the return value from this function then it will cause a compiler warning as it is 'unsafe' to continue trying to send a SMS text message if this function returns FALSE.

### Example

See "*droneCell*" (see page 103) for some worked examples.

### See Also

sendSMS

## sendSMS

Attempts to send the SMS text message.

This function will return FALSE if the message has not been sent. This may be because the DroneCell is not plugged in, not powered on, or the SIM card is currently full.

### Syntax

*dronecell*.sendSMS()
Where *dronecell* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

boolean

### Note

If this function returns TRUE it doesn't mean that the text message has actually been sent yet - but rather that it has been queued up on the SIM card. Consequently it is guaranteed to be sent once the DroneCell gets a signal even if the DroneCell is turned off and back on in the meantime.

### Example

See "*droneCell*" (see page 103) for some worked examples.

### See Also

initSMS

## Class: Marquee
## Defined in: segled.h

A marquee provides the following functions:

| Function Summary | |
|---|---|
| boolean | isActive<br>Does the marquee have an active scrolling message?If the marquee was created as a repeating message, and a message has been set, then this will always return TRUE unless marqueeStop(MARQUEE* marquee) is called to stop it. |
| | setCharDelay<br>Change the duration (in microseconds) for scrolling characters in the marquee. |
| | setEndDelay<br>Change the delay for an auto-repeating message or make it non-repeating by specifying a delay of 0. |

## Function Details

### isActive

Does the marquee have an active scrolling message?

If the marquee was created as a repeating message, and a message has been set, then this will always return TRUE unless marqueeStop(MARQUEE* marquee) is called to stop it.

If this is a one shot message then this will return TRUE until the whole message has been displayed when it will then display FALSE.

#### Syntax

*marquee*.isActive()
Where *marquee* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

boolean

## setCharDelay

Change the duration (in microseconds) for scrolling characters in the marquee.

### Syntax

*marquee*.setCharDelay(delay)

Where *marquee* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| TICK_COUNT | 'delay' | The delay in µS. |

### Returns

None

### Note

This may not take effect until the next message scroll.

## setEndDelay

Change the delay for an auto-repeating message or make it non-repeating by specifying a delay of 0.

### Syntax

*marquee*.setEndDelay(delay)

Where *marquee* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| TICK_COUNT | 'delay' | The delay in µS. |

### Returns

None

### Note

This may not take effect until the next message scroll.

## Class: Display
## Defined in: Displays/_display_common.h

All displays support the following functions (where possible):

| Function Summary | |
|---|---|
| DISPLAY_COLUMN | getNumColumns<br>Returns the total number of columns available. |
| DISPLAY_LINE | getNumLines<br>Returns the total number of lines available. |
| | clear<br>Clear the display and set the cursor to the home position. |
| | home<br>Move the cursor to the home position ie top left corner. |
| | setXY<br>Move the cursor to a given location. |
| | setLineWrap<br>Turn automatic line wrapping on or off. |
| | setAutoScroll<br>Turn auto-scrolling on or off. |
| | setBacklight<br>Turn the display back light on or off (assuming that it has one and that it can be controlled in software). |
| | setBrightness<br>Set the brightness of the display. |
| | setContrast<br>Set the contrast of the display to a number between 0 (least) and 100 (most). |
| | horizGraph<br>Draws a horizontal bar graph on the display. |
| | vertGraph<br>Draws a vertical bar graph on the display. |

## Function Details

### getNumColumns

Returns the total number of columns available.

#### Syntax

*display*.getNumColumns()
Where *display* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

DISPLAY_COLUMN

#### Note

This will remain constant for a given display model.

---

### getNumLines

Returns the total number of lines available.

#### Syntax

*display*.getNumLines()
Where *display* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

DISPLAY_LINE

#### Note

This will remain constant for a given display model.

---

### clear

Clear the display and set the cursor to the home position.

#### Syntax

*display*.clear()
Where *display* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

## home
Move the cursor to the home position ie top left corner.

### Syntax
*display*.home()
Where *display* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

## setXY
Move the cursor to a given location.

### Syntax
*display*.setXY(column,line)
Where *display* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| DISPLAY_COLUMN | 'column' | The column, or X co-ordinate, from 0 to max-1. |
| DISPLAY_LINE | 'line' | The line, or Y co-ordinate, from 0 to max-1. |

### Returns
None

### Note
Moving to 0,0 (the top left corner) can also be achieved by calling home().

### See Also
home

## setLineWrap

Turn automatic line wrapping on or off.

If line wrapping is turned on then if the cursor goes off the right side of the display then it will wrap around to the start of the next line.

If line wrapping is turned off then any characters beyond the end of the display are discarded.

The default setting is that line wrapping is turned off.

### Syntax

*display*.setLineWrap(linewrap)
Where *display* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'linewrap' | TRUE to enable line-wrapping or FALSE to disable. |

### Returns

None

## setAutoScroll

Turn auto-scrolling on or off.

This setting dictates what happens when the cursor moves passed the bottom of the display:

If auto-scrolling is turned on then the contents of the screen are scrolled up one line and the previous contents of the top line are lost. The cursor will remain on the bottom line which is now blank.

If auto-scrolling is turned off then the cursor will wrap around to the top of the display and the current contents of the display will remain unchanged.

The default setting is that auto-scrolling is turned off.

### Syntax

*display*.setAutoScroll(autoscroll)
Where *display* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'autoscroll' | TRUE to enable auto scrolling or FALSE to disable. |

### Returns
None

---

## setBacklight
Turn the display back light on or off (assuming that it has one and that it can be controlled in software).

The default setting is that the back light is turned off - to save battery power.

### Syntax
*display*.setBacklight(backlight)
Where *display* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'backlight' | TRUE to turn the backlight on (if there is one) or FALSE to turn it off. |

### Returns
None

---

## setBrightness
Set the brightness of the display.

If your display cannot control the brightness via software then this command is ignored.

### Syntax
*display*.setBrightness(brightness)
Where *display* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PERCENTAGE | 'brightness' | The brightness value from 0=dark to 100=bright. |

### Returns
None

---

## setContrast
Set the contrast of the display to a number between 0 (least) and 100 (most).

If your display cannot control the contrast via software then this command is ignored.

### Syntax

*display*.setContrast(contrast)
Where *display* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PERCENTAGE | 'contrast' | The contrast value from 0=low to 100=high. |

### Returns

None

## horizGraph

Draws a horizontal bar graph on the display.

### Syntax

*display*.horizGraph(graphX,graphY,graphVal,graphMax,graphChars)
Where *display* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| DISPLAY_COLUMN | 'graphX' | The X co-ordinate of the bottom left X of the graph. |
| DISPLAY_LINE | 'graphY' | The Y co-ordinate of the bottom left Y of the graph. |
| uint16_t | 'graphVal' | The value to be shown in the graph. Between 0 and graphMax. |
| uint16_t | 'graphMax' | The maximum value that will be shown in the graph. |
| uint8_t | 'graphChars' | The maximum length of the graph in display character cells. |

### Returns

None

### Note

You cannot mix horizontal graphs and vertical graphs on the display at the same time - this is due to the limit of custom characters provided by the displays.

The Sparkfun serLCD product does not allow the use of custom characters - so the graph is very low resolution.

### See Also
vertGraph

## vertGraph
Draws a vertical bar graph on the display.

### Syntax
*display*.vertGraph(graphX,graphY,graphVal,graphMax,graphChars)
Where *display* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| DISPLAY_COLUMN | 'graphX' | The X co-ordinate of the bottom left X of the graph. |
| DISPLAY_LINE | 'graphY' | The Y co-ordinate of the bottom left Y of the graph. |
| uint16_t | 'graphVal' | The value to be shown in the graph. Between 0 and graphMax. |
| uint16_t | 'graphMax' | The maximum value that will be shown in the graph. |
| uint8_t | 'graphChars' | The maximum length of the graph in display character cells. |

### Returns
None

### Note
You cannot mix horizontal graphs and vertical graphs on the display at the same time - this is due to the limit of custom characters provided by the displays.

The Sparkfun serLCD product does not allow the use of custom characters - so the graph is very low resolution.

### See Also
horizGraph

## Class: GraphicDisplay
## Defined in: Displays/_gdisplay_common.h

All graphic displays support Widgets (see "*Widget*" (see page 577) ) and the following functions (where possible):

| Function Summary | |
|---|---|
| | shutdown<br>Shuts down the display. |
| | setPaper<br>Sets the paper colour. |
| | setInk<br>Sets the ink colour. |
| | moveTo<br>Moves the graphics cursor to the absolute x,y position on the display without drawing anything. |
| | moveBy<br>Moves the graphics cursor by adding the specified x,y offsets to the current graphics cursor. |
| | plotAt<br>Moves the graphics cursor to the absolute x,y position on the display and plots that pixel in the current ink colour. |
| | plotBy<br>Plots the pixel at the current graphics cursor plus the specified x,y offsets in the current ink colour. |
| `boolean` | readAt<br>Moves the graphics cursor to the absolute x,y position on the display and reads the colour of the pixel. |
| `boolean` | readBy<br>Reads the pixel at the current graphics cursor plus the specified x,y offsets. |
| | lineTo<br>Draws a line from the graphics cursor to the specified absolute x,y position in the current ink colour. |

## Function Summary

| | |
|---|---|
| | lineBy<br>Draws a line from the current graphics cursor to the current graphics cursor plus the specified x,y offsets in the current ink colour. |
| PIXEL | getXRes<br>Returns the total number of pixels across the display. |
| PIXEL | getYRes<br>Returns the total number of pixels down the display. |
| PIXEL | getXPos<br>Returns the x position of the current graphics cursor. |
| PIXEL | getYPos<br>Returns the y position of the current graphics cursor. |
| | triangleAt<br>Draws, or fills, a triangle in the current ink colour between the three specified co-ordinates. |
| | triangleBy<br>Draws, or fills, a triangle in the current ink colour by adding the three specified co-ordinate offsets to the current graphics cursor. |
| | rectangleAt<br>Draws, or fills, a rectangle in the current ink colour using the two specified co-ordinates as the top left and bottom right corners. |
| | rectangleBy<br>Draws, or fills, a rectangle in the current ink colour by adding the two specified co-ordinate offsets to the current graphics cursor to give the top left and bottom right corners. |
| | circle<br>Draws, or fills, a circle in the current ink colour centred on the current graphics cursor and of the specified radius. |
| | replace<br>Replaces all pixels within the specified rectangle that are of one colour with a new colour. |
| | draw<br>Draws a text character, or string, in the current ink colour at |

## Function Summary

the current graphics cursor position.

## Function Details

### shutdown

Shuts down the display.

Most graphical displays don't react too kindly to you just switching them off - and can even blow the display. This command should therefore be used, where possible, as it shuts the display down properly.

Perhaps you could use a push button, which when pressed, does things like shutting down the display and turning off motors etc and then enters an infinite loop.

Once shut down the display cannot be woken up again via software until the power is turned back on again.

#### Syntax

*graphicdisplay*.shutdown()
Where *graphicdisplay* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

---

### setPaper

Sets the paper colour.

#### Syntax

*graphicdisplay*.setPaper(red,green,blue)
*graphicdisplay*.setPaper(paperColour)
*graphicdisplay*.setPaper(colour)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'red' | The amount of red - in the range 0 (min) to 255 (max). |
| uint8_t | 'green' | The amount of green - in the range 0 (min) to 255 (max). |

| Type | Name | Description |
|---|---|---|
| uint8_t | 'blue' | The amount of blue - in the range 0 (min) to 255 (max). |
| Color | 'paperColour' | The paper colour. |
| COLOR* | 'colour' | The colour. |

### Returns
None

### Note
Drawing non-transparent text will use the paper colour as the text background.

If the display is cleared then the whole display is set to the new background colour.

## setInk
Sets the ink colour.

### Syntax
*graphicdisplay*.setInk(red,green,blue)
*graphicdisplay*.setInk(inkColour)
*graphicdisplay*.setInk(colour)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'red' | The amount of red - in the range 0 (min) to 255 (max). |
| uint8_t | 'green' | The amount of green - in the range 0 (min) to 255 (max). |
| uint8_t | 'blue' | The amount of blue - in the range 0 (min) to 255 (max). |
| Color | 'inkColour' | The ink colour. |
| COLOR* | 'colour' | The colour. |

### Returns
None

## moveTo
Moves the graphics cursor to the absolute x,y position on the display without drawing anything.

### Syntax
*graphicdisplay*.moveTo(x,y)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'x' | The x value across the screen. |
| PIXEL | 'y' | The y value down the screen. |

### Returns
None

## moveBy
Moves the graphics cursor by adding the specified x,y offsets to the current graphics cursor.

### Syntax
*graphicdisplay*.moveBy(x,y)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'x' | The x value across the screen. |
| PIXEL | 'y' | The y value down the screen. |

### Returns
None

## plotAt
Moves the graphics cursor to the absolute x,y position on the display and plots that pixel in the current ink colour.

On return the graphics cursor is set to the specified x,y position.

### Syntax
*graphicdisplay*.plotAt(x,y)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'x' | The x value across the screen. |
| PIXEL | 'y' | The y value down the screen. |

### Returns
None

## plotBy
Plots the pixel at the current graphics cursor plus the specified x,y offsets in the current ink colour.

Note that the current graphics cursor is left unchanged.

### Syntax
*graphicdisplay*.plotBy(x,y)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'x' | The x value across the screen. |
| PIXEL | 'y' | The y value down the screen. |

### Returns
None

## readAt
Moves the graphics cursor to the absolute x,y position on the display and reads the colour of the pixel.

Returns FALSE if there is no response from the display.

On return the graphics cursor is set to the specified x,y position.

### Syntax
*graphicdisplay*.readAt(x,y,colour)
*graphicdisplay*.readAt(x,y,pixelColour)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'x' | The x value across the screen. |
| PIXEL | 'y' | The y value down the screen. |
| COLOR* | 'colour' | The colour. |
| Color | 'pixelColour' | The returned colour value. |

### Returns
boolean

### Note
The compiler will issue a warning if you don't test the return value.

## readBy
Reads the pixel at the current graphics cursor plus the specified x,y offsets.

Returns FALSE if there is no response from the display.

Note that the current graphics cursor is left unchanged.

### Syntax
*graphicdisplay*.readBy(x,y,colour)
*graphicdisplay*.readBy(x,y,pixelColour)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'x' | The x value across the screen. |
| PIXEL | 'y' | The y value down the screen. |
| COLOR* | 'colour' | The colour. |
| Color | 'pixelColour' | The returned colour value. |

### Returns
boolean

### Note
The compiler will issue a warning if you don't test the return value.

## lineTo

Draws a line from the graphics cursor to the specified absolute x,y position in the current ink colour.

On return the graphics cursor is set to the specified x,y position.

### Syntax

*graphicdisplay*.lineTo(x,y)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'x' | The x value across the screen. |
| PIXEL | 'y' | The y value down the screen. |

### Returns

None

## lineBy

Draws a line from the current graphics cursor to the current graphics cursor plus the specified x,y offsets in the current ink colour.

On return the graphics cursor is set to the end of the line.

### Syntax

*graphicdisplay*.lineBy(x,y)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'x' | The x value across the screen. |
| PIXEL | 'y' | The y value down the screen. |

### Returns

None

### Example

To draw a rectangle that is 10 pixels wide and 10 high using the current graphics cursor as the top left corner:

```
display.lineBy(10, 0); // Top
display.lineBy(0, 10); // Right
display.lineBy(-10, 0); // Bottom
display.lineBy(0, -10); // Left
```

## getXRes

Returns the total number of pixels across the display.

### Syntax

*graphicdisplay*.getXRes()
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

PIXEL

## getYRes

Returns the total number of pixels down the display.

### Syntax

*graphicdisplay*.getYRes()
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

PIXEL

## getXPos

Returns the x position of the current graphics cursor.

### Syntax

*graphicdisplay*.getXPos()
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns
PIXEL

---

## getYPos
Returns the y position of the current graphics cursor.

### Syntax
*graphicdisplay*.getYPos()
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
PIXEL

---

## triangleAt
Draws, or fills, a triangle in the current ink colour between the three specified co-ordinates.

On return the graphics cursor will be set to x1,y1.

### Syntax
*graphicdisplay*.triangleAt(x1,y1,x2,y2,x3,y3,fill)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters
| Type | Name | Description |
|---|---|---|
| PIXEL | 'x1' | The x value for point number 1. |
| PIXEL | 'y1' | The y value for point number 1. |
| PIXEL | 'x2' | The x value for point number 2. |
| PIXEL | 'y2' | The y value for point number 2. |
| PIXEL | 'x3' | The x value for point number 3. |
| PIXEL | 'y3' | The y value for point number 3. |
| boolean | 'fill' | FALSE to draw the outline or TRUE to fill the shape. |

#### Returns
None

#### Note
Note that drawing beyond the edges of the screen will have an 'unknown' effect.

---

### triangleBy
Draws, or fills, a triangle in the current ink colour by adding the three specified co-ordinate offsets to the current graphics cursor.

On return the graphics cursor will be left unchanged.

#### Syntax
*graphicdisplay*.triangleBy(x1,y1,x2,y2,x3,y3,fill)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'x1' | The x value for point number 1. |
| PIXEL | 'y1' | The y value for point number 1. |
| PIXEL | 'x2' | The x value for point number 2. |
| PIXEL | 'y2' | The y value for point number 2. |
| PIXEL | 'x3' | The x value for point number 3. |
| PIXEL | 'y3' | The y value for point number 3. |
| boolean | 'fill' | FALSE to draw the outline or TRUE to fill the shape. |

#### Returns
None

#### Note
Note that drawing beyond the edges of the screen will have an 'unknown' effect.

---

### rectangleAt
Draws, or fills, a rectangle in the current ink colour using the two specified co-ordinates as the top left and bottom right corners.

On return the graphics cursor will be set to x1,y1.

### Syntax

*graphicdisplay*.rectangleAt(x1,y1,x2,y2,fill)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'x1' | The x value for point number 1. |
| PIXEL | 'y1' | The y value for point number 1. |
| PIXEL | 'x2' | The x value for point number 2. |
| PIXEL | 'y2' | The y value for point number 2. |
| boolean | 'fill' | FALSE to draw the outline or TRUE to fill the shape. |

### Returns

None

### Note

Note that drawing beyond the edges of the screen will have an 'unknown' effect.

## rectangleBy

Draws, or fills, a rectangle in the current ink colour by adding the two specified co-ordinate offsets to the current graphics cursor to give the top left and bottom right corners.

On return the graphics cursor will be left unchanged.

### Syntax

*graphicdisplay*.rectangleBy(x1,y1,x2,y2,fill)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'x1' | The x value for point number 1. |
| PIXEL | 'y1' | The y value for point number 1. |
| PIXEL | 'x2' | The x value for point number 2. |
| PIXEL | 'y2' | The y value for point number 2. |

| Type | Name | Description |
|---|---|---|
| boolean | 'fill' | FALSE to draw the outline or TRUE to fill the shape. |

### Returns

None

### Note

Note that drawing beyond the edges of the screen will have an 'unknown' effect.

## circle

Draws, or fills, a circle in the current ink colour centred on the current graphics cursor and of the specified radius.

On return the graphics cursor will be left unchanged.

### Syntax

*graphicdisplay*.circle(radius,fill)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PIXEL | 'radius' | The radius of the circle. |
| boolean | 'fill' | FALSE to draw the outline or TRUE to fill the shape. |

### Returns

None

### Note

Note that drawing beyond the edges of the screen will have an 'unknown' effect.

## replace

Replaces all pixels within the specified rectangle that are of one colour with a new colour.

### Syntax

*graphicdisplay*.replace(fromColor1,toColor1,x,y,w,h)
*graphicdisplay*.replace(fromColor2,toColor2,x,y,w,h)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| COLOR* | 'fromColor1' | Replace pixels of this colour |

| Type | Name | Description |
|---|---|---|
| COLOR* | 'toColor1' | The new colour to be used |
| PIXEL | 'x' | The x value across the screen. |
| PIXEL | 'y' | The y value down the screen. |
| PIXEL | 'w' | The rectangle width. |
| PIXEL | 'h' | The rectangle height. |
| Color | 'fromColor2' | Replace pixels of this colour |
| Color | 'toColor2' | The new colour to be used |

### Returns
None

### Example
To change any red pixels to green pixels in the rectangle whose top left corner is at 10,11 and is 20 pixels wide and 15 high:-

```
COLOR red = MAKE_COLOR_RGB( 255,0,0 );
COLOR green = MAKE_COLOR_RGB( 0,255,0 );
display.replace(&red, &green, 10,11, 20,15);
```

Or, if using the C++ Color class, then:-

```
Color red( 255,0,0 );
Color green( 0,255,0 );
display.replace(red, green, 10,11, 20,15);
```

## draw
Draws a text character, or string, in the current ink colour at the current graphics cursor position.

On return the graphics cursor is moved to the right of the drawn text.

### Syntax
*graphicdisplay*.draw(c,transparent)
*graphicdisplay*.draw(str,transparent)
Where *graphicdisplay* is the name you have given to the device in Project Designer.

## Parameters

| Type | Name | Description |
|---|---|---|
| char | 'c' | The character to be sent to the stream. |
| boolean | 'transparent' | If FALSE then the background of each character is set to the current paper colour otherwise each character is plotted on top of the existing pixel colours. |
| char * | 'str' | The string, in RAM, that you want to print to the stream. |

## Returns

None

## Class: File
## Defined in: Storage/FileSystem/FAT.h

A File supports the following functions:

| Function Summary | |
|---|---|
| | close<br>Flushes any outstanding changes to the disk and then closes the file to prevent further access until it is opened once again. |
| | flush<br>Flushes any outstanding changes to the disk but leaves the file open so you can continue to access it. |
| boolean | setPos<br>Set the current position in the file for reading and writing. |
| uint32_t | getPos<br>Returns the current position in the file. |
| uint32_t | size<br>Returns the current total size of the file in bytes. |
| size_t | read<br>Reads one or more bytes from the file into memory. |
| size_t | write<br>Writes one or more bytes from memory to the file. |

## Function Details

### close

Flushes any outstanding changes to the disk and then closes the file to prevent further access until it is opened once again.

#### Syntax

*file*.close()

#### Parameters

None

#### Returns

None

### See Also

open

---

## flush

Flushes any outstanding changes to the disk but leaves the file open so you can continue to access it.

Identical to closing the file and then re-opening it but much faster.

### Syntax

*file*.flush()

### Parameters

None

### Returns

None

---

## setPos

Set the current position in the file for reading and writing.

When a file is opened the current position is set to the start of the file unless you open in append mode in which case it will be set to the end of the file.

This function allows you to change the current position.

The function will return FALSE if you have specified a position beyond the end of the file - in which case the current position will remain unchanged.

### Syntax

*file*.setPos(pos)

### Parameters

| **Type** | **Name** | **Description** |
|---|---|---|
| uint32_t | 'pos' | The position in the file. |

### Returns

boolean

---

## getPos

Returns the current position in the file.

### Syntax
*file*.getPos()

### Parameters
None

### Returns
uint32_t

## size
Returns the current total size of the file in bytes.

### Syntax
*file*.size()

### Parameters
None

### Returns
uint32_t

## read
Reads one or more bytes from the file into memory.

Returns the actual number of bytes read - which could be less than asked for if the end of file has been reached.

You can optionally change to a different position in the file by specifying a position - otherwise the read will continue on from the current position.

On completion the current position is updated for the next call.

### Syntax
*file*.read(pos,len,dest)
*file*.read(len,dest)

### Parameters
| Type | Name | Description |
|---|---|---|
| uint32_t | 'pos' | The position in the file. |

| Type | Name | Description |
|---|---|---|
| size_t | 'len' | The number of bytes to be transferred. |
| void * | 'dest' | The starting memory address to store the information read from the file. |

### Returns
size_t

### Example
Example: to open an existing file and dump its contents out using rprintf

```
File file;
char buffer[80];
if(disk_sdCard.open(file,"/TEST.TXT",'r')==0){
    size_t bytes;
    while( bytes=file.read(sizeof(buffer)-1,buffer)) > 0 ){
        // Terminate the string
        buffer[bytes]=0;
        cout << buffer;
    }
    file.close();
}
```

## write
Writes one or more bytes from memory to the file.

Returns the actual number of bytes written - which could be less than asked for if the disk has become full.

You can optionally change to a different position in the file by specifying a position - otherwise the write will continue on from the current position.

On completion the current position is updated for the next call.

### Syntax
*file*.write(pos,len,src)
*file*.write(len,src)

### Parameters

| Type | Name | Description |
|---|---|---|
| uint32_t | 'pos' | The position in the file. |
| size_t | 'len' | The number of bytes to be transferred. |
| void * | 'src' | The starting address in memory of the data to be written to the file. |

**Returns**

size_t

## Class: Sensor
## Defined in: Sensors/_sensor_commoh.h

All sensors provide the following functions

| Function Summary | |
|---|---|
| boolean | read<br>Read the sensor and store its current values. |
| | dump<br>Dump the last read sensor values to the standard output device. |
| | dumpTo<br>Dump the contents of the sensor to the specified output stream. |
| TICK_COUNT | getTimeLastRead<br>Returns the value of the clock at the time when the sensor was last read. |

### Function Details

#### read

Read the sensor and store its current values. Returns TRUE if the sensor has been read successfully, or FALSE if the sensor is busy in which case the read values are unchanged.

#### Syntax

*sensor*.read()

#### Parameters

None

#### Returns

boolean

#### Note

The read command will take a snapshot of the sensor into local variables allowing you to access them at your leisure whilst knowing that they will not change until the next read() command is made for the same sensor.

## dump

Dump the last read sensor values to the standard output device.

### Syntax

*sensor*.dump()

### Parameters

None

### Returns

None

### Note

This function is the same as writing:-

```
cout << sensor;
```

## dumpTo

Dump the contents of the sensor to the specified output stream.

### Syntax

*sensor*.dumpTo(dest)

### Parameters

| Type | Name | Description |
|---|---|---|
| FILE * | 'dest' | The destination to dump the sensor info to. This can be any stream. |

### Returns

None

### See Also

dump

## getTimeLastRead

Returns the value of the clock at the time when the sensor was last read.

### Syntax

*sensor*.getTimeLastRead()

### Parameters

None

### Returns

TICK_COUNT

### Note

This is used internally, with certain sensors, to make sure that there is a sufficient delay between performing each read.

## Class: Accelerometer
## Defined in: Sensors/Acceleration/_acceleration_common.h

All accelerometers provide the following functions:

| Function Summary | |
|---|---|
| | calibrateX<br>Manually calibrate the X axis. |
| | calibrateY<br>Manually calibrate the Y axis. |
| | calibrateZ<br>Manually calibrate the Z axis. |
| | calibrate<br>Manually calibrate the X, Y and Z axes all in one go. |
| ACCEL_TYPE | getX<br>Returns the X axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getY<br>Returns the Y axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getZ<br>Returns the Z axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |

## Function Details

### calibrateX
Manually calibrate the X axis.

#### Syntax
*accelerometer*.calibrateX(minX,maxX)
Where *accelerometer* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| ACCEL_TYPE | 'minX' | The smallest value you received for the x axis during your calibration test. |
| ACCEL_TYPE | 'maxX' | The largest value you received for the x axis during your calibration test. |

### Returns
None

### See Also
calibrate

---

## calibrateY
Manually calibrate the Y axis.

### Syntax
*accelerometer*.calibrateY(minY,maxY)
Where *accelerometer* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| ACCEL_TYPE | 'minY' | The smallest value you received for the y axis during your calibration test. |
| ACCEL_TYPE | 'maxY' | The largest value you received for the y axis during your calibration test. |

### Returns
None

### See Also
calibrate

---

## calibrateZ
Manually calibrate the Z axis.

### Syntax
*accelerometer*.calibrateZ(minY,maxY)
Where *accelerometer* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| ACCEL_TYPE | 'minY' | The smallest value you received for the y axis during your calibration test. |
| ACCEL_TYPE | 'maxY' | The largest value you received for the y axis during your calibration test. |

### Returns
None

### See Also
calibrate

---

## calibrate

Manually calibrate the X, Y and Z axes all in one go.

### Syntax

*accelerometer*.calibrate(minX,maxX,minY,maxY,minZ,maxZ)
Where *accelerometer* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| ACCEL_TYPE | 'minX' | The smallest value you received for the x axis during your calibration test. |
| ACCEL_TYPE | 'maxX' | The largest value you received for the x axis during your calibration test. |
| ACCEL_TYPE | 'minY' | The smallest value you received for the y axis during your calibration test. |
| ACCEL_TYPE | 'maxY' | The largest value you received for the y axis during your calibration test. |
| ACCEL_TYPE | 'minZ' | The smallest value you received for the z axis during your calibration test. |
| ACCEL_TYPE | 'maxZ' | The largest value you received for the z axis during your calibration test. |

### Returns

None

### Example

To calibrate your sensor:-

1. Use the example code generated by Project Designer to repeatedly dump out the x,y,z values
2. Carefully rotate the sensor around the x, y and then z axes whilst taking care not to bounce it up and down
3. Note down the minimum and maximum values that you see for each axis
4. In appInitSoftware: calibrate each axis by passing in the minimum and maximum values that you noted down. For example if your device is called 'myAccel' and the X values ranged between -935 and 1361, Y between -900 and 1300 and Z between -950 and 1200 then:-

```
myAccel.calibrate(-935,1361, -900,1300, -950,1200);
```

Now re-run your tests and you should find that it will return values between -1000 and +1000.

Alternatively you can calibrate each axis individually.

## getX

Returns the X axis acceleration in mG (1/1000ths of the gravitational constant) from the last read().

### Syntax

*accelerometer*.getX()
Where *accelerometer* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

ACCEL_TYPE

### Example

Assuming you have called the sensor 'myAccel' in Project Designer you can dump the value to the standard output device as follows:

```
myAccel.read();
ACCEL_TYPE x = myAccel.getX();
ACCEL_TYPE y = myAccel.getY();
ACCEL_TYPE z = myAccel.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

## getY

Returns the Y axis acceleration in mG (1/1000ths of the gravitational constant) from the last read().

### Syntax

*accelerometer*.getY()
Where *accelerometer* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

ACCEL_TYPE

### Example

Assuming you have called the sensor 'myAccel' in Project Designer you can dump the value to the standard output device as follows:

```
myAccel.read();
ACCEL_TYPE x = myAccel.getX();
ACCEL_TYPE y = myAccel.getY();
ACCEL_TYPE z = myAccel.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

## getZ

Returns the Z axis acceleration in mG (1/1000ths of the gravitational constant) from the last read().

### Syntax

*accelerometer*.getZ()
Where *accelerometer* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

ACCEL_TYPE

### Example

Assuming you have called the sensor 'myAccel' in Project Designer you can dump the value to the standard output device as follows:

```
myAccel.read();
ACCEL_TYPE x = myAccel.getX();
ACCEL_TYPE y = myAccel.getY();
ACCEL_TYPE z = myAccel.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

## Class: Adxl345
## Defined in: Sensors/Acceleration/AnalogDevices/ADXL345.h

The ADXL345 provides the following functions:

**Function Summary**

| | |
|---|---|
| | refresh25Hz<br>Configures the device to output new values 25 times per second. |
| | refresh50Hz<br>Configures the device to output new values 50 times per second. |
| | refresh100Hz<br>Configures the device to output new values 100 times per second. |
| | refresh200Hz<br>Configures the device to output new values 200 times per second. |
| | refresh400Hz<br>Configures the device to output new values 400 times per second. |
| | refresh800Hz<br>Configures the device to output new values 800 times per second. |
| | refresh1600Hz<br>Configures the device to output new values 1,600 times per second. |
| | refresh3200Hz<br>Configures the device to output new values 3,200 times per second. |

### Function Details

#### refresh25Hz

Configures the device to output new values 25 times per second.

**Syntax**

*adxl345*.refresh25Hz()

Where *adxl345* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

---

### refresh50Hz
Configures the device to output new values 50 times per second.

#### Syntax
*adxl345*.refresh50Hz()
Where *adxl345* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

---

### refresh100Hz
Configures the device to output new values 100 times per second.

#### Syntax
*adxl345*.refresh100Hz()
Where *adxl345* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

---

### refresh200Hz
Configures the device to output new values 200 times per second.

#### Syntax
*adxl345*.refresh200Hz()
Where *adxl345* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

---

### refresh400Hz
Configures the device to output new values 400 times per second.

#### Syntax
*adxl345*.refresh400Hz()
Where *adxl345* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

---

### refresh800Hz
Configures the device to output new values 800 times per second.

#### Syntax
*adxl345*.refresh800Hz()
Where *adxl345* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

---

### refresh1600Hz
Configures the device to output new values 1,600 times per second.

#### Syntax
*adxl345*.refresh1600Hz()
Where *adxl345* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

---

### refresh3200Hz
Configures the device to output new values 3,200 times per second.

#### Syntax
*adxl345*.refresh3200Hz()
Where *adxl345* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

---

## Class: Compass
## Defined in: Sensors/Compass/_compass_common.h

All compasses provide the following functions:

| Function Summary | |
|---|---|
| COMPASS_TYPE | getBearing<br>Returns the compass bearing, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getRoll<br>Returns the compass roll, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getPitch<br>Returns the compass pitch, in degrees, at the time of the last read(). |

## Function Details

### getBearing

Returns the compass bearing, in degrees, at the time of the last read().

#### Syntax

*compass*.getBearing()
Where *compass* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

COMPASS_TYPE

#### Note

In 3D the 'bearing' is also sometimes called 'yaw'.

#### Example

Assuming you have called the sensor 'myCompass' in Project Designer you can dump the value to the standard output device as follows:

```
myCompass.read();
COMPASS_TYPE val = myCompass.getBearing();
cout << "Bearing=" << val;
```

## getRoll

Returns the compass roll, in degrees, at the time of the last read().

### Syntax

*compass*.getRoll()
Where *compass* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

COMPASS_TYPE

### Example

Assuming you have called the sensor 'myCompass' in Project Designer you can dump the value to the standard output device as follows:

```
myCompass.read();
COMPASS_TYPE val = myCompass.getRoll();
cout << "Roll=" << val;
```

## getPitch

Returns the compass pitch, in degrees, at the time of the last read().

### Syntax

*compass*.getPitch()
Where *compass* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

COMPASS_TYPE

### Example

Assuming you have called the sensor 'myCompass' in Project Designer you can dump the value to the standard output device as follows:

```
myCompass.read();
COMPASS_TYPE val = myCompass.getPitch();
cout << "Pitch=" << val;
```

## Class: Hmc5843
## Defined in: Sensors/Compass/Honeywell/HMC5843.h

This compass provides the following additional functions:

### Function Summary

| | |
|---|---|
| int16_t | getRawX<br>Returns the raw magnetometer X value. |
| int16_t | getRawY<br>Returns the raw magnetometer Y value. |
| int16_t | getRawZ<br>Returns the raw magnetometer Z value. |
| | refresh1Hz<br>Changes the default refresh rate to once per second. |
| | refresh2Hz<br>Changes the default refresh rate to twice per second (ie every 500ms). |
| | refresh5Hz<br>Changes the default refresh rate to five times per second (ie every 200ms). |
| | refresh10Hz<br>Changes the default refresh rate to ten times per second (ie every 100ms). |
| | refresh20Hz<br>Changes the default refresh rate to twenty times per second (ie every 50ms). |
| | refresh50Hz<br>Changes the default refresh rate to fifty times per second (ie every 20ms). |
| boolean | isFunctional<br>Returns TRUE if the device is functioning correctly. |

### Function Details

#### getRawX

Returns the raw magnetometer X value.

#### Syntax
*hmc5843*.getRawX()
Where *hmc5843* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
int16_t

---

### getRawY
Returns the raw magnetometer Y value.

#### Syntax
*hmc5843*.getRawY()
Where *hmc5843* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
int16_t

---

### getRawZ
Returns the raw magnetometer Z value.

#### Syntax
*hmc5843*.getRawZ()
Where *hmc5843* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
int16_t

---

### refresh1Hz
Changes the default refresh rate to once per second.

#### Syntax
*hmc5843*.refresh1Hz()

Where *hmc5843* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

---

## refresh2Hz
Changes the default refresh rate to twice per second (ie every 500ms).

### Syntax
*hmc5843*.refresh2Hz()
Where *hmc5843* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

---

## refresh5Hz
Changes the default refresh rate to five times per second (ie every 200ms).

### Syntax
*hmc5843*.refresh5Hz()
Where *hmc5843* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

---

## refresh10Hz
Changes the default refresh rate to ten times per second (ie every 100ms).

### Syntax
*hmc5843*.refresh10Hz()
Where *hmc5843* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

---

## refresh20Hz
Changes the default refresh rate to twenty times per second (ie every 50ms).

### Syntax
*hmc5843*.refresh20Hz()
Where *hmc5843* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

---

## refresh50Hz
Changes the default refresh rate to fifty times per second (ie every 20ms).

### Syntax
*hmc5843*.refresh50Hz()
Where *hmc5843* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

---

## isFunctional
Returns TRUE if the device is functioning correctly.

Bad board layout and design can cause the magnetometer to return fixed values indefinitely.

### Syntax
*hmc5843*.isFunctional()
Where *hmc5843* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

boolean

## Class: Hmc5883L
## Defined in: Sensors/Compass/Honeywell/HMC5883L.h

This compass provides the following additional functions:

| Function Summary | |
|---|---|
| int16_t | getRawX<br>Returns the raw magnetometer X value. |
| int16_t | getRawY<br>Returns the raw magnetometer Y value. |
| int16_t | getRawZ<br>Returns the raw magnetometer Z value. |
| | refresh1_5Hz<br>Changes the default refresh rate to one and a half times per second. |
| | refresh3Hz<br>Changes the default refresh rate to three times per second. |
| | refresh7_5Hz<br>Changes the default refresh rate to seven and a half times per second. |
| | refresh15Hz<br>Changes the default refresh rate to fifteen times per second. |
| | refresh30Hz<br>Changes the default refresh rate to thirty times per second. |
| | refresh75Hz<br>Changes the default refresh rate to seventy five times per second. |
| boolean | isFunctional<br>Returns TRUE if the device is functioning correctly. |

### Function Details

### getRawX

Returns the raw magnetometer X value.

#### Syntax

*hmc5883l*.getRawX()

Where *hmc5883l* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
int16_t

---

## getRawY
Returns the raw magnetometer Y value.

### Syntax
*hmc5883l*.getRawY()
Where *hmc5883l* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
int16_t

---

## getRawZ
Returns the raw magnetometer Z value.

### Syntax
*hmc5883l*.getRawZ()
Where *hmc5883l* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
int16_t

---

## refresh1_5Hz
Changes the default refresh rate to one and a half times per second.

### Syntax
*hmc5883l*.refresh1_5Hz()
Where *hmc5883l* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

---

### refresh3Hz
Changes the default refresh rate to three times per second.

#### Syntax
*hmc5883l*.refresh3Hz()
Where *hmc5883l* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

---

### refresh7_5Hz
Changes the default refresh rate to seven and a half times per second.

#### Syntax
*hmc5883l*.refresh7_5Hz()
Where *hmc5883l* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

---

### refresh15Hz
Changes the default refresh rate to fifteen times per second.

#### Syntax
*hmc5883l*.refresh15Hz()
Where *hmc5883l* is the name you have given to the device in Project Designer.

#### Parameters
None

### Returns
None

## refresh30Hz
Changes the default refresh rate to thirty times per second.

### Syntax
*hmc5883l*.refresh30Hz()
Where *hmc5883l* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

## refresh75Hz
Changes the default refresh rate to seventy five times per second.

### Syntax
*hmc5883l*.refresh75Hz()
Where *hmc5883l* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

## isFunctional
Returns TRUE if the device is functioning correctly.

Bad board layout and design can cause the magnetometer to return fixed values indefinitely.

### Syntax
*hmc5883l*.isFunctional()
Where *hmc5883l* is the name you have given to the device in Project Designer.

### Parameters
None

**Returns**

boolean

## Class: Hmc6343
## Defined in: Sensors/Compass/Honeywell/HMC6343.h

This compass provides the following additional functions:

| Function Summary | |
|---|---|
| | refresh1Hz<br>Changes the default refresh rate to once per second. |
| | refresh5Hz<br>Changes the default refresh rate to five times per second (ie every 200ms). |
| | refresh10Hz<br>Changes the default refresh rate to ten times per second (ie every 100ms). |

### Function Details

### refresh1Hz

Changes the default refresh rate to once per second.

#### Syntax

*hmc6343*.refresh1Hz()
Where *hmc6343* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

---

### refresh5Hz

Changes the default refresh rate to five times per second (ie every 200ms).

#### Syntax

*hmc6343*.refresh5Hz()
Where *hmc6343* is the name you have given to the device in Project Designer.

#### Parameters

None

### Returns
None

## refresh10Hz
Changes the default refresh rate to ten times per second (ie every 100ms).

### Syntax
*hmc6343*.refresh10Hz()
Where *hmc6343* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

## Class: Hmc6352
## Defined in: Sensors/Compass/Honeywell/HMC6352.h

This compass provides the following additional functions:

| Function Summary | |
|---|---|
| | refresh1Hz<br>Changes the default refresh rate to once per second. |
| | refresh5Hz<br>Changes the default refresh rate to five times per second (ie every 200ms). |
| | refresh10Hz<br>Changes the default refresh rate to ten times per second (ie every 100ms). |
| | refresh20Hz<br>Changes the default refresh rate to twenty times per second (ie every 50ms). |

### Function Details

### refresh1Hz

Changes the default refresh rate to once per second.

#### Syntax

*hmc6352*.refresh1Hz()
Where *hmc6352* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

---

### refresh5Hz

Changes the default refresh rate to five times per second (ie every 200ms).

#### Syntax

*hmc6352*.refresh5Hz()
Where *hmc6352* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

---

## refresh10Hz
Changes the default refresh rate to ten times per second (ie every 100ms).

### Syntax
*hmc6352*.refresh10Hz()
Where *hmc6352* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

---

## refresh20Hz
Changes the default refresh rate to twenty times per second (ie every 50ms).

### Syntax
*hmc6352*.refresh20Hz()
Where *hmc6352* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

---

## Class: Current
## Defined in: Sensors/Current/_current_common.h

All current sensors provide the following functions:

| Function Summary | |
|---|---|
| CURRENT_TYPE | getAmps<br>Returns the number of amps going through the sensor at the time of the last read(). |

### Function Details

#### getAmps

Returns the number of amps going through the sensor at the time of the last read().

**Syntax**

*current*.getAmps()
Where *current* is the name you have given to the device in Project Designer.

**Parameters**

None

**Returns**

CURRENT_TYPE

## Class: Distance
## Defined in: Sensors/Distance/_distance_common.h

All distance sensors provide the following functions:

| Function Summary | |
|---|---|
| DISTANCE_TYPE | getDistance<br>Returns the distance in whole cm detected by the sensor during the last read(). |

### Function Details

### getDistance

Returns the distance in whole cm detected by the sensor during the last read().

If the sensor is capable of returning multiple distance values then this will return the minimum value.

#### Syntax

*distance*.getDistance()
Where *distance* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

DISTANCE_TYPE

## Class: Srf08
## Defined in: Sensors/Distance/Devantech/SRF08_Sonar.h

This sensor adds the following functions:

| Function Summary | |
|---|---|
| `uint8_t` | getLightLevel<br>Returns the light level from the SRF08 light sensor. |

### Function Details

#### getLightLevel

Returns the light level from the SRF08 light sensor.

**Syntax**

*srf08*.getLightLevel()
Where *srf08* is the name you have given to the device in Project Designer.

**Parameters**

None

**Returns**

uint8_t

## Class: Mb7077
## Defined in: Sensors/Distance/Maxbotix/MB70777.h

This sensor can be used above, or below, water and so provides the following functions:

| Function Summary | |
|---|---|
| boolean | isInWater<br>Return TRUE if you have told the sensor that it is underwater or FALSE if not. |
| | setInWater<br>Tell the sensor whether it is underwater or not. |

### Function Details

### isInWater

Return TRUE if you have told the sensor that it is underwater or FALSE if not.

#### Syntax

*mb7077*.isInWater()
Where *mb7077* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

boolean

#### Note

The sensor doesn't automatically know whether it is underwater or not - you must tell it!

---

### setInWater

Tell the sensor whether it is underwater or not. Given that is an acoustic sensor then this is necessary as sound travels at different speed through air and through water.

#### Syntax

*mb7077*.setInWater(inWater)
Where *mb7077* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'inWater' | Set to TRUE if the sensor is underwater or FALSE if not. |

### Returns

None

## Class: Sharp5_wide
## Defined in: Sensors/Distance/Sharp/GP2wide.h

This sensor adds the following functions:

| Function Summary | |
|---|---|
| uint8_t | getBeamNumber<br>Returns the current beam number that is being evaluated. |
| DISTANCE_TYPE | getBeam<br>Returns the distance, in cm, for the specified light beam. |

## Function Details

### getBeamNumber

Returns the current beam number that is being evaluated.

#### Syntax

*sharp5_wide*.getBeamNumber()
Where *sharp5_wide* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

uint8_t

---

### getBeam

Returns the distance, in cm, for the specified light beam.

#### Syntax

*sharp5_wide*.getBeam(beam)
Where *sharp5_wide* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'beam' | The individual LED number from 0 to 4. |

#### Returns

DISTANCE_TYPE

### Note

If the specified light beam is not valid then this will return 0.

The standard 'getDistance()' method will return the average distance across all beams.

## Class: Encoder
## Defined in: Sensors/Encoder/_encoder_common.h

All encoders provide the following functions:

| Function Summary | |
|---|---|
| ENCODER_TYPE | getTicks<br>Returns the number of ticks, as an ENCODER_TYPE, at the time when you performed the last read(). |
| | subtract<br>Subtract a given value from the encoder counter. |
| uint16_t | ticksPerRevolution<br>Returns the number of counter ticks generated for each 360° revolution. |
| | setInterpolating<br>Turn interpolation on or off. |
| boolean | isInterpolating<br>Returns whether or not the encoder is using interpolation. |
| TICK_COUNT | getTimeSinceLastTick<br>Returns the number of µS between your call to read() and when the last tick was received. |
| TICK_COUNT | getDurationOfLastTick<br>Returns the duration, in µS, of the last received tick. |

## Function Details

### getTicks

Returns the number of ticks, as an ENCODER_TYPE, at the time when you performed the last read(). This number will become more negative if the encoder is rotating one way, or more positive if it is rotating the other way.

#### Syntax

*encoder*.getTicks()
Where *encoder* is the name you have given to the device in Project Designer.

#### Parameters

None

### Returns
ENCODER_TYPE

### Note
If the values of the encoder are going in the wrong direction then simply tick the 'Inverted' check box in Project Designer.

If the encoder continues to run in the same direction for a period of time then the number of ticks returned by this function will eventually wrap around to 0. See subtract(ENCODER_TYPE count) for advice on how to avoid this from happening.

### See Also
read

## subtract
Subtract a given value from the encoder counter.

This allows us to stop the internal counter from overflowing if the encoder continues to rotate in the same direction. This normally means updating another variable and then subtracting a value from the encoder. See the example below.

### Syntax
*encoder*.subtract(ticks)
Where *encoder* is the name you have given to the device in Project Designer.

### Parameters
| Type | Name | Description |
|---|---|---|
| ENCODER_TYPE | 'ticks' | The number of ticks to reduce the current tick count by. |

### Returns
None

### Note
This will only effect the internal counter and so getTicks() will continue to return the original value until the next time read() is called.

### Example
Assumptions:

- our encoder is called myEncoder in Project Designer
- 1000 ticks represent a distance travelled of 1cm.

We start by having a variable called 'distanceCM', which needs to be signed as the wheel can go forward and in reverse. So let's use an 'int32_t'.

Next: every time we have got at least 1000 ticks we will update our variable with 1 more centimetre and reduce the encoder counter by 1000. This means that the remaining value in the encoder will be between -999 and +999 represents the fraction of a centimetre.

So here is the code:-

```
int32_t distanceCM; // Distance in cm - automatically initialised to 0

// This routine is called repeatedly - its your main loop
TICK_CONTROL appControl(LOOP_COUNT loopCount, TICK_COUNT loopStart){
    // Read the encoder and store the value
    myEncoder.read();

    // Get the current number of ticks
    TICK_COUNT ticks = myEncoder.getTicks();

    // Adjust for any whole cm
    if( ticks >= 1000 ){
        myEncoder.subtract(1000);
        // Update my local variable as well
        ticks -= 1000;
        distanceCM++;
    }else if( ticks <= -1000){
        myEncoder.subtract(-1000);
        // Update my local variable as weill
        ticks += 1000;
        distanceCM--;
    }
    // Print distance travelled to standard out
    double total = ticks;
    total /= 1000;
    total += (double)distanceCM;
    cout << "Distance = " << total << '\n';
}
```

## ticksPerRevolution

Returns the number of counter ticks generated for each 360° revolution.

### Syntax

*encoder*.ticksPerRevolution()

Where *encoder* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

uint16_t

## setInterpolating

Turn interpolation on or off.

### Syntax

*encoder*.setInterpolating(interpolating)
Where *encoder* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'interpolating' | TRUE to turn on interpolating or FALSE to turn it off. |

### Returns

None

### Note

The default value can be set in Project Designer. It is not generally recommended to keep turning interpolation on and off repeatedly since the interpolation logic requires a few ticks before it starts returning sensible values.

## isInterpolating

Returns whether or not the encoder is using interpolation.

### Syntax

*encoder*.isInterpolating()
Where *encoder* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

boolean

## getTimeSinceLastTick

Returns the number of µS between your call to read() and when the last tick was received.

It can be used with a low resolution encoder in conjunction with getDurationOfLastTick() to estimate the fraction of a tick.

### Syntax

*encoder*.getTimeSinceLastTick()
Where *encoder* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
TICK_COUNT

### Note
This function only returns sensible values if the encoder is using interpolation.

If the encoder has not been rotating for a long time then it may return a strange value. You can guard against this by checking the motor is currently being asked to move or whether it is at a stand still.

---

## getDurationOfLastTick
Returns the duration, in µS, of the last received tick.

Among other uses - this could be used to calculate the current RPM of a motor.

### Syntax
*encoder*.getDurationOfLastTick()
Where *encoder* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
TICK_COUNT

### Example
To calculate the current RPM of a motor then:-

```
// Read encoder
myEncoder.read();

// Can only do this if interpolation is turned on and we have a tick
TICK_COUNT usPerTick = getDurationOfLastTick();
if( myEncoder.isInterpolating() && usPerTick != 0 ){
    uint32_t uSPerMinute = 60000000; // 60 milllion
    uint32_t ticksPerMinute = usPerMinute / usPerTick;
    uint32_t revsPerMinute = ticksPerMinute / ticksPerRevolution();
    cout << "RPM:" << revsPerMinute << '\n';
}
```

The following code will guess the current fraction of a tick. This is only useful for low resolution encoders. The guess is only accurate if the current speed is fairly constant. The logic is: if the previous tick tool 100µS and happend 10µS ago then we are about 10% into the next tick.

```
TICK_COUNT guess = myEncoder.getTimeSinceLastTick();
guess *= 100; // Coz we calc a percentage
guess /= myEncoder.getDurationOfLastTick();
uint8_t percent = (guess > 100) ? 100 : guess;
```

This will return the fraction as a percentage from 0 to 100%.

## Class: Gps
## Defined in: Sensors/GPS/_gps_common.h

All GPS devices provide the following functions:

| Function Summary | |
|---|---|
| boolean | isValid<br>Returns whether or not the GPS has got a valid fix position following a read(). |
| double | getFixTime<br>Returns the current fix time from the last read() in the format HHMMSS. |
| double | getLongitudeDegrees<br>Returns the current longitude in degrees from the last read(). |
| double | getLongitudeRadians<br>Returns the current longitude in radians from the last read(). |
| double | getLatitudeDegrees<br>Returns the current latitude in degrees from the last read(). |
| double | getLatitudeRadians<br>Returns the current latitude in radians from the last read(). |
| double | getTrackDegrees<br>Returns the current track in degrees from the last read(). |
| double | getTrackRadians<br>Returns the current track in radians from the last read(). |
| double | getSpeedKnots<br>Returns the current speed in knots from the last read(). |
| double | getSpeedCMperSecond<br>Returns the current speed in cm per second from the last read(). |
| double | getSpeedMPH<br>Returns the current speed in miles per hour from the last read(). |
| int | getNumSatellites<br>Returns the current number of satellites used to create the fix from the last read(). |

## Function Summary

| | |
|---|---|
| double | getAltitudeMeter<br>Returns the current altitude above sea level in metres. |
| boolean | isUpdated<br>Indicates whether the GPS has processed any new messages during the last read(). |
| boolean | isFixTimeUpdated<br>Indicates whether the GPS has processed any new messages containing a fix time during the last read(). |
| boolean | isLongitudeUpdated<br>Indicates whether the GPS has processed any new messages containing a longitude during the last read(). |
| boolean | isLatitudeUpdated<br>Indicates whether the GPS has processed any new messages containing a latitude during the last read(). |
| boolean | isPositionUpdated<br>Indicates whether the GPS has processed any new messages containing a latitude and/or a latitude during the last read(). |
| boolean | isSpeedUpdated<br>Indicates whether the GPS has processed any new messages containing a speed during the last read(). |
| boolean | isSatellitesUpdated<br>Indicates whether the GPS has processed any new messages containing the number of satellites during the last read(). |
| boolean | isAltitudeUpdated<br>Indicates whether the GPS has processed any new messages containing the altitude during the last read(). |
| boolean | isTrackUpdated<br>Indicates whether the GPS has processed any new messages containing the track during the last read(). |

## Function Details

### isValid

Returns whether or not the GPS has got a valid fix position following a read().

### Syntax
*gps*.isValid()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
Even though the GPS may not yet have a positional fix then it may still be returning fix times.

This can be useful to test if the GPS is attached and working (ie sending messages)

### See Also
read

---

## getFixTime
Returns the current fix time from the last read() in the format HHMMSS.

### Syntax
*gps*.getFixTime()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
double

### Note
The returned value is in Universal Time - ie it may not be the same time zone as where you are sitting. You may be on 'GMT + 1' - but what time zone is a satellite on when it is constantly orbiting the earth and passing over every time zone? Answer is: universal time.

---

## getLongitudeDegrees
Returns the current longitude in degrees from the last read().

Negative values are 'West' and positive values are 'East'.

### Syntax
*gps*.getLongitudeDegrees()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
double

### See Also
isValid

---

## getLongitudeRadians
Returns the current longitude in radians from the last read().

Negative values are 'West' and positive values are 'East'.

### Syntax
*gps*.getLongitudeRadians()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
double

### See Also
isValid

---

## getLatitudeDegrees
Returns the current latitude in degrees from the last read().

Negative values are 'South' and positive values are 'North'.

### Syntax
*gps*.getLatitudeDegrees()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
double

### See Also
isValid

---

## getLatitudeRadians

Returns the current latitude in radians from the last read().

Negative values are 'South' and positive values are 'North'.

### Syntax

*gps*.getLatitudeRadians()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

double

### See Also

isValid

---

## getTrackDegrees

Returns the current track in degrees from the last read().

### Syntax

*gps*.getTrackDegrees()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

double

### See Also

isValid

---

## getTrackRadians

Returns the current track in radians from the last read().

### Syntax

*gps*.getTrackRadians()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters

None

#### Returns
double

#### See Also
isValid

---

### getSpeedKnots
Returns the current speed in knots from the last read().

The GPS normally provides this information based on previous location and time readings. It won't be very accurate for small speeds.

#### Syntax
*gps*.getSpeedKnots()
Where *gps* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
double

#### See Also
isValid

---

### getSpeedCMperSecond
Returns the current speed in cm per second from the last read().

The GPS normally provides this information based on previous location and time readings. It won't be very accurate for small speeds.

#### Syntax
*gps*.getSpeedCMperSecond()
Where *gps* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
double

#### See Also
isValid

---

## getSpeedMPH

Returns the current speed in miles per hour from the last read().

The GPS normally provides this information based on previous location and time readings. It won't be very accurate for small speeds.

### Syntax

*gps*.getSpeedMPH()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

double

### See Also

isValid

---

## getNumSatellites

Returns the current number of satellites used to create the fix from the last read().

### Syntax

*gps*.getNumSatellites()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

int

### See Also

isValid

---

## getAltitudeMeter

Returns the current altitude above sea level in metres.

If the longitude and latitude are thought of as x,y co-ordinates then this function returns the z

co-ordinate.

### Syntax

*gps*.getAltitudeMeter()

Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
double

### See Also
isValid

## isUpdated
Indicates whether the GPS has processed any new messages during the last read().

This function is provided to help cut out any unrequired processing in your own project code - ie if all of the GPS data is unchanged from the last time then it may well be that your code doesn't need to take any action.

### Syntax
*gps*.isUpdated()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
A return value of TRUE just means that a message has been received - not that the values in the message have been changed from last time. For example: if your robot is standing still and a new message specifying the current longitude and latitude has been received then this will return TRUE - but the actual longitude and latitude values may still be the same since the robot isn't moving.

Hence this function can be used to indicate that we are receiving messages from the GPS.

### See Also
read

## isFixTimeUpdated
Indicates whether the GPS has processed any new messages containing a fix time during the last read().

### Syntax
*gps*.isFixTimeUpdated()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
A return value of TRUE just means that a message has been received - not that the values in the message have been changed from last time.

Hence this function can be used to indicate that we are receiving fix-time messages from the GPS.

### See Also
read

## isLongitudeUpdated
Indicates whether the GPS has processed any new messages containing a longitude during the last read().

### Syntax
*gps*.isLongitudeUpdated()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
A return value of TRUE just means that a message has been received - not that the values in the message have been changed from last time.

Hence this function can be used to indicate that we are receiving positional messages from the GPS.

### See Also
read
isPositionUpdated
getLongitudeDegrees

getLongitudeRadians

---

## isLatitudeUpdated

Indicates whether the GPS has processed any new messages containing a latitude during the last read().

### Syntax

*gps*.isLatitudeUpdated()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

boolean

### Note

A return value of TRUE just means that a message has been received - not that the values in the message have been changed from last time.

Hence this function can be used to indicate that we are receiving positional messages from the GPS.

### See Also

read
isPositionUpdated
getLatitudeDegrees
getLatitudeRadians

---

## isPositionUpdated

Indicates whether the GPS has processed any new messages containing a latitude and/or a latitude during the last read().

### Syntax

*gps*.isPositionUpdated()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

boolean

### Note
A return value of TRUE just means that a message has been received - not that the values in the message have been changed from last time.

Hence this function can be used to indicate that we are receiving positional messages from the GPS.

### See Also
read
getLatitudeDegrees
getLatitudeRadians
getLongitudeDegrees
getLongitudeRadians

## isSpeedUpdated
Indicates whether the GPS has processed any new messages containing a speed during the last read().

### Syntax
*gps*.isSpeedUpdated()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
A return value of TRUE just means that a message has been received - not that the values in the message have been changed from last time.

Hence this function can be used to indicate that we are receiving speed messages from the GPS.

### See Also
read
getSpeedCMperSecond
getSpeedKnots
getSpeedMPH

## isSatellitesUpdated
Indicates whether the GPS has processed any new messages containing the number of satellites during the last read().

### Syntax
*gps*.isSatellitesUpdated()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
A return value of TRUE just means that a message has been received - not that the values in the message have been changed from last time.

Hence this function can be used to indicate that we are receiving satellite messages from the GPS.

### See Also
read
getNumSatellites

## isAltitudeUpdated
Indicates whether the GPS has processed any new messages containing the altitude during the last read().

### Syntax
*gps*.isAltitudeUpdated()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
A return value of TRUE just means that a message has been received - not that the values in the message have been changed from last time.

Hence this function can be used to indicate that we are receiving altitude messages from the GPS.

### See Also
read
getAltitudeMeter

## isTrackUpdated

Indicates whether the GPS has processed any new messages containing the track during the last read().

### Syntax

*gps*.isTrackUpdated()
Where *gps* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

boolean

### Note

A return value of TRUE just means that a message has been received - not that the values in the message have been changed from last time.

Hence this function can be used to indicate that we are receiving track messages from the GPS.

### See Also

read
getTrackDegrees
getTrackRadians

## Class: Gyro
## Defined in: Sensors/Gyro/_gyro_common.h

All gyros provide the following functions:

| Function Summary | |
|---|---|
| GYRO_TYPE | getX<br>Returns the X axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getY<br>Returns the Y axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getZ<br>Returns the Z axis rotational velocity in degrees per second at the time of the last read(). |

### Function Details

### getX

Returns the X axis rotational velocity in degrees per second at the time of the last read().

#### Syntax

*gyro*.getX()
Where *gyro* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

GYRO_TYPE

#### Example

Assuming you have called the sensor 'myGyro' in Project Designer you can dump the value to the standard output device as follows:

```
myGyro.read();
GYRO_TYPE x = myGyro.getX();
GYRO_TYPE y = myGyro.getY();
GYRO_TYPE z = myGyro.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

### See Also
read

## getY
Returns the Y axis rotational velocity in degrees per second at the time of the last read().

### Syntax
*gyro*.getY()
Where *gyro* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
GYRO_TYPE

### Example
Assuming you have called the sensor 'myGyro' in Project Designer you can dump the value to the standard output device as follows:

```
myGyro.read();
GYRO_TYPE x = myGyro.getX();
GYRO_TYPE y = myGyro.getY();
GYRO_TYPE z = myGyro.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

### See Also
read

## getZ
Returns the Z axis rotational velocity in degrees per second at the time of the last read().

### Syntax
*gyro*.getZ()
Where *gyro* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns

GYRO_TYPE

### Example

Assuming you have called the sensor 'myGyro' in Project Designer you can dump the value to the standard output device as follows:

```
myGyro.read();
GYRO_TYPE x = myGyro.getX();
GYRO_TYPE y = myGyro.getY();
GYRO_TYPE z = myGyro.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

### See Also

read

## Class: Itg3200
## Defined in: Sensors/Gyro/InvenSense/ITG3200.h

The ITG3200 provides the following functions:

| Function Summary | |
|---|---|
| TEMPERATURE_TYPE | getCelsius<br>Returns the ambient temperature in Celsius at the time of the last read(). |
| TEMPERATURE_TYPE | getFahrenheit<br>Returns the ambient temperature in Fahrenheit at the time of the last read(). |

### Function Details

#### getCelsius

Returns the ambient temperature in Celsius at the time of the last read().

**Syntax**

*itg3200*.getCelsius()
Where *itg3200* is the name you have given to the device in Project Designer.

**Parameters**

None

**Returns**

TEMPERATURE_TYPE

---

#### getFahrenheit

Returns the ambient temperature in Fahrenheit at the time of the last read().

**Syntax**

*itg3200*.getFahrenheit()
Where *itg3200* is the name you have given to the device in Project Designer.

**Parameters**

None

**Returns**

TEMPERATURE_TYPE

## Class: Humidity
## Defined in: Sensors/Humidity/_humidity_common.h

All humidity sensors provide the following functions:

| Function Summary | |
|---|---|
| HUMIDITY_TYPE | getHumidity<br>Returns the humidity as a percentage at the time of the last read(). |

### Function Details

### getHumidity

Returns the humidity as a percentage at the time of the last read().

#### Syntax

*humidity*.getHumidity()

Where *humidity* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

HUMIDITY_TYPE

#### Example

Assuming your device is called 'myHumidity' in Project Designer then:-

```
myHumidity.read();
HUMIDITY_VALUE val = getHumidity()();
cout << Humidity: << val;
```

#### See Also

read

## Class: Imu
## Defined in: Sensors/IMU/_imu_common.h

All IMU devices support the following functions:

| Function Summary | |
|---|---|
| ACCEL_TYPE | getAccelX<br>Returns the X axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getAccelY<br>Returns the Y axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| ACCEL_TYPE | getAccelZ<br>Returns the Z axis acceleration in mG (1/1000ths of the gravitational constant) from the last read(). |
| GYRO_TYPE | getGyroX<br>Returns the X axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getGyroY<br>Returns the Y axis rotational velocity in degrees per second at the time of the last read(). |
| GYRO_TYPE | getGyroZ<br>Returns the Z axis rotational velocity in degrees per second at the time of the last read(). |
| COMPASS_TYPE | getBearing<br>Returns the compass bearing, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getRoll<br>Returns the compass roll, in degrees, at the time of the last read(). |
| COMPASS_TYPE | getPitch<br>Returns the compass pitch, in degrees, at the time of the last read(). |

## Function Details

### getAccelX

Returns the X axis acceleration in mG (1/1000ths of the gravitational constant) from the last read().

#### Syntax

*imu*.getAccelX()
Where *imu* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

ACCEL_TYPE

#### Example

Assuming you have called the sensor 'myIMU' in Project Designer you can dump the value to the standard output device as follows:

```
myIMU.read();
ACCEL_TYPE x = myIMU.getX();
ACCEL_TYPE y = myIMU.getY();
ACCEL_TYPE z = myIMU.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

---

### getAccelY

Returns the Y axis acceleration in mG (1/1000ths of the gravitational constant) from the last read().

#### Syntax

*imu*.getAccelY()
Where *imu* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

ACCEL_TYPE

#### Example

Assuming you have called the sensor 'myIMU' in Project Designer you can dump the value to the standard output device as follows:

```
myIMU.read();
ACCEL_TYPE x = myIMU.getX();
ACCEL_TYPE y = myIMU.getY();
ACCEL_TYPE z = myIMU.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

## getAccelZ

Returns the Z axis acceleration in mG (1/1000ths of the gravitational constant) from the last read().

### Syntax

*imu*.getAccelZ()
Where *imu* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

ACCEL_TYPE

### Example

Assuming you have called the sensor 'myIMU' in Project Designer you can dump the value to the standard output device as follows:

```
myIMU.read();
ACCEL_TYPE x = myIMU.getX();
ACCEL_TYPE y = myIMU.getY();
ACCEL_TYPE z = myIMU.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

## getGyroX

Returns the X axis rotational velocity in degrees per second at the time of the last read().

### Syntax

*imu*.getGyroX()
Where *imu* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

GYRO_TYPE

### Example

Assuming you have called the sensor 'myIMU' in Project Designer you can dump the value to the standard output device as follows:

```
myIMU.read();
GYRO_TYPE x = myIMU.getX();
GYRO_TYPE y = myIMU.getY();
GYRO_TYPE z = myIMU.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

### See Also

read

## getGyroY

Returns the Y axis rotational velocity in degrees per second at the time of the last read().

### Syntax

*imu*.getGyroY()
Where *imu* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

GYRO_TYPE

### Example

Assuming you have called the sensor 'myIMU' in Project Designer you can dump the value to the standard output device as follows:

```
myIMU.read();
GYRO_TYPE x = myIMU.getX();
GYRO_TYPE y = myIMU.getY();
GYRO_TYPE z = myIMU.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

### See Also

read

## getGyroZ

Returns the Z axis rotational velocity in degrees per second at the time of the last read().

### Syntax

*imu*.getGyroZ()
Where *imu* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
GYRO_TYPE

#### Example
Assuming you have called the sensor 'myIMU' in Project Designer you can dump the value to the standard output device as follows:

```
myIMU.read();
GYRO_TYPE x = myIMU.getX();
GYRO_TYPE y = myIMU.getY();
GYRO_TYPE z = myIMU.getZ();
cout << "X=" << x << " Y=" << y << " Z=" << z;
```

#### See Also
read

### getBearing
Returns the compass bearing, in degrees, at the time of the last read().

#### Syntax
*imu*.getBearing()
Where *imu* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
COMPASS_TYPE

#### Note
In 3D the 'bearing' is also sometimes called 'yaw'.

#### Example
Assuming you have called the sensor 'myIMU' in Project Designer you can dump the value to the standard output device as follows:

```
myIMU.read();
COMPASS_TYPE yaw = myIMU.getBearing();
COMPASS_TYPE roll = myIMU.getRoll();
COMPASS_TYPE pitch = myIMU.getPitch();
cout << "Yaw=" << yaw << " Roll=" << roll << " Pitch=" << pitch;
```

## getRoll

Returns the compass roll, in degrees, at the time of the last read().

### Syntax

*imu*.getRoll()

Where *imu* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

COMPASS_TYPE

### Example

Assuming you have called the sensor 'myIMU' in Project Designer you can dump the value to the standard output device as follows:

```
myIMU.read();
COMPASS_TYPE yaw = myIMU.getBearing();
COMPASS_TYPE roll = myIMU.getRoll();
COMPASS_TYPE pitch = myIMU.getPitch();
cout << "Yaw=" << yaw << " Roll=" << roll << " Pitch=" << pitch;
```

## getPitch

Returns the compass pitch, in degrees, at the time of the last read().

### Syntax

*imu*.getPitch()

Where *imu* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

COMPASS_TYPE

### Example

Assuming you have called the sensor 'myIMU' in Project Designer you can dump the value to the standard output device as follows:

```
myIMU.read();
COMPASS_TYPE yaw = myIMU.getBearing();
COMPASS_TYPE roll = myIMU.getRoll();
COMPASS_TYPE pitch = myIMU.getPitch();
cout << "Yaw=" << yaw << " Roll=" << roll << " Pitch=" << pitch;
```

## Class: Razor
## Defined in: Sensors/IMU/Sparkfun/razor.h

The Razor also provides the following functions:

| Function Summary | |
|---|---|
| | ledOn<br>Turns on the LED on the Razor board. |
| | ledOff<br>Turns off the LED on the Razor board. |
| | ledSet<br>Turn the LED on the Razor board either on or off. |

### Function Details

### ledOn

Turns on the LED on the Razor board.

#### Syntax

*razor*.ledOn()
Where *razor* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

### ledOff

Turns off the LED on the Razor board.

#### Syntax

*razor*.ledOff()
Where *razor* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

## ledSet

Turn the LED on the Razor board either on or off.

### Syntax

*razor*.ledSet(state)

Where *razor* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'state' | TRUE to turn the LED on or FALSE to turn it off. |

### Returns

None

## Class: Pressure
## Defined in: Sensors/Pressure/_pressure_common.h

All pressure sensors support the following functions:

| Function Summary | |
|---|---|
| PRESSURE_TYPE | getPa<br>Returns the last read pressure in Pa. |
| PRESSURE_TYPE | getkPa<br>Returns the last read pressure in kPa. |
| double | relativeDepth<br>Returns the under water depth in meters relative to when the power was switched on. |
| double | altitude<br>Returns the current altitude in meters above sea level. |
| double | relativeAltitude<br>Returns the current altitude in meters above, or below, the location where the power was turned on. |

### Function Details

### getPa

Returns the last read pressure in Pa.

#### Syntax

*pressure*.getPa()
Where *pressure* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

PRESSURE_TYPE

---

### getkPa

Returns the last read pressure in kPa.

#### Syntax

*pressure*.getkPa()

Where *pressure* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
PRESSURE_TYPE

---

## relativeDepth
Returns the under water depth in meters relative to when the power was switched on.

### Syntax
*pressure*.relativeDepth()
Where *pressure* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
double

### Note
You should make sure that your submersible is floating on the surface when the power is switched on.

---

## altitude
Returns the current altitude in meters above sea level.

### Syntax
*pressure*.altitude()
Where *pressure* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
double

---

## relativeAltitude
Returns the current altitude in meters above, or below, the location where the power was turned on.

## Syntax

*pressure*.relativeAltitude()

Where *pressure* is the name you have given to the device in Project Designer.

## Parameters

None

## Returns

double

## Note

Note that this value is only equivalent to 'height above ground' if your device was placed on the ground when power was applied and if the surrounding terrain is totally flat.

## Class: Bmp085
## Defined in: Sensors/Pressure/Bosch/BMP085.h

The BMP085 sensor also provides the following functions:

| Function Summary | |
|---|---|
| TEMPERATURE_TYPE | getCelsius<br>Returns the temperature in celsius. |

### Function Details

### getCelsius

Returns the temperature in celsius.

#### Syntax

*bmp085*.getCelsius()

Where *bmp085* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

TEMPERATURE_TYPE

#### See Also

Temperature::getFahrenheit

## Class: Temperature
## Defined in: Sensors/Temperature/_temperature_common.h

All temperature sensors support the following functions.

**Function Summary**

| | |
|---|---|
| TEMPERATURE_TYPE | getCelsius<br>Returns the temperature from the last read() in celsius. |
| TEMPERATURE_TYPE | Temperature::getFahrenheit<br>Convert a temperature from celsius to fahrenheit. |

**Function Details**

### getCelsius

Returns the temperature from the last read() in celsius.

#### Syntax

*temperature*.getCelsius()

Where *temperature* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

TEMPERATURE_TYPE

#### Example

Assuming you have called your sensor 'myTemp' in Project Designer:-

```
// Read the sensor and store the value
myTemp.read();

// Get value in celsius
TEMPERATURE_TYPE c = myTemp.getCelsius();

// Convert it to Fahrenheit
TEMPERATURE_TYPE f = Temperature::toFahrenheit(c);

// Print out the values
cout << "C=" << c << " F=" << f << '\n';
```

### Temperature::getFahrenheit

Convert a temperature from celsius to fahrenheit.

### Syntax

Temperature::getFahrenheit(celsius)

### Parameters

| Type | Name | Description |
|---|---|---|
| TEMPERATURE_TYPE | 'celsius' | The temperature in celsius. |

### Returns

TEMPERATURE_TYPE

## Class: Tpa81
## Defined in: Sensors/Temperature/Devantech/TPA81.h

The TPA81 also provides the following function:

| Function Summary | |
|---|---|
| TEMPERATURE_TYPE | getSensorCelsius<br>Returns the temperature, in celsius, from one of the individual temperature sensors. |

### Function Details

### getSensorCelsius

Returns the temperature, in celsius, from one of the individual temperature sensors.

#### Syntax

*tpa81*.getSensorCelsius(sensor)
Where *tpa81* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'sensor' | The index number of the individual sensor you want to access. This should be in the range 0 to 7. |

#### Returns

TEMPERATURE_TYPE

#### Note

This will return '0' if the specified individual sensor is invalid.

The function getCelsius() will return the ambient temperature across all sensors.

#### See Also

read
Temperature::getFahrenheit

## Class: Voltage
## Defined in: Sensors/Voltage/_voltage_common.h

All voltage sensors provide the following functions:

| Function Summary | |
|---|---|
| VOLTAGE_TYPE | getVolts<br>Returns the voltage at the time of the last read(). |

### Function Details

#### getVolts

Returns the voltage at the time of the last read().

#### Syntax

*voltage*.getVolts()
Where *voltage* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

VOLTAGE_TYPE

## Class: Somo14D
## Defined in: Audio/SOMO14D.h

The SOMO14D provides the following functions:

| Function Summary | |
|---|---|
| | play<br>Start playing the specified track number. |
| | stop<br>Stop playing. |
| | setVolume<br>Set the volume level. |
| `boolean` | isBusy<br>Returns TRUE if the device is currently playing a track or FALSE if not. |

## Function Details

### play

Start playing the specified track number.

#### Syntax

*somo14d*.play(track)
Where *somo14d* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint16_t | 'track' | The track number to play - in the range 0 to 511. |

#### Returns

None

---

### stop

Stop playing.

#### Syntax

*somo14d*.stop()
Where *somo14d* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

---

## setVolume
Set the volume level.

### Syntax
*somo14d*.setVolume(volume)
Where *somo14d* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'volume' | The volume level in the range 0 to 7. |

### Returns
None

---

## isBusy
Returns TRUE if the device is currently playing a track or FALSE if not.

### Syntax
*somo14d*.isBusy()
Where *somo14d* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
In order for this to work you must specify the pin to use for 'Input pin from Busy' in Project Designer.

---

## Class: TonePlayer
## Defined in: Audio/tone.h

The tone player supports the following functions:

| Function Summary | |
|---|---|
| | play<br>Play a single tone or a whole tune. |
| | stop<br>Stop playing the current tone or tune. |
| boolean | isPlaying<br>Return TRUE if a tone or tune is currently playing or FALSE if not. |
| | attach<br>Attach a callback function that is called when the current tone has stopped playing. |
| | detach<br>Remove any callback registered with this tone player. |

### Function Details

#### play

Play a single tone or a whole tune.

**Syntax**

*toneplayer*.play(frequency,durationMS)
*toneplayer*.play(tune)
Where *toneplayer* is the name you have given to the device in Project Designer.

**Parameters**

| Type | Name | Description |
|---|---|---|
| uint16_t | 'frequency' | The frequency of the tone you want to play or 0 if you want to add a pause. |
| uint32_t | 'durationMS' | The duration, in mS, of how long you want to play the tone for. A value of 0 will cause the note to be played continuously until either a new note is played or you call stop(). |

| Type | Name | Description |
|---|---|---|
| char * | 'tune' | The RTTTL string, in program memory, of the tune to be played. See http://en.wikipedia.org/wiki/Ring_Tone_Transfer_Language for more details on the specification. There are loads of web sites you can use to download free RTTTL files - Google is your friend. |

### Returns
None

### Example
Project Designer will create some example code to play the Star Wars theme music.

## stop
Stop playing the current tone or tune.

### Syntax
*toneplayer*.stop()
Where *toneplayer* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

## isPlaying
Return TRUE if a tone or tune is currently playing or FALSE if not.

### Syntax
*toneplayer*.isPlaying()
Where *toneplayer* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

## attach
Attach a callback function that is called when the current tone has stopped playing.

### Syntax

*toneplayer*.attach(callback)
Where *toneplayer* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| void (* callback)(TonePlayer& player) | 'callback' | The function you want WebbotLib to call when the current tone has stopped playing. |

### Returns

None

### Example

Assuming you have called your tone player 'tone' in Project Designer:-

```
void callback(TonePlayer& player){
    // The current tone has stopped
    // Play another note
    player.play(NOTE_F6,1000);

    // Don't notify me again
    player.detach();
}
// Initialise the software
TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    // Register my callback routine
    tone.attach(&callback);

    // Play the note of A in octave 4 (440Hz) for 2 seconds
    // after that time then 'callback' will be called
    tone.play(NOTE_A4, 2000);

    return 0;
}
```

### See Also

detach

## detach

Remove any callback registered with this tone player.

### Syntax

*toneplayer*.detach()
Where *toneplayer* is the name you have given to the device in Project Designer.

### Parameters

None

**Returns**

None

## Class: Pin
## Defined in: iopin.h

Digital I/O pins provide the following functions:

| Function Summary | |
|---|---|
| | high<br>Set an output pin to high. |
| | low<br>Set an output pin to low. |
| | toggle<br>Toggle an output pin. |
| | set<br>Set an output pin to the specified state. |
| boolean | get<br>Returns the current state of the input or output pin. |
| boolean | isHigh<br>Test whether the input or output pin is currently high. |
| boolean | isLow<br>Tests whether the input or output pin is currently low. |
| | makeOutput<br>Ensures that the pin is set as an output pin and sets it to the specified state. |
| | makeInput<br>Ensures that the pin is set as an input pin and enables, or disables, the internal pull-up resistor. |
| boolean | isInput<br>Returns TRUE if the pin is currently an input pin or FALSE if it is an output pin. |
| boolean | isOutput<br>Returns TRUE if the pin is currently an output pin or FALSE if it is an input pin. |
| TICK_COUNT | pulseIn<br>Measure the duration of an incoming pulse in μS. |

## Function Summary

| | |
|---|---|
| | pulseOut<br>Sends an output pulse of the specified duration. |

## Function Details

### high

Set an output pin to high.

#### Syntax

*pin*.high()

Where *pin* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

#### Note

If the pin is not currently configured as an output pin then this will cause a runtime error to be indicated.

---

### low

Set an output pin to low.

#### Syntax

*pin*.low()

Where *pin* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

#### Note

If the pin is not currently configured as an output pin then this will cause a runtime error to be indicated.

---

### toggle

Toggle an output pin. If it the pin was low it is set to high. If the pin was high it is set to low.

#### Syntax
*pin*.toggle()
Where *pin* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
None

#### Note
If the pin is not currently configured as an output pin then this will cause a runtime error to be indicated.

---

### set
Set an output pin to the specified state.

#### Syntax
*pin*.set(state)
Where *pin* is the name you have given to the device in Project Designer.

#### Parameters

| **Type** | **Name** | **Description** |
|---|---|---|
| boolean | 'state' | TRUE to set the pin high or FALSE to set the pin low. |

#### Returns
None

#### Note
If the pin is not currently configured as an output pin then this will cause a runtime error to be indicated.

#### See Also
makeOutput

---

### get
Returns the current state of the input or output pin.

For input pins it will return TRUE if the attached signal is high or FALSE if not.

For output pins it will return TRUE if the output is currently high or FALSE if not.

### Syntax
*pin*.get()
Where *pin* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
This is exactly the same as calling the 'isHigh' function.

---

## isHigh
Test whether the input or output pin is currently high.

For input pins it will return TRUE if the attached signal is high or FALSE if not.

For output pins it will return TRUE if the output is currently high or FALSE if not.

### Syntax
*pin*.isHigh()
Where *pin* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
This is exactly the same as calling the 'get' function.

---

## isLow
Tests whether the input or output pin is currently low.

For input pins it will return TRUE if the attached signal is low or FALSE if not.

For output pins it will return TRUE if the output is currently low or FALSE if not.

### Syntax
*pin*.isLow()
Where *pin* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

---

## makeOutput
Ensures that the pin is set as an output pin and sets it to the specified state.

### Syntax
*pin*.makeOutput(state)
Where *pin* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'state' | TRUE to set the pin high or FALSE to set the pin low. |

### Returns
None

---

## makeInput
Ensures that the pin is set as an input pin and enables, or disables, the internal pull-up resistor.

### Syntax
*pin*.makeInput(pullup)
Where *pin* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'pullup' | TRUE will enable the internal pull-up resistor and FALSE will disable it. If in doubt then use FALSE. |

### Returns
None

---

## isInput
Returns TRUE if the pin is currently an input pin or FALSE if it is an output pin.

### Syntax
*pin*.isInput()

Where *pin* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

boolean

---

## isOutput

Returns TRUE if the pin is currently an output pin or FALSE if it is an input pin.

### Syntax

*pin*.isOutput()
Where *pin* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

boolean

---

## pulseIn

Measure the duration of an incoming pulse in µS.

### Syntax

*pin*.pulseIn(activeHigh)
Where *pin* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'activeHigh' | TRUE if the pulse is __\|‾‾‾‾‾\|___ or FALSE if it is ‾‾‾\|___\|‾‾‾ |

### Returns

TICK_COUNT

### Note

This will ensure that the pin is set as an input pin.

---

## pulseOut

Sends an output pulse of the specified duration.

### Syntax

*pin*.pulseOut(duration,activeHigh)
Where *pin* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| TICK_COUNT | 'duration' | The duration of the pulse in µS. |
| boolean | 'activeHigh' | TRUE if the pulse is __\|-----\|___ or FALSE if it is ---\|___\|--- |

### Returns

None

### Note

This ensure that the pin is set as an output pin.

## Class: Led
## Defined in: led.h

An LED supports the following functions:

| Function Summary | |
|---|---|
| | on<br>Turns on the LED. |
| | off<br>Turns off the LED. |
| | set<br>Turn the LED on or off. |

## Function Details

### on

Turns on the LED.

#### Syntax

*led*.on()

Where *led* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

### off

Turns off the LED.

#### Syntax

*led*.off()

Where *led* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

## set

Turn the LED on or off.

### Syntax

*led*.set(state)
Where *led* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'state' | TRUE to turn on the LED or FALSE to turn it off. |

### Returns

None

## Class: Switch
## Defined in: switch.h

All switches/buttons support the following functions:

### Function Summary

| | |
|---|---|
| `boolean` | pressed<br>Returns TRUE if the switch/button is pressed (closed) or FALSE if released (open). |
| `boolean` | released<br>Returns TRUE if the switch/button is released (open) or FALSE if pressed (closed). |

### Function Details

### pressed

Returns TRUE if the switch/button is pressed (closed) or FALSE if released (open).

#### Syntax

*switch*.pressed()
Where *switch* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

boolean

### released

Returns TRUE if the switch/button is released (open) or FALSE if pressed (closed).

#### Syntax

*switch*.released()
Where *switch* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

boolean

## Class: Pwm
## Defined in: pwm.h

A generic PWM/analogue output provides the following functions:

| Function Summary | |
|---|---|
| PERCENTAGE | getPercent<br>Get the current duty cycle as a percentage. |
| | setPercent<br>Set the new duty cycle as a percentage. |

### Function Details

#### getPercent

Get the current duty cycle as a percentage.

**Syntax**

*pwm*.getPercent()

Where *pwm* is the name you have given to the device in Project Designer.

**Parameters**

None

**Returns**

PERCENTAGE

---

#### setPercent

Set the new duty cycle as a percentage.

**Syntax**

*pwm*.setPercent(duty)

Where *pwm* is the name you have given to the device in Project Designer.

**Parameters**

| Type | Name | Description |
|---|---|---|
| PERCENTAGE | 'duty' | The new duty cycle as a percentage in the range from 0 to 100. |

**Returns**

None

## Class: Color
## Defined in: color.h

Provides generic support for colour conversions.

The library is currently aware of both the RGB and YUV colour spaces and this module allows you to convert between the two. Note that not all of the colours in one colour space are exactly reproducible in another and hence converting from one colour space and back again will not necessarily give the same result.

A colour variable can be created using:-

```
Color myColor;
```

Although this variable will start with an 'unknown' value.

However: you can assign it a known value using the 'setRGB' and 'setYUV' commands.

You can copy one colour to another with the '=' operator just as if they were numbers ie:

```
Color myColor1, myColor2;
myColor1.setRGB(20,32,45);
myColor2 = myColor1;
```

You can also test if two colours are identical with the '==' and '!=' operators just as if they were numbers ie:

```
if( myColor1 == myColor2){
    cout << "Same";
}
```

You can convert to a particular colour space using the makeRGB and makeYUV functions. The return value of these functions then allow you to access the individual colour components.

| Function Summary | |
|---|---|
| | Color<br>Construct a colour with the specified r,g,b values. |
| | setRGB<br>Set a Color to an rgb value. |
| | setYUV<br>Set a Color to a yuv value. |
| `COLOR_RGB*` | makeRGB<br>Copy the colour and convert it, if necessary, to the RGB color module. |

## Function Summary

| | |
|---|---|
| COLOR_YUV* | makeYUV<br>Copy the colour and convert it, if necessary, to the YUV color module. |
| | dump<br>Dump the contents of this colour to the specified output or to stdout if not specified. |

## Function Details

### Color

Construct a colour with the specified r,g,b values.

#### Syntax

Color(r,g,b)

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'r' | The amount of red in the range 0 to 255. |
| uint8_t | 'g' | The amount of green in the range 0 to 255. |
| uint8_t | 'b' | The amount of blue in the range 0 to 255. |

#### Returns

None

### setRGB

Set a Color to an rgb value.

#### Syntax

*color*.setRGB(r,g,b)

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'r' | The amount of red in the range 0 to 255. |
| uint8_t | 'g' | The amount of green in the range 0 to 255. |
| uint8_t | 'b' | The amount of blue in the range 0 to 255. |

### Returns
None

---

## setYUV
Set a Color to a yuv value.

### Syntax
*color*.setYUV(y,u,v)

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'y' | The amount of 'y' in the range 0 to 255. |
| uint8_t | 'u' | The amount of 'u' in the range 0 to 255. |
| uint8_t | 'v' | The amount of 'v' in the range 0 to 255. |

### Returns
None

---

## makeRGB
Copy the colour and convert it, if necessary, to the RGB color module.

### Syntax
*color*.makeRGB()
*color*.makeRGB(dest)

### Parameters

| Type | Name | Description |
|---|---|---|
| Color | 'dest' | Where to store the resultant colour. |

### Returns
COLOR_RGB*

### Note
If the color is already RGB then no conversion will take place.

The return value allows you to access the red, green and blue components of the colour.

Repeatedly converting from one colour space to another and back again will cause a loss of detail.

### Example

```
// Create a YUV color
Color color1;
color1.setYUV(10,23,45);

// Copy it to color2 and
// convert color2 to RGB
Color color2;
COLOR_RGB* rgb = color1.makeRGB(color2);

// Print the rgb components
cout << "r=" << rgb->r << " g=" << rgb->g << " b=" << rgb->b;
```

## makeYUV

Copy the colour and convert it, if necessary, to the YUV color module.

### Syntax

*color*.makeYUV()
*color*.makeYUV(dest)

### Parameters

| **Type** | **Name** | **Description** |
|---|---|---|
| Color | 'dest' | Where to store the resultant colour. |

### Returns

COLOR_YUV*

### Note

If the color is already YUV then no conversion will take place.

The return value allows you to access the y, u and v components of the colour.

Repeatedly converting from one colour space to another and back again will cause a loss of detail.

### Example

```
// Create an RGB color
Color color1;
color1.setRGB(10,23,45);

// Copy it to color2 and
// convert color2 to YUV
Color color2;
COLOR_YUV* yuv = color1.makeYUV(color2);

// Print the yuv components
cout << "y=" << yuv->y << " u=" << yuv->u << " v=" << yuv->v;
```

## dump

Dump the contents of this colour to the specified output or to stdout if not specified.

### Syntax

*color*.dump(dest)
*color*.dump()

### Parameters

| Type | Name | Description |
|---|---|---|
| FILE * | 'dest' | Specifies the output device that you want the dump to be sent to. |

### Returns

None

## Class: Camera
## Defined in: Cameras/_camera_common.h

All cameras provide the following functions:

**Function Summary**

| | |
|---|---|
| uint16_t | getXresolution<br>Return the camera X resolution (ie number of pixels across). |
| uint16_t | getYresolution<br>Return the camera Y resolution (ie number of pixels down). |
| uint8_t | getNumColorBins<br>Returns the number of colour bins supported by this camera. |
| boolean | setBin<br>Sets the value of a given colour bin. |
| uint8_t | getBlobs<br>Return the number of blobs which match the specified colour bin (or all colour bins). |
| | fetchBlob<br>Return the given blob from the list in the blob variable you have specified. |
| boolean | getPixel<br>Returns the colour of a given pixel. |
| char * | getVersion<br>Returns the version number of the firmware installed on the camera (if available). |
| | setMinBlobSize<br>Reduces the amount of data returned by the camera by specifying the smallest blob size (width * height) that we are interested in. |

### Function Details

### getXresolution

Return the camera X resolution (ie number of pixels across).

#### Syntax

*camera*.getXresolution()

Where *camera* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
uint16_t

### Note
Some cameras allow you to change the resolution.

---

## getYresolution
Return the camera Y resolution (ie number of pixels down).

### Syntax
*camera*.getYresolution()
Where *camera* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
uint16_t

### Note
Some cameras allow you to change the resolution.

---

## getNumColorBins
Returns the number of colour bins supported by this camera.

### Syntax
*camera*.getNumColorBins()
Where *camera* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
uint8_t

### Note
This number is constant for each camera.

## setBin

Sets the value of a given colour bin.

This will return FALSE if the specified colour bin is higher than the number of bins this camera can support.

The min and max colour values represent two diagonally opposite corners of a colour cube.

### Syntax

*camera*.setBin(bin,min,max)
Where *camera* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'bin' | The colour bin in the range 0 to number of bins - 1. |
| Color | 'min' | The minimum colour for the bin. |
| Color | 'max' | The maximum colour for the bin. |

### Returns

boolean

### Note

The min/max colours can be specified in any colour space you like (ie RGB or YUV) and will be automatically converted internally into the colour space used by the camera. Note that the sending of the colour bin to the camera may be delayed until such time as they will be used.

### Example

To set colour bin #2 to cover a range of REDs on a camera called 'camera1' then we could do:-

```
Color min,max; // create two empty variables
min.setRGB(128,0,0); // set min to RGB(128,0,0)
max.setRGB(255,128,128); // set max to RGB(255,128,128)
camera1.setBin(2, min, max); // Set the color bin
```

## getBlobs

Return the number of blobs which match the specified colour bin (or all colour bins).

This will ask the camera to return the matching blob areas which match the specified colour bin. The colour bin can either by any individual bin or can be the 'magic' value of CAMERA_ALL_BINS to mean ANY camera bin.

The value returned is the number of matching blobs. This can be used along with fetchBlob to iterate through the list of blobs.

### Syntax

*camera*.getBlobs(bin)
Where *camera* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'bin' | The colour bin in the range 0 to number of bins - 1. |

### Returns

uint8_t

### Note

The list returned is always sorted into descending rectangle size - so the biggest rectangle is always at the top of the list.

### Example

```
uint8_t blobs = camera.getBlobs( CAMERA_ALL_BINS);
if(blobs == 0){
    // No blobs are visible
}else{
    for(uint8_t b = 0; b<blobs; b++){
        // Fetch the blobs - the biggest first
        Blob blob;
        camera.fetchBlob(b, blob);

        // See Cameras/_camera_common.h to see the fields in a CAMERA_BLOB
        // But in summary these include the left, right, top and bottom of
        // the rectangle. The x,y position of the center of the blob relative
        // to the center of the screen. The total size of the blob and the
        // the colour bin.
        cout << "Center=" << blob.xCenter << ',' << blob.yCenter;
        cout << " Size=" << blob.pixels;
    }
}
```

### See Also

setMinBlobSize

## fetchBlob

Return the given blob from the list in the blob variable you have specified.

### Syntax

*camera*.fetchBlob(blobNo,blob)
Where *camera* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'blobNo' | The blob number you want to access. |
| Blob | 'blob' | The blob variable where the returned information will be stored. |

### Returns
None

### See Also
getBlobs

## getPixel
Returns the colour of a given pixel.

The return value is FALSE if there was an error talking to the camera in which case the returned colour has an unreliable value. A value of TRUE means that the function has completed and a returned colour is available.

### Syntax
*camera*.getPixel(x,y,rtn)
Where *camera* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint16_t | 'x' | The x co-ordinate of the pixel. |
| uint16_t | 'y' | The y co-ordinate of the pixel. |
| Color | 'rtn' | The variable to return the color. |

### Returns
boolean

### Example
Example to read the pixel at 10,10 :-

```
uint16_t x = 10;
uint16_t y = 10;
Color color;
if(camera.getPixel(x, y, color)){
    // The color is in 'color' so lets dump it out
    color.dump();
}else{
    // We had problems talking to the camera
}
```

## getVersion

Returns the version number of the firmware installed on the camera (if available).

### Syntax

*camera*.getVersion()
Where *camera* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

char *

## setMinBlobSize

Reduces the amount of data returned by the camera by specifying the smallest blob size (width * height) that we are interested in. The default value is 4 pixels (ie 2x2, 4x1, or 1x4).

### Syntax

*camera*.setMinBlobSize(minSize)
Where *camera* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint32_t | 'minSize' | The minimum size of the blobs. |

### Returns

None

## Class: Blackfin
## Defined in: Cameras/Surveyor/blackfin.h

The Surveyor Blackfin camera also provides the following functions:

| Function Summary | |
|---|---|
| | setResolution<br>Changes the working resolution of the camera. |
| `BLACKFIN_RESOLUTION` | getResolution<br>Returns the current camera resolution. |
| `uint32_t` | countPixels<br>Returns the number of pixels that match a given colour bin either for a rectangle or for the full image. |
| | getMeanColor<br>Get the mean colour across the whole camera image. |

## Function Details

### setResolution

Changes the working resolution of the camera.

This will automatically add a two second delay to allow the camera to adjust to the new resolution.

#### Syntax

*blackfin*.setResolution(resolution)
Where *blackfin* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| BLACKFIN_RESOLUTION | 'resolution' | The new resolution required. |

#### Returns

None

#### Note

Since it takes a number of seconds for the camera to settle down after a resolution change then it is not something that you should be doing on a frequent basis.

Project Designer allows you to set the default resolution - so there should be little reason to change it within your own code.

## getResolution

Returns the current camera resolution.

### Syntax

*blackfin*.getResolution()
Where *blackfin* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

BLACKFIN_RESOLUTION

## countPixels

Returns the number of pixels that match a given colour bin either for a rectangle or for the full image.

### Syntax

*blackfin*.countPixels(bin)
*blackfin*.countPixels(bin,x1,x2,y1,y2)
Where *blackfin* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'bin' | The colour bin in the range 0 to number of bins - 1. |
| uint16_t | 'x1' | Either the left or right edge of the rectangle. |
| uint16_t | 'x2' | Either the left or right edge of the rectangle. |
| uint16_t | 'y1' | Either the top or bottom edge of the rectangle. |
| uint16_t | 'y2' | Either the top or bottom edge of the rectangle. |

### Returns

uint32_t

## getMeanColor

Get the mean colour across the whole camera image.

### Syntax

*blackfin*.getMeanColor(rtn)

Where *blackfin* is the name you have given to the device in Project Designer.

## Parameters

| Type | Name | Description |
|---|---|---|
| Color | 'rtn' | The variable to return the color. |

## Returns

None

## Class: SonyPS2
## Defined in: Controller/Sony/ps2.h

The Sony PS2 controller supports the following functions:

| Function Summary | |
|---|---|
| boolean | calibrate<br>Calibrate the joysticks on the controller. |
| boolean | read<br>Read the values from the controller and store them. |
| | setRumble<br>Turn each of the rumble motors on or off. |
| boolean | buttonDown<br>Return TRUE if the button has just been pressed or FALSE if it was already pressed or is not pressed at all. |
| boolean | buttonUp<br>Return TRUE if the button has just been released or FALSE if it was already released or is still held down. |
| boolean | buttonHeld<br>Return TRUE if the button continues to be held down. |
| boolean | buttonPressed<br>Return TRUE if the button is currently pressed. |
| uint8_t | buttonPressure<br>Return a value representing how hard a button has been pressed. |
| PS2_BUTTONS | buttonsChanged<br>Returns information on which buttons have changed state since the previous call to read the controller. |
| PS2_BUTTONS | buttonsRaw<br>Return the status of all 16 buttons. |
| boolean | isAnalogMode<br>Returns TRUE if the controller is already in analogue mode. |
| boolean | setAnalogMode<br>Activates the joysticks. |

## Function Summary

| | |
|---|---|
| int8_t | joystick<br>Read a joystick value relative to its centre point including any dead zone. |
| int8_t | joystickRaw<br>Reads the raw value from a joystick. |

## Function Details

### calibrate

Calibrate the joysticks on the controller.

This command should only be called once - when your program starts up - appInitSoftware is a good place. It will activate the joysticks and then read their current positions and remember these as being the 'centre' locations.

Without this call you may notice some non-zero readings from the joystick if it is set to the centre position.

You can also specify a 'deadzone' - ie the radius around the centre point that will be considered as being the centre. Try a value of around 27.

This will return TRUE if the calibration was successful or FALSE if communication with the controller could not be done.

#### Syntax

*sonyps2*.calibrate(deadzone)
Where *sonyps2* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'deadzone' | The radius around the centre point that will be considered as being the centre. |

#### Returns

boolean

### read

Read the values from the controller and store them.

All of the functions that return joystick values and button states work with the data last read by this command.

The function will return TRUE if the controller was read successfully or FALSE if there was a problem.

#### Syntax
*sonyps2*.read()
Where *sonyps2* is the name you have given to the device in Project Designer.

#### Parameters
None

#### Returns
boolean

---

### setRumble
Turn each of the rumble motors on or off.

#### Syntax
*sonyps2*.setRumble(rumbleLeft,rumbleRight)
Where *sonyps2* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'rumbleLeft' | Set the left rumble motor to a value between 0 (off) and 255 (maximum). |
| boolean | 'rumbleRight' | TRUE to turn on or FALSE to turn off. |

#### Returns
None

#### Note
The motor settings will only change when the next call to read the controller is used.

---

### buttonDown
Return TRUE if the button has just been pressed or FALSE if it was already pressed or is not pressed at all.

#### Syntax
*sonyps2*.buttonDown(button)
Where *sonyps2* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PS2_BUTTON | 'button' | The single button to test. |

### Returns

boolean

## buttonUp

Return TRUE if the button has just been released or FALSE if it was already released or is still held down.

### Syntax

*sonyps2*.buttonUp(button)
Where *sonyps2* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PS2_BUTTON | 'button' | The single button to test. |

### Returns

boolean

## buttonHeld

Return TRUE if the button continues to be held down.

### Syntax

*sonyps2*.buttonHeld(button)
Where *sonyps2* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PS2_BUTTON | 'button' | The single button to test. |

### Returns

boolean

## buttonPressed

Return TRUE if the button is currently pressed.

### Syntax

*sonyps2*.buttonPressed(button)

Where *sonyps2* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PS2_BUTTON | 'button' | The single button to test. |

### Returns

boolean

---

## buttonPressure

Return a value representing how hard a button has been pressed.

This will return 0 if the button is not pressed at all up to a value of 255 if its has been pressed hard. If the controller has not been put into analog mode then it will only return 0 or 255.

### Syntax

*sonyps2*.buttonPressure(button)
Where *sonyps2* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PS2_BUTTON | 'button' | The single button to test. |

### Returns

uint8_t

---

## buttonsChanged

Returns information on which buttons have changed state since the previous call to read the controller.

The returned value is the combination of all the buttons that have changed or 0 if no buttons have been pressed or released.

This allows you to perform a quick check to test if anything has changed.

### Syntax

*sonyps2*.buttonsChanged()
Where *sonyps2* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns
PS2_BUTTONS

## buttonsRaw
Return the status of all 16 buttons.

The returned value has one bit per button. The bit is set if that button has been pressed.

### Syntax
*sonyps2*.buttonsRaw()
Where *sonyps2* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
PS2_BUTTONS

### Example
This function is useful if you want to do a 'Press or release any button to continue' function ie

```
// Read and save the current button state
ps2.read();
PS2_BUTTONS current = ps2.buttonsRaw();
do{
    // Read the buttons again
    ps2.read();
} while (ps2.buttonsRaw() == current);
```

You can also 'OR' together various buttons. So to test if the SELECT and START buttons have both been pressed:-

```
#define COMBO (PS2_BTN_SELECT | PS2_BTN_START)
ps2.read();
if( (ps2.buttonsRaw() & COMBO) == COMBO){
    // Both pressed
}
```

## isAnalogMode
Returns TRUE if the controller is already in analogue mode.

### Syntax
*sonyps2*.isAnalogMode()
Where *sonyps2* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
This is only valid after one successful 'read' command has been issued.

## setAnalogMode
Activates the joysticks.

Unless this command is issued successfully then you will only be able to access the pushbuttons on the controller.

Returns FALSE if communication with the controller failed.

### Syntax
*sonyps2*.setAnalogMode()
Where *sonyps2* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

### Note
Note that the 'calibrate' function will automatically try to call this command.

## joystick
Read a joystick value relative to its centre point including any dead zone.

The returned value will be 0 if the value is within the centre dead zone. Negative values represent 'left' or 'up, and positive values represent 'right' or 'down'.

### Syntax
*sonyps2*.joystick(stick)
Where *sonyps2* is the name you have given to the device in Project Designer.

### Parameters
| Type | Name | Description |
|---|---|---|
| PS2_STICK | 'stick' | The joystick to test. |

### Returns
int8_t

## joystickRaw
Reads the raw value from a joystick.

The returned value will be in the range 0 to 255 and ignores any calibration and dead zone settings.

### Syntax
*sonyps2*.joystickRaw(stick)
Where *sonyps2* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| PS2_STICK | 'stick' | The joystick to test. |

### Returns
int8_t

### See Also
calibrate

## Class: SegLED
## Defined in: segled.h

An 8 segment LED supports the following functions:

| Function Summary | |
|---|---|
| | on<br>Turn on an individual segment of the LED display. |
| | off<br>Turn off an individual segment of the LED display. |
| | set<br>Turns an individual segment on or off. |
| uint8_t | put<br>Output a character to the LED display. |

### Function Details

**on**

Turn on an individual segment of the LED display.

#### Syntax

*segled*.on(segment)
Where *segled* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| SEGLED_SEGMENT | 'segment' | The segment of the LED to be changed. |

#### Returns

None

---

**off**

Turn off an individual segment of the LED display.

#### Syntax

*segled*.off(segment)
Where *segled* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| SEGLED_SEGMENT | 'segment' | The segment of the LED to be changed. |

### Returns

None

## set

Turns an individual segment on or off.

### Syntax

*segled*.set(segment,state)
Where *segled* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| SEGLED_SEGMENT | 'segment' | The segment of the LED to be changed. |
| boolean | 'state' | TRUE to turn the segment on or FALSE to turn it off. |

### Returns

None

## put

Output a character to the LED display.

The return value is the same character.

Given the restrictions of such a simple display then only certain characters can be shown.

Currently supported are:

- A to Z
- a to z
- 0 to 9
- . (Decimal point or full stop)
- - (minus sign or hyphen)
- {[(
- }])
- All other characters will be displayed as spaces ie all segments will be off

### Syntax

*segled*.put(c)

Where *segled* is the name you have given to the device in Project Designer.

## Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'c' | The character to write to the LED. |

## Returns

uint8_t

## Class: I2C_Eeprom
## Defined in: Storage/i2cEEPROM.h

An I2C EEPROM provides the following functions:

| Function Summary | |
|---|---|
| uint8_t | readByte<br>Reads a single byte from the EEPROM. |
| | readBytes<br>Read a number of bytes from the EEPROM into memory. |
| uint8_t | writeByte<br>Writes a single byte to the EEPROM. |
| | writeBytes<br>Writes a number of bytes from memory to the EEPROM. |
| EEPROM_ADDR | totalBytes<br>Returns the total number of bytes the EEPROM can store. |

## Function Details

### readByte

Reads a single byte from the EEPROM.

#### Syntax

*i2c_eeprom*.readByte(raddr)
Where *i2c_eeprom* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| EEPROM_ADDR | 'raddr' | The location in the EEPROM that you want to read from. |

#### Returns

uint8_t

---

### readBytes

Read a number of bytes from the EEPROM into memory.

#### Syntax

*i2c_eeprom*.readBytes(raddr,dest,numBytes)
Where *i2c_eeprom* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| EEPROM_ADDR | 'raddr' | The location in the EEPROM that you want to read from. |
| void * | 'dest' | The start address of where you want to store the read bytes. |
| size_t | 'numBytes' | The number of bytes. |

#### Returns
None

---

### writeByte
Writes a single byte to the EEPROM.

#### Syntax
*i2c_eeprom*.writeByte(waddr)
Where *i2c_eeprom* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| EEPROM_ADDR | 'waddr' | The location in the EEPROM that you want to write to. |

#### Returns
uint8_t

---

### writeBytes
Writes a number of bytes from memory to the EEPROM.

#### Syntax
*i2c_eeprom*.writeBytes(waddr,src,numBytes)
Where *i2c_eeprom* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| EEPROM_ADDR | 'waddr' | The location in the EEPROM that you want to write to. |
| void * | 'src' | The start address of where you want to write the bytes from. |
| size_t | 'numBytes' | The number of bytes. |

### Returns
None

## totalBytes
Returns the total number of bytes the EEPROM can store.

### Syntax
*i2c_eeprom*.totalBytes()
Where *i2c_eeprom* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
EEPROM_ADDR

## Class: SPI_Eeprom
## Defined in: Storage/spiEEPROM.h

An SPI EEPROM provides the following functions:

| Function Summary | |
|---|---|
| uint8_t | readByte<br>Reads a single byte from the EEPROM. |
| | readBytes<br>Read a number of bytes from the EEPROM into memory. |
| uint8_t | writeByte<br>Writes a single byte to the EEPROM. |
| | writeBytes<br>Writes a number of bytes from memory to the EEPROM. |
| EEPROM_ADDR | totalBytes<br>Returns the total number of bytes the EEPROM can store. |

## Function Details

### readByte

Reads a single byte from the EEPROM.

#### Syntax

*spi_eeprom*.readByte(raddr)
Where *spi_eeprom* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| EEPROM_ADDR | 'raddr' | The location in the EEPROM that you want to read from. |

#### Returns

uint8_t

---

### readBytes

Read a number of bytes from the EEPROM into memory.

#### Syntax

*spi_eeprom*.readBytes(raddr,dest,numBytes)
Where *spi_eeprom* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| EEPROM_ADDR | 'raddr' | The location in the EEPROM that you want to read from. |
| void * | 'dest' | The start address of where you want to store the read bytes. |
| size_t | 'numBytes' | The number of bytes. |

### Returns
None

## writeByte
Writes a single byte to the EEPROM.

### Syntax
*spi_eeprom*.writeByte(waddr)
Where *spi_eeprom* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| EEPROM_ADDR | 'waddr' | The location in the EEPROM that you want to write to. |

### Returns
uint8_t

## writeBytes
Writes a number of bytes from memory to the EEPROM.

### Syntax
*spi_eeprom*.writeBytes(waddr,src,numBytes)
Where *spi_eeprom* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| EEPROM_ADDR | 'waddr' | The location in the EEPROM that you want to write to. |
| void * | 'src' | The start address of where you want to write the bytes from. |
| size_t | 'numBytes' | The number of bytes. |

### Returns
None

---

## totalBytes
Returns the total number of bytes the EEPROM can store.

### Syntax
*spi_eeprom*.totalBytes()
Where *spi_eeprom* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
EEPROM_ADDR

---

## Class: SdCard
## Defined in: Storage/sdCard.h

An SD Card supports the following functions:

| Function Summary | |
|---|---|
| boolean | readSectors<br>Read one or more 512 byte sectors from the card into memory. |
| boolean | writeSectors<br>Write one or more 512 byte sectors from memory to the card. |
| uint32_t | totalSectors<br>Returns the total number of 512 byte sectors on the card. |

## Function Details

### readSectors

Read one or more 512 byte sectors from the card into memory.

Returns TRUE if the card was successfully accessed or FALSE if there was an error.

#### Syntax

*sdcard*.readSectors(absSector,dst,numSectors)
Where *sdcard* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint32_t | 'absSector' | The sector number to access on the card. |
| void * | 'dst' | The memory address to read the sector data to. |
| uint8_t | 'numSectors' | The number of 512 byte sectors to transfer. |

#### Returns

boolean

### writeSectors

Write one or more 512 byte sectors from memory to the card.

Returns TRUE if the card was successfully accessed or FALSE if there was an error.

### Syntax

*sdcard*.writeSectors(absSector,src,numSectors)
Where *sdcard* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint32_t | 'absSector' | The sector number to access on the card. |
| void * | 'src' | The memory address to write the data from. |
| uint8_t | 'numSectors' | The number of 512 byte sectors to transfer. |

### Returns

boolean

---

## totalSectors

Returns the total number of 512 byte sectors on the card.

### Syntax

*sdcard*.totalSectors()
Where *sdcard* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

uint32_t

---

## Class: Disk
## Defined in: Storage/FileSystem/FAT.h

A Disk represents a FAT file system and provides the following the functions:

| Function Summary | |
|---|---|
| `boolean` | isReady<br>Returns TRUE if the disk has been successfully initialised or FALSE if there was a problem. |
| `boolean` | flush<br>Writes any un-saved changed out to the disk. |
| `uint32_t` | freeKB<br>Returns the amount of free space on the disk in kilobytes (1024 bytes). |
| `int8_t` | mkDir<br>Create a directory. |
| `int8_t` | open<br>Open a file. |
| `boolean` | remove<br>Delete a file. |
| `boolean` | exists<br>Test if a file already exists. |
| `boolean` | findFirst<br>Find the first file entry in a directory. |

## Function Details

### isReady

Returns TRUE if the disk has been successfully initialised or FALSE if there was a problem.

The problem could be that there is no disk inserted or it has not been formatted.

#### Syntax

*disk*.isReady()

#### Parameters

None

### Returns
boolean

---

## flush
Writes any un-saved changed out to the disk.

Returns TRUE if successful or FALSE if there was an error of some kind.

### Syntax
*disk*.flush()

### Parameters
None

### Returns
boolean

---

## freeKB
Returns the amount of free space on the disk in kilobytes (1024 bytes).

### Syntax
*disk*.freeKB()

### Parameters
None

### Returns
uint32_t

### Example
Assuming your disk is called 'disk_sdCard' then:

```
cout.print_P( PSTR("Free space=") );
cout.print( disk_sdCard.freeKB());
cout.print_P( PSTR("kb\n") );
```

---

## mkDir
Create a directory.

The returned value can be:

```
0 if the directory was successfully created
-1 if it already exists
-2 if the parent directory has run out of space
-3 if the disk is full
```

### Syntax

*disk*.mkDir(dirName)

### Parameters

| Type | Name | Description |
|---|---|---|
| const char * | 'dirName' | The name of directory to be created. |

### Returns

int8_t

---

## open

Open a file.

The function will return 0 if the file has been opened successfully in which case the File variable is initialised and can then be used in the remaining file operations.

A negative return value indicates an error as follows:-

```
-1 = The file doesn't exist when opening for read
-2 = The file already exists when opening for write
-3 = The directory doesn't exist
-4 = You are trying to write a read only file
-5 = The disk is full
-6 = The mode parameter is invalid
```

### Syntax

*disk*.open(file,fileName,mode)

### Parameters

| Type | Name | Description |
|---|---|---|
| File | 'file' | The "*File*" (see page 412) variable to store the details on the open file. |
| const char * | 'fileName' | The filename in DOS format.<br><br>ie the filename can be 8 characters, a full stop, followed by a 3 character extension. |

| Type | Name | Description |
|---|---|---|
| char | 'mode' | Indicates how you want to use the file: |

```
'r' - To open an existing file for read access
'w' - To create a new file for read write access
'a' - To write to the end of an existing file (ie
append), or create the file if it doesn't exist.
```

### Returns
int8_t

### See Also
close

## remove
Delete a file.

This will return FALSE if the file doesn't exist or is marked as read-only, or TRUE if it was deleted successfully.

### Syntax
*disk*.remove(fileName)

### Parameters

| Type | Name | Description |
|---|---|---|
| const char * | 'fileName' | The filename in DOS format.<br><br>ie the filename can be 8 characters, a full stop, followed by a 3 character extension. |

### Returns
boolean

### Note
It's up to you to make sure that you don't delete a file that you still have open for reading or writing. Doing so would have unknown results.

## exists
Test if a file already exists.

Return TRUE if it does exist, or FALSE if it doesn't.

### Syntax
*disk*.exists(fileName)

### Parameters

| Type | Name | Description |
|---|---|---|
| const char * | 'fileName' | The filename in DOS format.<br><br>ie the filename can be 8 characters, a full stop, followed by a 3 character extension. |

### Returns
boolean

## findFirst
Find the first file entry in a directory.

Returns FALSE if the directory doesn't exist or contains no files. Otherwise it will return TRUE and the specified iterator is initialised with the first file entry.

### Syntax
*disk*.findFirst(iterator,dirFind)

### Parameters

| Type | Name | Description |
|---|---|---|
| FileIterator | 'iterator' | The FileIterator variable that is to be initialised by this call. |
| const char * | 'dirFind' | The name of the directory to search within. |

### Returns
boolean

### Example
Assuming your disk is called disk_sdCard then you can list the files in the root folder using:

```
FileIterator f;
boolean ok = disk.findFirst(f, "/");
while(ok){
    cout.print( f.getName() );
    count.print('(').print(f.getSize()).print_P(PSTR(") bytes"))
    if(f.isDirectory(){
        cout.print_P( PSTR(" <dir>") );
    }
    cout.println();
    ok = f.findNext();
}
```

## Class: FileIterator
## Defined in: Storage/FileSystem/FAT.h

A FileIterator allows you step through the contents of a directory and has the following functions:

### Function Summary

| | |
|---|---|
| `const char *` | getName<br>Return the filename of the current entry. |
| `uint32_t` | getSize<br>Returns the file size in bytes. |
| `boolean` | isDirectory<br>Returns TRUE if the current entry is a directory or FALSE if it is a file. |
| `boolean` | findNext<br>Return the next file in the directory. |

### Function Details

### getName

Return the filename of the current entry.

#### Syntax

*fileiterator*.getName()

#### Parameters

None

#### Returns

const char *

---

### getSize

Returns the file size in bytes.

#### Syntax

*fileiterator*.getSize()

#### Parameters

None

#### Returns
uint32_t

---

### isDirectory
Returns TRUE if the current entry is a directory or FALSE if it is a file.

#### Syntax
*fileiterator*.isDirectory()

#### Parameters
None

#### Returns
boolean

---

### findNext
Return the next file in the directory.

Returns FALSE if there are no more files.

#### Syntax
*fileiterator*.findNext()

#### Parameters
None

#### Returns
boolean

#### See Also
findFirst

---

## Class: I2cBus
## Defined in: i2cBus.h

An I2C bus provides the following low-level functions:

| Function Summary | |
|---|---|
| | setBitRate<br>Sets the communications speed of the I2C bus. |
| boolean | start<br>A low level function to start communicating with a given slave device if you are writing your own I2C communications from scratch. |
| | stop<br>This is a low-level call to indicate that the communication session has finished (ie 'hang up the phone'). |
| uint8_t | get<br>This is a low-level function to read, and return, a byte from the bus. |
| boolean | put<br>This is a low-level function to write a byte across the bus. |

## Function Details

### setBitRate

Sets the communications speed of the I2C bus.

When the bus is initialised it is set to use a default of 100 kilo-bits per second which is satisfactory for most devices.

If some devices fail to work as expected then check the data sheet for the device to find the required bit rate. If you have more than one device connected to the same bus then you should check the maximum speed that they can each cope with - and then select the lowest number.

The speed only needs to be set once and is therefore best done in 'appInitSoftware'.

A planned future enhancement is to set up the speed for each device in Project Designer so that it can generate the correct code to set the best bit rate.

#### Syntax

*i2cbus*.setBitRate(bitrateKHz)

Where *i2cbus* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint16_t | 'bitrateKHz' | The communication speed in kilo bits per second ie a value of '100' means '100 kilo bits per second' or '100,000 bits per second'. |

### Returns
None

### Note
You can only change the speed of a hardware bus.

### Example
To change the speed of a bus called 'i2cMaster' to use 10,000 bits per second:

```
i2cMaster.setBitRate(10);
```

## start
A low level function to start communicating with a given slave device if you are writing your own I2C communications from scratch.

This will try to start communicating with the given device. It's like dialling a phone number. If there is no reply then the function will return FALSE, otherwise it will return TRUE.

Once a communication link has been established you can use get() (if you are in read/listen mode) or put() (if you are in write/talk mode).

You can swap between read and write mode by re-issuing another i2cStart to the same device with another mode (called a 'repeated start').

At the end of the conversation, and before you can contact a different device, you **must** hang up the phone by calling stop().

### Syntax
*i2cbus*.start(i2cAddr,writeMode)
Where *i2cbus* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'i2cAddr' | The address of the device you want to communicate with. |
| boolean | 'writeMode' | TRUE for write/talk mode or FALSE if read/listen mode. |

### Returns
boolean

## stop
This is a low-level call to indicate that the communication session has finished (ie 'hang up the phone').

Once a start() has been issued and you have finished talking to the device then you must terminate the call with a stop().

### Syntax
*i2cbus*.stop()
Where *i2cbus* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

## get
This is a low-level function to read, and return, a byte from the bus.

### Syntax
*i2cbus*.get(isLastByte)
Where *i2cbus* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'isLastByte' | Specifies whether you will be reading more bytes (FALSE) or this is the last one (TRUE). |

### Returns
uint8_t

### Note
Note that this can only be used after a successful call to start() has been used to place the bus into read mode.

## put
This is a low-level function to write a byte across the bus.

The function returns TRUE if successful or FALSE if not.

### Syntax

*i2cbus*.put(data)

Where *i2cbus* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'data' | The data byte to be sent. |

### Returns

boolean

### Note

Note that this can only be used after a successful call to start() has been used to place the bus into write mode.

## Class: I2cDevice
## Defined in: i2cBus.h

All I2C devices support the following functions:

| Function Summary | |
|---|---|
| boolean | readRegisters<br>A complete communication session to read one or more sequential registers from the device. |
| boolean | receive<br>Performs a complete communication with a slave device to receive a number of bytes. |
| boolean | writeRegisters<br>Performs a complete communication session with a device to write to a sequential series of registers. |
| boolean | writeRegister<br>Performs a complete communication session with a device to write to a value to a single register. |
| boolean | send<br>Performs a complete communication session with a device to write a number of bytes. |
| boolean | transfer<br>Performs a complete communication with a slave device to write a number of bytes and then read a number of bytes. |

## Function Details

### readRegisters

A complete communication session to read one or more sequential registers from the device. Returns TRUE if successful or FALSE if there was an error.

#### Syntax

*i2cdevice*.readRegisters(startReg,responseLen,response)
Where *i2cdevice* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'startReg' | The start register number. |

| Type | Name | Description |
|---|---|---|
| size_t | 'responseLen' | The number of bytes to be received. |
| uint8_t * | 'response' | The memory address to store the returned the values. |

### Returns

boolean

### Note

This is the same as the following **pseudo** code:

```
// Connect in write mode
i2cStart( WRITE );
// Write the start register
i2cPut( startreg );
// Connect in read mode
i2cStart( READ );
for(size_t i=0; i<responseLen; i++){
    response[i] = i2cGet();
}
// Hang up
i2cStop();
```

### Example

Assuming you have created a 'Generic I2C Device' in Project Designer called myDevice and you want to read the two registers 10 and 11 then you would use:

```
// Create an array to hold the two byte response
uint8_t response[2];
if( myDevice.readRegisters(10, 2, response) ){
    uint8_t reg10 = response[0];
    uint8_t reg11 = response[1];
}else{
    // Communication failed
}
```

Note how the C operator 'sizeof' can be used to make sure that the number of bytes being read is actually the same as the number of bytes in the 'response' array by changing the 'read' as follows:-

```
myDevice.readRegisters(10, sizeof(response), response)
```

You are encouraged to use this technique as it means that if you decide, later on, to read 4 bytes then you can just change the 'response' definition to:

```
uint8_t response[4];
```

and nothing else needs to change.

## receive

Performs a complete communication with a slave device to receive a number of bytes.

The return value specifies if it was successful (TRUE) or not (FALSE).

### Syntax

*i2cdevice*.receive(responseLen,response)
Where *i2cdevice* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| size_t | 'responseLen' | The number of bytes to be received. |
| uint8_t * | 'response' | The memory address to store the returned the values. |

### Returns

boolean

### Note

This is the same as the following **pseudo** code:

```
// Connect in read mode
i2cStart( READ );
for(size_t i=0; i<responseLen; i++){
    response[i] = i2cGet();
}
// Hang up
i2cStop();
```

## writeRegisters

Performs a complete communication session with a device to write to a sequential series of registers.

Returns TRUE if successful or FALSE if there was an error.

### Syntax

*i2cdevice*.writeRegisters(startReg,writeLen,writeData)
Where *i2cdevice* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'startReg' | The start register number. |
| size_t | 'writeLen' | The number of bytes to be written. |

| Type | Name | Description |
|---|---|---|
| uint8_t * | 'writeData' | The memory address of the data to be written. |

### Returns
boolean

### Note
This is the same as the following **pseudo** code:

```
// Connect in write mode
i2cStart( WRITE );
// Write the start register
i2cPut( startreg );
// Write the data values
for(size_t i=0; i<writeLen; i++){
    i2cPut(writeData[writeLen]);
}
// Hang up
i2cStop();
```

### Example
Assuming you have created a 'Generic I2C Device' in Project Designer called myDevice and you want to set the two registers 10 and 11 to the values 1 and 2 then you would use:

```
// Create an array of the values to be written
uint8_t data[] = { 1, 2 };
if( myDevice.writeRegisters(10, 2, data) ){
    // ok
}else{
    // Communication failed
}
```

Note how the C operator 'sizeof' can be used to make sure that the number of bytes being read is actually the same as the number of bytes in the 'data' array by changing the 'write' as follows:-

```
myDevice.writeRegisters(10, sizeof(data), data)
```

You are encouraged to use this technique as it means that if you decide, later on, to write 4 bytes then you can just change the 'data' definition to:

```
uint8_t data[] = { 1, 2, 45, 16 };
```

and nothing else needs to change.

## writeRegister
Performs a complete communication session with a device to write to a value to a single register.

Returns TRUE if successful or FALSE if there was an error.

### Syntax

*i2cdevice*.writeRegister(reg,value)
Where *i2cdevice* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'reg' | The register number. |
| uint8_t | 'value' | The value to be written. |

### Returns

boolean

### Note

This is the same as the following **pseudo** code:

```
// Connect in write mode
i2cStart( WRITE );
// Write the start register
i2cPut( reg );
// Write the data value
i2cPut( value );
// Hang up
i2cStop();
```

### Example

Assuming you have created a 'Generic I2C Device' in Project Designer called myDevice and you want to set register 10 to the value 56 then you would use:

```
if( myDevice.writeRegister(10, 56) ){
    // ok
}else{
    // Communication failed
}
```

## send

Performs a complete communication session with a device to write a number of bytes.

Returns TRUE if successful or FALSE if there was an error.

### Syntax

*i2cdevice*.send(writeLen,writeData)
*i2cdevice*.send(prefixLen,prefixData,writeLen,writeData)
Where *i2cdevice* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| size_t | 'writeLen' | The number of bytes to be written. |
| uint8_t * | 'writeData' | The memory address of the data to be written. |
| size_t | 'prefixLen' | The number of prefix bytes to be written. |
| uint8_t * | 'prefixData' | The memory address of the prefix data to be written. |

### Returns

boolean

### Note

This is the same as the following **pseudo** code:

```
// Connect in write mode
i2cStart( WRITE );
// Write the prefix values
for(size_t i=0; i<prefixLen; i++){
    i2cPut(prefixData[prefixLen]);
}
// Write the data values
for(size_t i=0; i<writeLen; i++){
    i2cPut(writeData[writeLen]);
}
// Hang up
i2cStop();
```

## transfer

Performs a complete communication with a slave device to write a number of bytes and then read a number of bytes.

The return value specifies if it was successful (TRUE) or not (FALSE).

### Syntax

*i2cdevice*.transfer(writeLen,writeData,responseLen,response)
Where *i2cdevice* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| size_t | 'writeLen' | The number of bytes to be written. |
| uint8_t * | 'writeData' | The memory address of the data to be written. |

| Type | Name | Description |
|---|---|---|
| size_t | 'responseLen' | The number of bytes to be received. |
| uint8_t * | 'response' | The memory address to store the returned the values. |

### Returns
boolean

### Note
This is the same as the following **pseudo** code:

```
// Connect in write mode
i2cStart( WRITE );
// Write the data
for(int i=0; i<writeLen; i++){
    i2cPut( writeData[writeLen] );
}
// Connect in read mode
i2cStart( READ );
// Read response
for(int i=0; i<responseLen; i++){
    response[i] = i2cGet();
}
// Hang up
i2cStop();
```

## Class: SpiBus
## Defined in: _spi_common.h

An SPI bus provides the following functions:

| Function Summary | |
|---|---|
| SPI_CLOCK | getClockPrescaler<br>Get the current prescale value used to set the bus speed. |
| | setClockPrescaler<br>Set the prescaler value used to control the bus speed. |

## Function Details

### getClockPrescaler

Get the current prescale value used to set the bus speed.

#### Syntax

*spibus*.getClockPrescaler()
Where *spibus* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

SPI_CLOCK

---

### setClockPrescaler

Set the prescaler value used to control the bus speed.

#### Syntax

*spibus*.setClockPrescaler(prescale)
Where *spibus* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| SPI_CLOCK | 'prescale' | The required prescaler value to be set. See "*SPI_CLOCK*" (see page ) for possible values. |

#### Returns

None

**Note**

This only has an effect with a hardware SPI bus - for a software SPI bus it will have no effect on the actual timings.

## Class: SpiDevice
## Defined in: _spi_common.h

An SPI device provides the following functions:

| Function Summary | |
|---|---|
| | select<br>Selects the device as the target you want to communicate with,Only one device can be selected at a time and WebbotLib will make sure that when one device is selected that all other devices on the same SPI bus are de-selected first. |
| | write<br>Send one or more bytes to the device. |
| `uint8_t` | xfer<br>Performs a transfer by sending the specified data and returning the response, |
| `uint8_t` | get<br>Reads a byte from the device. |
| | read<br>Read one or more bytes from the device. |

### Function Details

### select

Selects the device as the target you want to communicate with,

Only one device can be selected at a time and WebbotLib will make sure that when one device is selected that all other devices on the same SPI bus are de-selected first.

All of the calls that send, receive or transfer data will automatically call this function to select the device.

#### Syntax

*spidevice*.select(active)
Where *spidevice* is the name you have given to the device in Project Designer.

#### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'active' | TRUE to select the device or FALSE to un-select. |

### Returns
None

---

## write
Send one or more bytes to the device.

### Syntax
*spidevice*.write(byte)
*spidevice*.write(src,size)
Where *spidevice* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'byte' | The byte of data |
| void * | 'src' | The start address in memory of the data to be written to the device. |
| size_t | 'size' | The number of bytes to be transferred |

### Returns
None

---

## xfer
Performs a transfer by sending the specified data and returning the response,

### Syntax
*spidevice*.xfer(byte)
Where *spidevice* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'byte' | The byte of data |

### Returns
uint8_t

---

## get
Reads a byte from the device.

### Syntax
*spidevice*.get()

Where *spidevice* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

uint8_t

## read

Read one or more bytes from the device.

### Syntax

*spidevice*.read(dst,size)
Where *spidevice* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| void * | 'dst' | The start address in memory to store the data read from the device. |
| size_t | 'size' | The number of bytes to be transferred |

### Returns

None

## Class: RTC
## Defined in: RTC/_rtc_common.h

All real time clocks support the following functions:

| Function Summary | |
|---|---|
| | dump<br>Dump the current clock values to the standard output device. |
| | dumpTo<br>Dump the value of the clock to the specified output stream. |
| `boolean` | set<br>Attempt to set the clock to the specified date and time. |
| `DATE_TIME*` | read<br>Read the real time clock and return the values. |

### Function Details

### dump

Dump the current clock values to the standard output device.

#### Syntax

*rtc*.dump()
Where *rtc* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

---

### dumpTo

Dump the value of the clock to the specified output stream.

#### Syntax

*rtc*.dumpTo(dest)
Where *rtc* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| FILE * | 'dest' | The destination to dump the sensor info to. This can be any stream. |

### Returns
None

### See Also
dump

## set
Attempt to set the clock to the specified date and time.

This will return FALSE if the clock is not accessible.

### Syntax
*rtc*.set(year,month,day,hour,minute,second)
Where *rtc* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint16_t | 'year' | The year value |
| uint8_t | 'month' | The month value in the range 1 to 12. |
| uint8_t | 'day' | The day of the month value in the range 1 to 31 - or less - depending on the month. |
| uint8_t | 'hour' | The hour value in the range 0 to 23. |
| uint8_t | 'minute' | The minute value in the range 0 to 59. |
| uint8_t | 'second' | The second value in the range 0 to 59. |

### Returns
boolean

### Note
It is up to you to make sure that date is valid. For example: trying to set the date to 31 February will result in an unkown result.

## read
Read the real time clock and return the values.

The return value may be NULL if the clock is not accessible (ie not plugged in) and so you must always test the returned value in your own code.

### Syntax

*rtc*.read()
Where *rtc* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

DATE_TIME*

### Example

Assuming your RTC is called 'myClock' then:

```
const DATE_TIME* dt = myClock.read();
if(dt != NULL){
uint16_t year = dt->year; // ie 2011
uint8_t month = dt->month; // 1 to 12
uint8_t date = dt->date; // 1 to 31
uint8_t hours = dt->hours; // 0 to 23
uint8_t minutes = dt->minutes; // 0 to 59
uint8_t seconds = dt->seconds; // 0 to 59
}
```

## Class: OneWireDevice
## Defined in: oneWireBus.h

All one wire devices provide the following functions:

| Function Summary | |
|---|---|
| `boolean` | exists<br>Searches the bus to discover whether this device is plugged in. |
| `boolean` | found<br>Tests whether this device was found when the bus was initialised. |
| | dumpROM<br>Dumps the complete ROM ID of the device to the specified output stream. |

## Function Details

### exists

Searches the bus to discover whether this device is plugged in.

Returns TRUE if it does or FALSE if it doesn't.

#### Syntax

*onewiredevice*.exists()
Where *onewiredevice* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

boolean

---

### found

Tests whether this device was found when the bus was initialised.

Returns TRUE if it was or FALSE if not.

#### Syntax

*onewiredevice*.found()
Where *onewiredevice* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

## dumpROM
Dumps the complete ROM ID of the device to the specified output stream.

### Syntax
*onewiredevice*.dumpROM(f)
Where *onewiredevice* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| FILE * | 'f' | The output stream. |

### Returns
None

## Class: OneWireBus
## Defined in: oneWireBus.h

A one wire bus provides the following functions (but you should only need them if you are trying to support a device that isn't directly supported by WebbotLib):

| Function Summary | |
|---|---|
| | applyPower<br>Often referred to in the specifications as 'strong pull-up' this will drive the signal line high so that parasitic devices can obtain power from the bus. |
| boolean | readBit<br>Reads a single bit from the bus and returns TRUE for 1, or FALSE for 0. |
| | writeBit<br>Write a single bit to the bus. |
| | setStandardSpeed<br>Sets the bus to use 'standard' speed. |
| | setOverdriveSpeed<br>Sets the bus to use the faster 'overdrive' speed. |
| boolean | reset<br>Resets the one wire bus and returns the 'presence' bit. |
| | write<br>Writes an 8 bit byte to the bus. |
| uint8_t | read<br>Reads an 8 bit byte from the bus. |
| uint16_t | list<br>Dumps out the ROM IDs of every device that is connected to the bus and returns the total number of devices found. |
| | dumpROM<br>Dumps the ROM ID of the last device found by a call to 'first' or 'next'. |
| boolean | first<br>Find the first device connected to the bus. |

## Function Summary

| | |
|---|---|
| boolean | next<br>Find the next device connected to the bus. |
| uint8_t * | getROM<br>Return the 8 byte ROM ID of the device matched by the last call to first or last. |

## Function Details

### applyPower

Often referred to in the specifications as 'strong pull-up' this will drive the signal line high so that parasitic devices can obtain power from the bus.

This is normally followed by a delay and then resetting the bus.

#### Syntax

*onewirebus*.applyPower()
Where *onewirebus* is the name you have given to the device in Project Designer.

#### Parameters

None

#### Returns

None

#### Note

This will actually set the bus signal I/O pin high so that the current is being supplied by your micro controller pin. Since most one wire devices only require a small amount of current then this is satisfactory. However: if you are trying to code a new device that requires more current than the AVR is capable of providing (typically 30mA) then DON'T USE THIS METHOD as it will blow your AVR pin or worse.

Make sure you know what you are doing - if you use this call and a device then pulls the bus line low then you will create a short circuit and something will go bang!

---

### readBit

Reads a single bit from the bus and returns TRUE for 1, or FALSE for 0.

#### Syntax

*onewirebus*.readBit()
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

---

## writeBit
Write a single bit to the bus.

### Syntax
*onewirebus*.writeBit(bit)
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| boolean | 'bit' | TRUE for 1, or FALSE for 0. |

### Returns
None

---

## setStandardSpeed
Sets the bus to use 'standard' speed.

### Syntax
*onewirebus*.setStandardSpeed()
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

---

## setOverdriveSpeed
Sets the bus to use the faster 'overdrive' speed.

### Syntax
*onewirebus*.setOverdriveSpeed()
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
None

### Note
You will need to make sure that all of the devices you are using on the bus support overdrive.

---

## reset
Resets the one wire bus and returns the 'presence' bit.

### Syntax
*onewirebus*.reset()
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters
None

### Returns
boolean

---

## write
Writes an 8 bit byte to the bus.

### Syntax
*onewirebus*.write(wval)
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| uint8_t | 'wval' | The byte to be written to the bus. |

### Returns
None

---

## read
Reads an 8 bit byte from the bus.

### Syntax

*onewirebus*.read()
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

uint8_t

---

## list

Dumps out the ROM IDs of every device that is connected to the bus and returns the total number of devices found.

This is useful when trying to write code for new devices as you can, at least, make sure that they are recognised.

### Syntax

*onewirebus*.list()
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

uint16_t

---

## dumpROM

Dumps the ROM ID of the last device found by a call to 'first' or 'next'.

### Syntax

*onewirebus*.dumpROM(f)
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters

| Type | Name | Description |
|---|---|---|
| FILE * | 'f' | The output stream to dump information to. |

### Returns

None

---

## first

Find the first device connected to the bus. Return TRUE if there is one or FALSE if there are no devices at all.

### Syntax

*onewirebus*.first()
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

boolean

---

## next

Find the next device connected to the bus. Return TRUE if there is one or FALSE if there are no more.

### Syntax

*onewirebus*.next()
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

boolean

---

## getROM

Return the 8 byte ROM ID of the device matched by the last call to first or last.

### Syntax

*onewirebus*.getROM()
Where *onewirebus* is the name you have given to the device in Project Designer.

### Parameters

None

### Returns

uint8_t *

---

## Class: Widget
## Defined in: Displays/_gdisplay_common.h

All widgets provide the following functions:

| Function Summary | |
|---|---|
| | setBackgroundColour<br>Set the background colour for the widget. |
| | draw<br>Draws the main body of the widget. |
| | update<br>Draws the variable part of the widget (ie the value it is showing). |

### Function Details

### setBackgroundColour

Set the background colour for the widget.

#### Syntax

*widget*.setBackgroundColour(bkgndColour)

#### Parameters

| Type | Name | Description |
|---|---|---|
| Color | 'bkgndColour' | The background colour of the widget rectangle |

#### Returns

None

#### Note

You need to call the 'draw' method to redraw the widget with the new settings.

### draw

Draws the main body of the widget.

Depending upon the complexity of the widget this can be quite slow so you normally only need to call this once - from appInitSoftware say.

The only time you would need to call it again is if you change any of the properties of the widget at runtime - such as the background colour. Calling this function every time around the main loop will cause it to flash and slow down your program.

### Syntax
*widget*.draw()

### Parameters
None

### Returns
None

### See Also
update

---

## update
Draws the variable part of the widget (ie the value it is showing).

This function should be called frequently as it will change the widget to reflect the current display value - e.g. for a DialWidget it will move the needle to the new position.

Each widget is optimised so that this function only changes the display if it needs to.

### Syntax
*widget*.update()

### Parameters
None

### Returns
None

---

## Class: DialWidget
## Defined in: Displays/_gdisplay_common.h

A dial widget provides the following functions:

| Function Summary | |
|---|---|
| | DialWidget<br>Constructs a new dial widget. |
| | setNeedleColour<br>Change the needle colour at runtime. |
| | setDialColour<br>Change the colour of the face of the dial. |
| | setTickColour<br>Change the colour of the tick marks on the dial. |
| double | getValue<br>Returns the current value of the dial widget. |
| | setValue<br>Sets the new value for the dial widget. |

### Function Details

### DialWidget

Constructs a new dial widget.

#### Syntax

DialWidget(display,x,y,w,h,padding,bkgndColour,needleColour,dialColour,tickColour,minAngle,maxAngle,minVal,maxVal,tickEvery)

#### Parameters

| Type | Name | Description |
|---|---|---|
| GraphicDisplay | 'display' | The graphic display to show this widget on. |
| PIXEL | 'x' | The number of pixels from the left side of the display to the left side of the widget. |
| PIXEL | 'y' | The number of pixels from the top of the display to the top of the widget. |

| Type | Name | Description |
|---|---|---|
| PIXEL | 'w' | The width of the widget in pixels. |
| PIXEL | 'h' | The height of the widget in pixels. |
| PIXEL | 'padding' | The size, in pixels, of the border around the widget that will be shown in the background colour. |
| Color | 'bkgndColour' | The background colour of the widget rectangle |
| Color | 'needleColour' | The colour of the needle. |
| Color | 'dialColour' | The colour of the face of the dial. |
| Color | 'tickColour' | The colour of the tick marks around the edge of the dial. If this is the same as the dial colour then the tick marks will not be visible. |
| int16_t | 'minAngle' | The angle, in degrees, of the needle that represents the minimum value. |
| int16_t | 'maxAngle' | The angle, in degrees, of the needle that represents the maximum value. |
| double | 'minVal' | The minimum value that can be shown by the widget. |
| double | 'maxVal' | The maximum value that can be shown by the widget. |
| double | 'tickEvery' | Controls how many tick marks are shown on the dial. The total number of tick marks will be ((maxValue - minValue)/tickEvery)+1. A value of 0 will result in no tick marks. |

### Returns
None

## setNeedleColour
Change the needle colour at runtime.

For example: if your dial represents a 'fuel gauge' then you may want the needle to change to red when the fuel level is getting low.

### Syntax
*dialwidget*.setNeedleColour(needleColour)

### Parameters

| Type | Name | Description |
|---|---|---|
| Color | 'needleColour' | The colour of the needle. |

### Returns
None

### Note
The needle will be shown in the new colour on the next call to 'update'.

---

## setDialColour
Change the colour of the face of the dial.

### Syntax
*dialwidget*.setDialColour(dialColour)

### Parameters

| Type | Name | Description |
|---|---|---|
| Color | 'dialColour' | The colour of the face of the dial. |

### Returns
None

### Note
You will need to call the 'draw' function to redraw the widget with the new colour. This is not done automatically as you may be changing various properties at once.

---

## setTickColour
Change the colour of the tick marks on the dial.

### Syntax
*dialwidget*.setTickColour(tickColour)

### Parameters

| Type | Name | Description |
|---|---|---|
| Color | 'tickColour' | The colour of the tick marks around the edge of the dial. If this is the same as the dial colour then the tick marks will not be visible. |

### Returns

None

### Note

You will need to call the 'draw' function to redraw the widget with the new colour. This is not done automatically as you may be changing various properties at once.

## getValue

Returns the current value of the dial widget.

### Syntax

*dialwidget*.getValue()

### Parameters

None

### Returns

double

## setValue

Sets the new value for the dial widget.

### Syntax

*dialwidget*.setValue(value)

### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'value' | The new value for the needle. |

### Returns

None

## Class: NumberWidget
## Defined in: Displays/_gdisplay_common.h

A number widget provides the following functions:

| Function Summary | |
|---|---|
| | NumberWidget<br>Constructs a new number widget. |
| | setDigitColour<br>Change the digit foreground colour at runtime. |
| | setDisplayColour<br>Change the digit background colour at runtime. |
| `double` | getValue<br>Returns the current value of the number widget. |
| | setValue<br>Sets the new value for the number widget. |

## Function Details

### NumberWidget

Constructs a new number widget.

#### Syntax

NumberWidget(display,x,y,w,h,padding,bkgndColour,digitColour,displayColour,numDigits,numDecimals,minVal,maxVal)

#### Parameters

| Type | Name | Description |
|---|---|---|
| GraphicDisplay | 'display' | The graphic display to show this widget on. |
| PIXEL | 'x' | The number of pixels from the left side of the display to the left side of the widget. |
| PIXEL | 'y' | The number of pixels from the top of the display to the top of the widget. |
| PIXEL | 'w' | The width of the widget in pixels. |
| PIXEL | 'h' | The height of the widget in pixels. |

| Type | Name | Description |
|---|---|---|
| PIXEL | 'padding' | The size, in pixels, of the border around the widget that will be shown in the background colour. |
| Color | 'bkgndColour' | The background colour of the widget rectangle |
| Color | 'digitColour' | The foreground colour of the numerical digits. |
| Color | 'displayColour' | The background colour of the numerical digits. |
| uint8_t | 'numDigits' | The number of digits to be shown to the left of the decimal point. |
| uint8_t | 'numDecimals' | The number of digits to be shown to the right of the decimal point. |
| double | 'minVal' | The minimum value that can be shown by the widget. |
| double | 'maxVal' | The maximum value that can be shown by the widget. |

### Returns
None

## setDigitColour
Change the digit foreground colour at runtime.

### Syntax
*numberwidget*.setDigitColour(digitColour)

### Parameters

| Type | Name | Description |
|---|---|---|
| Color | 'digitColour' | The foreground colour of the numerical digits. |

### Returns
None

### Note
The digits will be shown in the new colour on the next call to 'update'.

## setDisplayColour

Change the digit background colour at runtime.

### Syntax

*numberwidget*.setDisplayColour(displayColour)

### Parameters

| Type | Name | Description |
|---|---|---|
| Color | 'displayColour' | The background colour of the numerical digits. |

### Returns

None

### Note

You will need to call 'draw' to redraw the widget with the new colour settings.

## getValue

Returns the current value of the number widget.

### Syntax

*numberwidget*.getValue()

### Parameters

None

### Returns

double

## setValue

Sets the new value for the number widget.

### Syntax

*numberwidget*.setValue(value)

### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'value' | The new value to be displayed. |

**Returns**

None

## Class: ThermometerWidget
## Defined in: Displays/_gdisplay_common.h

A thermometer widget provides the following functions:

| Function Summary | |
|---|---|
| | ThermometerWidget<br>Constructs a new thermometer widget. |
| | setMercuryColour<br>Changes the colour of the mercury in the thermometer. |
| | setOutlineColour<br>Changes the colour used to draw the outline of the thermometer. |
| | setTickColour<br>Change the colour of the tick marks on the thermometer. |
| `double` | getValue<br>Returns the current value of the thermometer widget. |
| | setValue<br>Sets the new value for the thermometer widget. |

### Function Details

#### ThermometerWidget

Constructs a new thermometer widget.

#### Syntax

ThermometerWidget(display,x,y,w,h,padding,bkgndColour,mercuryColour,outlineColour,tickColour,minVal,maxVal,tickEvery)

#### Parameters

| Type | Name | Description |
|---|---|---|
| GraphicDisplay | 'display' | The graphic display to show this widget on. |
| PIXEL | 'x' | The number of pixels from the left side of the display to the left side of the widget. |
| PIXEL | 'y' | The number of pixels from the top of the display to the top of the widget. |

| Type | Name | Description |
|---|---|---|
| PIXEL | 'w' | The width of the widget in pixels. |
| PIXEL | 'h' | The height of the widget in pixels. |
| PIXEL | 'padding' | The size, in pixels, of the border around the widget that will be shown in the background colour. |
| Color | 'bkgndColour' | The background colour of the widget rectangle |
| Color | 'mercuryColour' | The colour of the mercury in the thermometer. |
| Color | 'outlineColour' | The colour used to draw the outline of the thermometer. |
| Color | 'tickColour' | The colour of the tick marks around the edge of the thermometer. |
| double | 'minVal' | The minimum value that can be shown by the widget. |
| double | 'maxVal' | The maximum value that can be shown by the widget. |
| double | 'tickEvery' | Controls how many tick marks are shown on the thermometer. The total number of tick marks will be ((maxValue - minValue)/tickEvery)+1.<br><br>A value of 0 will result in no tick marks. |

### Returns
None

## setMercuryColour
Changes the colour of the mercury in the thermometer.

### Syntax
*thermometerwidget*.setMercuryColour(mercuryColour)

### Parameters

| Type | Name | Description |
|---|---|---|
| Color | 'mercuryColour' | The colour of the mercury in the thermometer. |

### Returns
None

### Note
You must call the 'draw' function to redisplay the widget using the new colour. This is not done automatically as you may be making several changes at the same time.

## setOutlineColour
Changes the colour used to draw the outline of the thermometer.

### Syntax
*thermometerwidget*.setOutlineColour(outlineColour)

### Parameters
| Type | Name | Description |
|---|---|---|
| Color | 'outlineColour' | The colour used to draw the outline of the thermometer. |

### Returns
None

### Note
You must call the 'draw' function to redisplay the widget using the new colour. This is not done automatically as you may be making several changes at the same time.

## setTickColour
Change the colour of the tick marks on the thermometer.

### Syntax
*thermometerwidget*.setTickColour(tickColour)

### Parameters
| Type | Name | Description |
|---|---|---|
| Color | 'tickColour' | The colour of the tick marks around the edge of the thermometer. |

### Returns
None

### Note
You will need to call the 'draw' function to redraw the widget with the new colour. This is not done automatically as you may be changing various properties at once.

## getValue

Returns the current value of the thermometer widget.

### Syntax

*thermometerwidget*.getValue()

### Parameters

None

### Returns

double

## setValue

Sets the new value for the thermometer widget.

### Syntax

*thermometerwidget*.setValue(value)

### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'value' | The new value to be displayed. |

### Returns

None

## Class: Vector2D
## Defined in: Maths/Vector2D.h

Two dimensional vectors having x and ycomponents.

Vectors can use many of the operators that you can use with more basic types. For example:

Assignment:

```
Vector2D vec1(1,2);
Vector2D vec2();
vec2 = vec1; // Set vec2 to the same values as used by vec1
```

Test for equality or inequality:

```
if( vec1 == vec2){
    // Both vectors are identical
}
if( vec1 != vec2){
    // The vectors are not identical
}
```

Addition:

```
vec3 = vec1 + vec2; // Add two vectors and store in a third
vec2 += vec1; // vec2 = vec2 + vec1
```

Subtraction:

```
vec3 = vec1 - vec2; // Subtract two vectors and store in a third
vec2 -= vec1; // vec2 = vec2 - vec1
```

Scaling:

```
vec2 = vec1 * 3.5; // Multiply each component of vec1 and store in vec2
vec2 *= 5; // Multiply each component of vec2 and store back in vec2
```

| Function Summary | |
|---|---|
| | Vector2D<br>Constructs a new vector object. |
| `double` | length<br>Returns the magnitude of the vector. |
| `double` | length2<br>Returns the magnitude squared of the vector. |
| | set<br>Set the x,y components of a vector to new values. |

## Function Summary

| | |
|---|---|
| | normalise<br>Scale the vector so that it has a magnitude of 1. |
| double | dot<br>Returns the dot product of this vector with another specified vector. |
| | dump<br>Print the contents of the vector to the specified output stream. |
| double | radians<br>Returns the angle, in radians, between this vector and another one. |

## Function Details

### Vector2D

Constructs a new vector object.

#### Syntax

Vector2D()
Vector2D(x,y)
Vector2D(other)

#### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'x' | The x value of the vector. |
| double | 'y' | The y value of the vector. |
| Vector2D | 'other' | Specifies another Vector2D. |

#### Returns

None

#### Note

If no parameters are specified then the vector has the x and y values set to 0.

### Example

```
Vector2D vec0();
Vector2D vec1(1,2);
Vector2D vec2(vec1);
```

'vec0' will be 0,0.

'vec1' will be 1,2.

'vec2' will copy the current values from 'vec1' and so will also be 1,2.

---

## length

Returns the magnitude of the vector.

### Syntax

*vector2d*.length()

### Parameters

None

### Returns

double

---

## length2

Returns the magnitude squared of the vector.

### Syntax

*vector2d*.length2()

### Parameters

None

### Returns

double

---

## set

Set the x,y components of a vector to new values.

### Syntax

*vector2d*.set(x,y)

### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'x' | The x value of the vector. |
| double | 'y' | The y value of the vector. |

### Returns
None

## normalise
Scale the vector so that it has a magnitude of 1.

### Syntax
*vector2d*.normalise()

### Parameters
None

### Returns
None

## dot
Returns the dot product of this vector with another specified vector.

### Syntax
*vector2d*.dot(other)

### Parameters

| Type | Name | Description |
|---|---|---|
| Vector2D | 'other' | Specifies another Vector2D. |

### Returns
double

## dump
Print the contents of the vector to the specified output stream.

### Syntax
*vector2d*.dump(stream)

*vector2d*.dump(stream,decPlaces)

### Parameters

| Type | Name | Description |
|---|---|---|
| Stream | 'stream' | The output stream. |
| uint8_t | 'decPlaces' | The number of decimal places to display in the output. If not specified it will default to 2. |

### Returns

None

## radians

Returns the angle, in radians, between this vector and another one.

### Syntax

*vector2d*.radians(other)

### Parameters

| Type | Name | Description |
|---|---|---|
| Vector2D | 'other' | Specifies another Vector2D. |

### Returns

double

### Note

The returned value is between 0 and PI. To convert it into degrees then multiply by (180.0 / PI);

## Class: Vector3D
## Defined in: Maths/Vector3D.h

Three dimensional vectors having x, y and z components.

Vectors can use many of the operators that you can use with more basic types. For example:

Assignment:

```
Vector3D vec1(1,2,3);
Vector3D vec2();
vec2 = vec1; // Set vec2 to the same values as used by vec1
```

Test for equality or inequality:

```
if( vec1 == vec2){
    // Both vectors are identical
}
if( vec1 != vec2){
    // The vectors are not identical
}
```

Addition:

```
vec3 = vec1 + vec2; // Add two vectors and store in a third
vec2 += vec1; // vec2 = vec2 + vec1
```

Subtraction:

```
vec3 = vec1 - vec2; // Subtract two vectors and store in a third
vec2 -= vec1; // vec2 = vec2 - vec1
```

Scaling:

```
vec2 = vec1 * 3.5; // Multiply each component of vec1 and store in vec2
vec2 *= 5; // Multiply each component of vec2 and store back in vec2
```

| **Function Summary** | |
|---|---|
| | Vector3D<br>Constructs a new vector object. |
| `double` | length<br>Returns the magnitude of the vector. |
| `double` | length2<br>Returns the magnitude squared of the vector. |
| | set<br>Set the x,y,z components of a vector to new values. |

## Function Summary

| | |
|---|---|
| | normalise<br>Scale the vector so that it has a magnitude of 1. |
| double | dot<br>Returns the dot product of this vector with another specified vector. |
| Vector3D | cross<br>Returns the cross product of this vector with another vector. |
| | dump<br>Print the contents of the vector to the specified output stream. |
| double | radians<br>Returns the angle, in radians, between this vector and another one. |

## Function Details

### Vector3D

Constructs a new vector object.

#### Syntax

Vector3D()
Vector3D(x,y,z)
Vector3D(other)

#### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'x' | The x value of the vector. |
| double | 'y' | The y value of the vector. |
| double | 'z' | The z value of the vector. |
| Vector3D | 'other' | Specifies another Vector3D. |

#### Returns

None

#### Note

If no parameters are specified then the vector has the x,y and z values set to 0.

### Example

```
Vector3D vec0();
Vector3D vec1(1,2,3);
Vector3D vec2(vec1);
```

'vec0' will be 0,0,0.

'vec1' will be 1,2,3.

'vec2' will copy the current values from 'vec1' and so will also be 1,2,3.

---

## length

Returns the magnitude of the vector.

### Syntax

*vector3d*.length()

### Parameters

None

### Returns

double

---

## length2

Returns the magnitude squared of the vector.

### Syntax

*vector3d*.length2()

### Parameters

None

### Returns

double

---

## set

Set the x,y,z components of a vector to new values.

### Syntax

*vector3d*.set(x,y,z)

### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'x' | The x value of the vector. |
| double | 'y' | The y value of the vector. |
| double | 'z' | The z value of the vector. |

### Returns

None

---

## normalise

Scale the vector so that it has a magnitude of 1.

### Syntax

*vector3d*.normalise()

### Parameters

None

### Returns

None

---

## dot

Returns the dot product of this vector with another specified vector.

### Syntax

*vector3d*.dot(other)

### Parameters

| Type | Name | Description |
|---|---|---|
| Vector3D | 'other' | Specifies another Vector3D. |

### Returns

double

---

## cross

Returns the cross product of this vector with another vector. The returned vector is at right angles to the other two.

### Syntax

*vector3d*.cross(other)

### Parameters

| Type | Name | Description |
|---|---|---|
| Vector3D | 'other' | Specifies another Vector3D. |

### Returns

Vector3D

### Example

```
Vector3D vec1(1,0,0);
Vector3D vec2(0,1,0);
Vector3D vec3 = vec1.cross(vec2);
```

## dump

Print the contents of the vector to the specified output stream.

### Syntax

*vector3d*.dump(stream)
*vector3d*.dump(stream,decPlaces)

### Parameters

| Type | Name | Description |
|---|---|---|
| Stream | 'stream' | The output stream. |
| uint8_t | 'decPlaces' | The number of decimal places to display in the output. If not specified it will default to 2. |

### Returns

None

## radians

Returns the angle, in radians, between this vector and another one.

### Syntax

*vector3d*.radians(other)

### Parameters

| Type | Name | Description |
|---|---|---|
| Vector3D | 'other' | Specifies another Vector3D. |

### Returns

double

### Note

The returned value is between 0 and PI. To convert it into degrees then multiply by (180.0 / PI);

## Class: Matrix3D
## Defined in: Maths/Matrix3D.h

A 3 x 3 matrix whose members are called:

```
m00 m01 m02
m10 m11 m12
m20 m21 m22
```

These members can be accessed directly ie:

```
Matrix3D matrix1;
matrix1.unit(); // Set matrix1 to 1,0,0; 0,1,0; 0,0,1
double val = matrix1.m00; // This will be 1
```

The following standard operators can be used with Matrix3D variables: '+=', '-=', '*=' to perform addition, subtraction and multiplication of matrices.

| Function Summary | |
|---|---|
| | Matrix3D<br>Constructs a new matrix variable. |
| | set<br>Changes the contents of an existing matrix to the specified values, or copies the values from another matrix. |
| | zero<br>Sets all the members of the matrix to zero. |
| | unit<br>Sets the matrix to a scale factor of 1 |
| `double` | determinant<br>Returns the determinant of the matrix. |
| | setRotateX<br>Initialises the matrix to represent a rotation around the X axis. |
| | setRotateY<br>Initialises the matrix to represent a rotation around the Y axis. |
| | setRotateZ<br>Initialises the matrix to represent a rotation around the Z axis. |

## Function Summary

| | |
|---|---|
| | setScale<br>Initialises the matrix to represent a scaling factor. |
| `Vector3D` | transform<br>Multiplies the specified vector by the matrix to return a new transformed vector. |
| | dump<br>Print the elements of the matrix to the specified output stream. |

## Function Details

### Matrix3D

Constructs a new matrix variable.

#### Syntax

Matrix3D()
Matrix3D(m00,m01,m02,m10,m11,m12,m20,m21,m22)
Matrix3D(other)

#### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'm00' | The value of row 0, column 0 |
| double | 'm01' | The value of row 0, column 1 |
| double | 'm02' | The value of row 0, column 2 |
| double | 'm10' | The value of row 1, column 0 |
| double | 'm11' | The value of row 1, column 1 |
| double | 'm12' | The value of row 1, column 2 |
| double | 'm20' | The value of row 2, column 0 |
| double | 'm21' | The value of row 2, column 1 |
| double | 'm22' | The value of row 2, column 2 |
| Matrix3D | 'other' | Specifies another Matrix3D |

### Returns
None

### Note
If no parameters are specified then each element of the matrix is set to zero.

### Example
To create a matrix that is set to zero:

```
Matrix3D mat();
```

To create a matrix with known values:

```
Matrix3D mat(1,2,3, 4,5,6, 7,8,9);
```

To create a matrix which is initialised to the same values as another matrix:

```
Matrix3D mat(matrix);
```

---

## set
Changes the contents of an existing matrix to the specified values, or copies the values from another matrix.

### Syntax
*matrix3d*.set(m00,m01,m02,m10,m11,m12,m20,m21,m22)
*matrix3d*.set(other)

### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'm00' | The value of row 0, column 0 |
| double | 'm01' | The value of row 0, column 1 |
| double | 'm02' | The value of row 0, column 2 |
| double | 'm10' | The value of row 1, column 0 |
| double | 'm11' | The value of row 1, column 1 |
| double | 'm12' | The value of row 1, column 2 |
| double | 'm20' | The value of row 2, column 0 |
| double | 'm21' | The value of row 2, column 1 |

| Type | Name | Description |
|---|---|---|
| double | 'm22' | The value of row 2, column 2 |
| Matrix3D | 'other' | Specifies another Matrix3D |

**Returns**

None

## zero

Sets all the members of the matrix to zero.

**Syntax**

*matrix3d*.zero()

**Parameters**

None

**Returns**

None

## unit

Sets the matrix to a scale factor of 1

```
1,0,0
0,1,0
0,0,1
```

**Syntax**

*matrix3d*.unit()

**Parameters**

None

**Returns**

None

## determinant

Returns the determinant of the matrix.

### Syntax
*matrix3d*.determinant()

### Parameters
None

### Returns
double

---

## setRotateX
Initialises the matrix to represent a rotation around the X axis.

### Syntax
*matrix3d*.setRotateX(radians)

### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'radians' | The rotation in radians |

### Returns
None

---

## setRotateY
Initialises the matrix to represent a rotation around the Y axis.

### Syntax
*matrix3d*.setRotateY(radians)

### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'radians' | The rotation in radians |

### Returns
None

---

## setRotateZ
Initialises the matrix to represent a rotation around the Z axis.

### Syntax

*matrix3d*.setRotateZ(radians)

### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'radians' | The rotation in radians |

### Returns

None

## setScale

Initialises the matrix to represent a scaling factor.

### Syntax

*matrix3d*.setScale(scale)

### Parameters

| Type | Name | Description |
|---|---|---|
| double | 'scale' | The scale factor |

### Returns

None

### Note

For a given scale factor 's' this will change the matrix to:

```
s,0,0
0,s,0
0,0,s
```

## transform

Multiplies the specified vector by the matrix to return a new transformed vector.

### Syntax

*matrix3d*.transform(vector)

### Parameters

| Type | Name | Description |
|---|---|---|
| Vector3D | 'vector' | A vector |

### Returns

Vector3D

### Example

Create a vector:

```
Vector3D vecIn( 1,0,0 );
```

Create a matrix that represent a rotation around the X axis:

```
Matrix3D mat;
mat.setRotateX(3.142);
```

Create another vector to hold the result:

```
Vector3D vecOut = mat.transform(vecIn);
```

Alternatively: if you will no longer require the original vecIn vector then you can overwrite it with the result to minimise the number of different variables you are using:

```
vecIn = mat.transform(vecIn);
```

## dump

Print the elements of the matrix to the specified output stream.

### Syntax

*matrix3d*.dump(stream)
*matrix3d*.dump(stream,decPlaces)

### Parameters

| Type | Name | Description |
|---|---|---|
| Stream | 'stream' | The output stream. |
| uint8_t | 'decPlaces' | The number of decimal places to display in the output. If not specified it will default to 2. |

### Returns

None

## Class: DCM
## Defined in: Maths/DCM.h

Implements a Direction Cosine Matrix (DCM) and is only available if you are using C++ with WebbotLib.

This can be used, along with a gyro, accelerometer and either a magnetometer or a compass, as the basis of a control system for airborne vehicles. Many boards (normally referred to as an IMU) already exist which combine these sensors with a microprocessor onto a single board. Examples include the Sparkfun Razor and the Mongoose. The combination of such a board along with this library assists you in off-loading the complex mathematics involved away from your main processor onto the slave board. The slave board can then keep sending out information on roll, pitch and yaw via a UART to the main board.

For the theory on DCM then I suggest reading:
http://gentlenav.googlecode.com/files/DCMDraft2.pdf

This code is based on work by Doug Weibel and Jose Julio but completely re-written for WebbotLib.

### Gyros
To make use of this class you must regularly update the DCM with the gyro readings. By 'regularly' I mean about every 20ms (50 times per second).

This is done by calling gyroDegrees or gyroRadians passing in a "*Vector3D*" (see page 596) that contains the gyro rotations around the x, y, and z axes in degrees or radians respectively.

You may wonder "why can't I just tell the DCM about the gyro device we are using and let WebbotLib do the reading itself?". Well the answer is: each board can be built differently so the concept of the x, y and z axes can vary. Also: how you mount the board onto your vehicle may also change the orientation of the axes. Consequently it is up to you to read the gyro and then decide which readings represent the x, y and z axes and whether or not to multiply each value by -1 to reverse its direction.

This library expects that the X axis is pointing in the direction of travel, the Y axis is pointing to the right and the Z axis is pointing down. Therefore the returned values represent:

- Positive pitch : nose up
- Positive roll : right wing down
- Positive yaw : clockwise rotation

Assuming that you are only using gyro information then the resultant pitch, roll and yaw returned values will accumulate errors and, in a short time, become useless.

## Drift Correction

We therefore introduce the 'driftCorrection' function which takes the values from an accelerometer and a compass to error correct the drift. The accelerometer value vector needs to be created by you by reading the accelerometer and aligning the x,y and z axes so that they represent the same directions as the gyro. The compass reading should be a bearing, in radians, from magnetic north.

Since the 'driftCorrection' function is correcting accumulative errors then you don't need to call it every time you update the gyro readings.

The error feedback loop uses a proportional and integral term for both the yaw and the roll/pitch. The floating point scale factors for these terms are set to default values for you but you can modify them yourself if you need to. The roll and pitch values are called 'm_Kp_ROLLPITCH' and 'm_Ki_ROLLPITCH'. The yaw values are called 'm_Kp_YAW' and 'm_Ki_YAW'.

For example: if you wanted to change the yaw constants and your DCM variable is called 'matrix' then your code would look like this:-

```
// The DCM
DCM matrix;

TICK_COUNT appInitSoftware(TICK_COUNT loopStart){
    // Print the default settings to standard out
    cout.print("Kp="); cout.print(matrix.m_Kp_YAW); cout.println();
    cout.print("Ki="); cout.print(matrix.m_Ki_YAW); cout.println();
    // Change the Yaw settings
    matrix.m_Kp_YAW = 1.5;
    matrix.m_Ki_YAW = 0.00003;
}
```

## Magnetometer

If you are using a magnetometer, rather than a compass, then you will be reading x,y and z values. Once again you may need to flip these to use the same axis orientation as the gyros. To calculate the magnetic heading you will need to process the values like this (assuming your DCM is called 'matrix' and you have stored the adjusted magnetometer readings in 'magnetomVector'):-

```
// Calculate the compass magnetic heading
// taking into account the current pitch and roll
double roll = matrix.getRollRadians();    // Get roll from the DCM
double pitch = matrix.getPitchRadians();  // Get pitch from the DCM

// Calculated cos/sin values once
double cos_roll = cos(roll);
double sin_roll = sin(roll);
double cos_pitch = cos(pitch);
double sin_pitch = sin(pitch);

// Tilt compensated Magnetic field X:
double MAG_X = magnetomVector.x*cos_pitch +
      magnetomVector.y*sin_roll*sin_pitch +
      magnetomVector.z*cos_roll*sin_pitch;

// Tilt compensated Magnetic field Y:
duble MAG_Y = magnetomVector.y*cos_roll - magnetomVector.z*sin_roll;

// Finally we calculate the Magnetic Heading
double magHeadingRadians = atan2(-MAG_Y,MAG_X);
```

Alternatively: if you don't have a compass or a magnetometer then you can just use the current yaw value from the DCM but this means that the yaw value will continue to drift:

```
double magHeadingRadians = matrix.getYawRadians();
```

## Compass
If you are using a compass rather than a magnetometer then make sure that it gives the correct value irrespective of orientation - ie with the compass pointing North does it give you the same value if the compass is placed on its side or upside down?

## Output
The output of the DCM is the current pitch, roll and yaw which can be returned in either radians or degrees.

What you do with this output is up to you! A slave board will normally send the values over a UART to the main board to help it navigate and steer.

## Visuals
There are various GUIs that can be used on computers to visualise the output by sending the roll, pitch and yaw values over a serial port. The Sparkfun Razor board contains a link to some Python code whereas the Mongoose has its own VB display software. You can use any of these so long as you output the information over the UART in the correct format.

| Function Summary | |
|---|---|
| | DCM<br>Create a new direction cosine matrix variable. |
| | reset<br>Reset the DCM to it's initial state. |
| | gyroDegrees<br>Update the DCM with the latest gyro readings in degrees per second. |
| | gyroRadians<br>Update the DCM with the latest gyro readings in radians per second. |
| | driftCorrection<br>Correct any drift errors. |
| double | getPitchRadians<br>Returns the current pitch in radians. |
| double | getPitchDegrees<br>Returns the current pitch in degrees. |
| double | getRollRadians<br>Returns the current roll in radians. |
| double | getRollDegrees<br>Returns the current roll in degrees. |
| double | getYawRadians<br>Returns the current yaw in radians. |
| double | getYawDegrees<br>Returns the current yaw in degrees. |

## Function Details

### DCM

Create a new direction cosine matrix variable.

#### Syntax

DCM()

#### Parameters

None

### Returns
None

### Example
To create a new variable called 'matrix' then:

```
DCM matrix;
```

---

## reset
Reset the DCM to it's initial state.

### Syntax
*dcm*.reset()

### Parameters
None

### Returns
None

### Note
This function is called automatically when a new DCM is created but you may wish to call it yourself if you want to reset it back to the power on state.

---

## gyroDegrees
Update the DCM with the latest gyro readings in degrees per second. This function should be called regularly - around 50 times per second.

### Syntax
*dcm*.gyroDegrees(gyroDeg)

### Parameters

| **Type** | **Name** | **Description** |
|---|---|---|
| Vector3D | 'gyroDeg' | The x, y and z gyro readings in degrees per second. |

### Returns
None

---

## gyroRadians

Update the DCM with the latest gyro readings in radians per second. This function should be called regularly - around 50 times per second.

### Syntax

*dcm*.gyroRadians(gyroRad)

### Parameters

| Type | Name | Description |
|---|---|---|
| Vector3D | 'gyroRad' | The x, y and z gyro readings in radians per second. |

### Returns

None

---

## driftCorrection

Correct any drift errors.

### Syntax

*dcm*.driftCorrection(accelVector,magHeadingRadians)

### Parameters

| Type | Name | Description |
|---|---|---|
| Vector3D | 'accelVector' | The x, y and z accelerometer values in whole G. |
| double | 'magHeadingRadians' | The magnetic compass heading in radians. |

### Returns

None

---

## getPitchRadians

Returns the current pitch in radians.

### Syntax

*dcm*.getPitchRadians()

### Parameters

None

### Returns
double

---

## getPitchDegrees
Returns the current pitch in degrees.

### Syntax
*dcm*.getPitchDegrees()

### Parameters
None

### Returns
double

---

## getRollRadians
Returns the current roll in radians.

### Syntax
*dcm*.getRollRadians()

### Parameters
None

### Returns
double

---

## getRollDegrees
Returns the current roll in degrees.

### Syntax
*dcm*.getRollDegrees()

### Parameters
None

### Returns
double

## getYawRadians

Returns the current yaw in radians.

### Syntax

*dcm*.getYawRadians()

### Parameters

None

### Returns

double

## getYawDegrees

Returns the current yaw in degrees.

### Syntax

*dcm*.getYawDegrees()

### Parameters

None

### Returns

double

## Common Data Types

One of the common problems encountered by newbies, and the more experienced among us to be honest, is to do with choosing suitable data types to store the information we want to hold. Note that this is nothing to do with WebbotLib but is discussed here to help avoid the pitfalls.

The use of unsuitable data types can lead to your program behaving un-expectedly with lots of time wasted in trying to understand what is going on.

What do I mean by a data type? With most micro-controllers the majority of the information you are storing, reading, using in calculations, or having returned from a calculation is some sort of number. Text tends to be read-only in a Robot environment.

There are two fundamental kinds of numbers: real numbers (also called floating point) and integers. Integers are whole numbers (like How many children do you have?) whereas real numbers can have something to the right of the decimal point (like How many dollars and cents have you got?). Note that real number can also be used to hold integers - they just end in '.0'

Since real numbers can store anything (well just about but an over-simplification) then "Can I just use real numbers throughout my program?" Well the quick answer is 'Yes' but the real answer is 'Yes - but don't do it unless you have to!' The reason is partly that real numbers take up more memory and also that micro-controllers don't contain hardware to do calculations with real numbers so some extra software gets automatically added to your code in the build process. This extra software is also slow to run by comparison with integers. Note that the added software overhead is an all or nothing thing - ie if your code doesn't use ANY real numbers then it won't be added, but as soon as you add a single real number to your program then the whole extra code gets added.

The 'C' standard defines two data types for real numbers: 'float' and 'double'. The avr-gcc C compiler treats them both the same so they are inter-changeable. So I could do the following:-

```
float a = 1.27; // a real number
float b = 3.45; // another real number
float c = a * b; // do a real number multiplication and store the result: 4.3815
```

Real numbers aren't the real issue - you either have to use them or you don't. If you've got lots of memory available (such as on the Axon series) then you may not care about the overhead and slower processing.

### What about integers?

Well the first sub-division of integers is to decide whether they can hold negative numbers or not. If they can then they are called 'signed integers' but if they only hold positive numbers then they are called 'unsigned integers'. Once again - we could take the careful approach and assume that everything is 'signed'. But sometimes this doesn't make sense: eg if we wanted to store the distance between two objects then it cannot be negative.

The next division of integers is the range of values they can store. This is important as it dictates how much space is required to store the value. The available values are historically hardware dependent but for AVR chips the choice is: 1 byte, 2 byte, 4 byte. Given that there are 8 bits per byte then these can also be called: 8 bits, 16 bits, 32 bits.

You will see in the description that follows that an 'unsigned 16 bit' number can store a maximum value that is almost twice the value that a 'signed 16 bit' number can hold and that the higher the number of bits (ie the more space required) then the higher the maximum value that can be stored.

The integer data types used within WebbotLib are as follows:

**uint8_t** Requires 1 byte of storage, can hold an unsigned number between 0 and +255

**int8_t** Requires 1 byte of storage, can hold a signed number between -128 and +127

**uint16_t** Requires 2 bytes of storage, can hold an unsigned number between 0 and +65535

**int16_t** Requires 2 bytes of storage, can hold a signed number between -32768 and +32767

**uint32_t** Requires 4 bytes of storage, can hold an unsigned number between 0 and +4294967295

**int32_t** Requires 4 bytes of storage, can hold a signed number between -2147483648 and +2147483647

So when deciding what kind of data type to use then start at the top of the list and work your way down until you find the first entry that matches the values you may need to store.

## Hidden Problems

The problems I eluded to earlier are caused by choosing the wrong data type. The avr-gcc compiler is quite relaxed about what value you store in a variable and there are no runtime checks at all to make sure that the value you are storing in a variable can be stored correctly.

Examples:

```
int8_t myVar = 130;  // oops an int8_t cannot store that value
int8_t var1 = 100;             // yep that stores ok
int8_t var2 = 3;               // yep that stores ok
int8_t result = var1 * var2;   // answer is +300 and that won't fit !!
```

So the quick rule of thumb when you are doing things like add or multiply is to look at the two values that are being worked with and work out what the range of the result could be.

ie if you are multiplying two **int8_t** together then worst cases are -128 x +127 or +127 x +127 giving results of -16,256 and +16,129. Scanning down the table then the best type to use to hold the answer correctly is an **int16_t**

However if you know that the two **int8_t** will only actually store a maximum used range of -10 to +10 then the worst case results are -10 x +10 = -100 and +10 x +10 = +100 in which case you could store the result in an **int8_t.**

# Version History

Don't forget: when upgrading to a new version you should ALWAYS regenerate your project code from Project Designer and then perform a clean build.

To perform a clean build in AvrStudio select the menu option 'Build, Clean'. If you are using the makefile then run 'make clean'.

## C++ Version 2.09

Fixed bug in HD44780 display when using 8 bit data bus

The library now compiles in GCC 4.7.0 although the supplied pre-compiled libraries are still compiled using GCC 4.3.3 (as supplied in WinAVR-20100110). This is because I have found issues in GCC 4.7.0.

## C++ Version 2.08

Updated ADXL345 to allow you to set the refresh rate

Fixed LengthSquared bug in Vector2D and Vector3D

Fixed bug in HMC5883L where Y and Z registers are reversed

Added C++ support for Vector3D and Matrix3D

Added class "*DCM*" (see page 609) to help write AHRS projects

## C++ Version 2.07

Added support for real time clocks: specifically the DS1307 and DS3234.

Added support for Dallas One Wire devices starting with the DS18B20 thermometer.

Added support for the HMC5883L compass.

Added support for the BMP085 pressure sensor (with functions to return altitude).

The Sharp wide angle distance sensors now returns the minimum distance across all 5 beams rather than the average distance.

All pressure sensors now return values in Pa rather than kPa (for better accuracy). So if you are already using one in your project then you will need to adjust your code see "*Pressure*" (see page 222)

Project Designer now has all 3 versions of the Sparkfun Razor board:-

1. "Sparkfun 9DOF Razor" is the original board
2. "Sparkfun 9DOF Razor SEN-10125" where the gyro was changed to use the ITG3200
3. "Sparkfun 9DOF Razor SEN-10736" where the compass was replaced to use the HMC5883L

The makefile generated by Project Designer is now better at recompiling parts of your program when dependent files have changed.

## C++ Version 2.06

All PWM based motor drivers now allow you to specify, in Project Designer, the actual PWM frequency to use.

Fixed an issue whereby rprintf could go wrong under interrupts

The return value of DroneCell function initSMS MUST be checked - otherwise you will get a compiler warning message.

## C++ Version 2.05

C++ support first added.

## Data Type Summary

| Name | Description |
|---|---|
| ACCEL_TYPE | (Based on int16_t).<br>Stores an acceleration in thousandths of the gravitational constant. |
| BAUD_RATE | (Based on uint32_t).<br>A communications baud rate (bits per second). |
| BLACKFIN_RESOLUTION | Can be one of the following values:<br>`BLACKFIN_160_BY_120`<br>`BLACKFIN_320_BY_240`<br>`BLACKFIN_640_BY_480`<br>`BLACKFIN_1280_BY_1024` |
| Blob | Holds the following information about a blob returned by a camera.<br><br>Contains the following fields:-<br>uint16_t *left* :<br>The left X co-ordinate of the blob.<br><br>uint16_t *right* :<br>The right X co-ordinate of the blob.<br><br>uint16_t *top* :<br>The top Y co-ordinate of the blob.<br><br>uint16_t *bottom* :<br>The bottom Y co-ordinate of the blob.<br><br>int16_t *xCenter* :<br>The x offset from the centre of the screen to the centre of the blob.<br><br>int16_t *yCenter* :<br>The y offset from the centre of the screen to the centre of the blob.<br><br>uint32_t *pixels* :<br>The total number of pixels in the blob - ie width x height.<br><br>uint8_t *bin* :<br>The matching colour bin number for this blob. |

| Name | Description |
|---|---|
| boolean | (Based on int8_t).<br>Stores TRUE or FALSE, which can also be typed as true or false. |
| char | Stores a single character. |
| char * | Stores a pointer to one or more characters. This is often used to reference a string which is terminated with a char of 0. |
| Color | Any kind of colour. See "*Color*" (see page 509) |
| COLOR_RGB* | A reference to a colour that is known to be RGB.<br><br>Contains the following fields:-<br>uint8_t *r* :<br>The red component in the range 0 to 255.<br><br>uint8_t *g* :<br>The green component in the range 0 to 255.<br><br>uint8_t *b* :<br>The blue component in the range 0 to 255. |
| COLOR_YUV* | A reference to a colour that is known to be YUV.<br><br>Contains the following fields:-<br>uint8_t *y* :<br>The y component in the range 0 to 255.<br><br>uint8_t *u* :<br>The u component in the range 0 to 255.<br><br>uint8_t *v* :<br>The v component in the range 0 to 255. |
| COLOR* | A reference to a COLOR in any colour space. |
| COMPASS_TYPE | (Based on int16_t).<br>Stores a compass reading in degrees. |
| const char * | Stores a pointer to one or more unmodifiable characters. This is often used to reference a string which is terminated with a char of 0. |

| **Name** | **Description** |
|---|---|
| CURRENT_TYPE | (Based on uint16_t).<br>The number of amps. |
| DATE_TIME* | A reference to a date and time.<br><br>Contains the following fields:-<br>uint16_t *year* :<br>The year.<br><br>uint8_t *month* :<br>The month. From 1 (January) to 12 (December).<br><br>uint8_t *date* :<br>The calendar date in the month in the range 1 to 31, or less, depending on the month and year.<br><br>uint8_t *hours* :<br>The hour in the day based on a 24 hour clock ie 0 to 23.<br><br>uint8_t *minutes* :<br>The number of minutes in the range 0 to 59.<br><br>uint8_t *seconds* :<br>The number of seconds in the range 0 to 59. |
| DISPLAY_COLUMN | (Based on uint8_t).<br>The display column number. From 0 to max-1. |
| DISPLAY_LINE | (Based on uint8_t).<br>The display line number. From 0 to max-1 where 0 is the top. |
| DISTANCE_TYPE | (Based on uint16_t).<br>Stores a distance in whole centimetres. A value of DISTANCE_MAX means that there is nothing in view. |
| double | Stores a floating point number. Unlike most C compilers the avr-gcc compiler treats float and double as the same types. |
| DRIVE_SPEED | (Based on int8_t).<br>The speed for an actuator. This is an integer value in the range -127 to +127. |
| EEPROM_ADDR | (Based on uint32_t). |

| **Name** | **Description** |
|---|---|
| | Holds the byte offset into an EEPROM of where you want to read or write information. |
| ENCODER_TYPE | (Based on int16_t).<br>Measures a number of encoder ticks (pulses). |
| File | Holds a reference to an open disk file. |
| FILE * | A FILE is part of the 'C' standard and is defined in <stdio.h>.<br><br>The word 'FILE' is somewhat misleading - because it does not always represent a file on a disk. Rather it represents a 'thing' that can be 'read from' and/or 'written to'.<br><br>C defines the following:<br><br>stdout - Standard Output - When you generate your code in Project Designer you can decide where this goes to.<br><br>stdin - Standard Input - This is the same device is stdout but is used to read characters from that device.<br><br>stderr - Standard Error - When you generate your code in Project Designer you can decide where this goes to. It can be the same as stdout.<br><br>In C++ these are streams and are also called cout, cin and cerr respectively. |
| FileIterator | A variable that is used to step through the file entries in a given directory. See "*FileIterator*" (see page 548) |
| float | Stores a floating point number. Unlike most C compilers the avr-gcc compiler treats float and double as the same types. |
| GraphicDisplay | A reference to a GraphicalDisplay. |
| GYRO_TYPE | (Based on int16_t).<br>An angular rotation measured in ° per second. |
| HUMIDITY_TYPE | (Based on uint16_t).<br>The humidity as a percentage. |
| int | Stores a signed integer - is the same as "*int16_t*" (see page ) |

| Name | Description |
|---|---|
| int16_t | Stores whole numbers in the range -32,768 to +32,767 and requires two bytes of storage. |
| int32_t | Stores whole numbers in the range -2,147,483,648 to +2,147,483,647 and requires 4 bytes of storage. |
| int8_t | Stores whole numbers in the range -128 to +127 and requires one byte of storage. |
| integer | Represents any kind of integer - signed or unsigned - and of any length.<br><br>Any of the following: int8_t, int16_t, int32_t, uint8_t, uint16_t, uint32_t or int |
| Matrix3D | A reference to a "*Matrix3D*" (see page 602) |
| PERCENTAGE | (Based on uint8_t).<br>A percentage in the range 0 to 100. |
| PIXEL | (Based on int16_t).<br>The x or y co-ordinate of a pixel on a graphical display. The value is signed because graphical display allow values to be specified relative to the current graphics cursor position. |
| PRESSURE_TYPE | (Based on int32_t).<br>Atmospheric pressure in Pascals (Pa) .<br><br>For convesion to other units then see http://en.wikipedia.org/wiki/Pascal_(unit)<br><br>. |
| PS2_BUTTON | (Based on uint16_t).<br>A Sony PS2 controller button.<br><br>Valid values are:<br>`PS2_BTN_SELECT`<br>`PS2_BTN_L3`<br>`PS2_BTN_R3`<br>`PS2_BTN_START`<br>`PS2_DPAD_UP`<br>`PS2_DPAD_RIGHT`<br>`PS2_DPAD_DOWN`<br>`PS2_DPAD_LEFT` |

**Name** **Description**

```
PS2_BTN_L2
PS2_BTN_R2
PS2_BTN_L1
PS2_BTN_R1
PS2_BTN_TRIANGLE
PS2_BTN_CIRCLE
PS2_BTN_X
PS2_BTN_SQUARE
```

PS2_BUTTONS

(Based on PS2_BUTTON).
A bit map of zero or many buttons achieved by OR'ing the individual button values together or 0=None.

PS2_STICK

(Based on int).
The name of a joystick:

```
PS2_STICK_RIGHT_X
PS2_STICK_RIGHT_Y
PS2_STICK_LEFT_X
PS2_STICK_LEFT_Y
// virtual joysticks
PS2_STICK_DPAD_X
PS2_STICK_DPAD_Y
PS2_STICK_SHAPE_X
PS2_STICK_SHAPE_Y
```

real

Any type of real number.

Any of the following: float or double

SEGLED_SEGMENT

(Based on int).
Identifies each of the segments in a segmented LED display.

The segments are called:

```
SEGMENT_A
SEGMENT_B
SEGMENT_C
SEGMENT_D
SEGMENT_E
SEGMENT_F
SEGMENT_G
SEGMENT_H
```

Where the segments correspond to:

```
--a---
|    |
f    b
|    |
```

| **Name** | **Description** |
|---|---|
| | `--g---`<br>`\|    \|`<br>`e    c`<br>`\|    \|`<br>`--d--- h` |
| Sensor | The name of a sensor - as specified in Project Designer. |
| size_t | Can hold a byte length - ie how many bytes. |
| SPI_CLOCK | The clock prescaler used by a hardware SPI bus to control the overall speed.<br><br>Possible values are:<br><br>`SPI_CLOCK_DIV2`<br>`SPI_CLOCK_DIV4`<br>`SPI_CLOCK_DIV8`<br>`SPI_CLOCK_DIV16`<br>`SPI_CLOCK_DIV32`<br>`SPI_CLOCK_DIV64`<br>`SPI_CLOCK_DIV128`<br><br>Larger values will cause the bus to run at a slower speed. |
| Stream | A reference to a "*Stream*" (see page 365) . This is the name of the device as specified in Project Designer. |
| TEMPERATURE_TYPE | (Based on int16_t).<br>Stores a temperature in degrees. |
| TICK_COUNT | (Based on uint32_t).<br>Stores a number in μS and is used for measuring time durations.<br><br>The largest possible number is 4,294,967,295 which represents a time of just over 71 minutes. |
| uint16_t | Stores whole numbers in the range 0 to 65,535 and requires two bytes of storage. |
| uint32_t | Stores whole numbers in the range 0 to 4,294,967,295 and requires 4 bytes of storage. |
| uint8_t | Stores whole numbers in the range 0 to 255 and requires one byte of storage. |

| **Name** | **Description** |
|---|---|
| uint8_t * | Stores the address of an 'uint8_t'. |
| Vector2D | A reference to a "*Vector2D*" (see page 591) |
| Vector3D | A reference to a "*Vector3D*" (see page 596) |
| void (* callback)(TonePlayer& player) | Defines a callback for the Tone Player that is called when the current tone has finished. |
| void (*rx_func)(unsigned char, Uart&, void*) | The address of a UART receive callback function. |
| void * | Stores a reference to something - ie its start address in memory. |
| VOLTAGE_TYPE | (Based on int16_t).<br>Stores a voltage. |